

การพัฒนาตัวแรงเชิงกำหนดโดยใช้ไวยากรณ์แบบกำหนดด้วยข้อมูล

นาย เมธา กิจสวัสดิ์

สถาบันวิทยบริการ

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต

สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์

คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

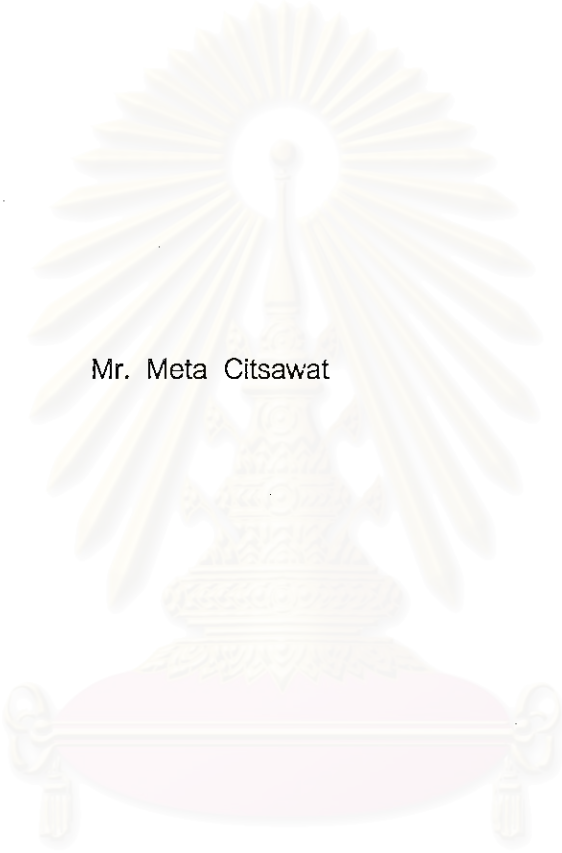
ปีการศึกษา 2544

ISBN 974-030-278-5

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

I20445416

A DEVELOPMENT OF DETERMINISTIC PARSER
USING DATA-DIRECTED GRAMMAR



Mr. Meta Citsawat

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering in Computer Engineering

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2001

ISBN 974-030-278-5

เมธา กิจสวัสดิ์ : การพัฒนาตัวแจงเชิงกำหนดโดยใช้ไวยากรณ์แบบกำหนดด้วยข้อมูล . (A DEVELOPMENT OF DETERMINISTIC PARSER USING DATA-DIRECTED GRAMMAR) อ. ที่ปรึกษา : รศ.ดร.วันชัย ธีระไพบุลย์, 95 หน้า. ISBN 974-030-278-5.

มีแนวคิดสองแบบที่แตกต่างกันในวิธีการแจงคือแนวคิดของการทำงานแบบบนลงล่าง (top-down) และการทำงานแบบล่างขึ้นบน (bottom-up) โดยที่แนวคิดทั้งสองได้สะท้อนถึงแนวคิดที่สำคัญสองอย่างคือหลักการให้เหตุผลหรือแบบกำหนดด้วยเป้าหมายซึ่งเน้นความรู้ที่มีอยู่ก่อน และหลักของการทดลอง ประสบการณ์ การสังเกตหรือแบบนำด้วยข้อมูลซึ่งเน้นข้อมูลเป็นหลัก โดยทั่วไปวิธีการแจงแต่ละแบบจะมีรูปแบบไวยากรณ์ของตัวเองรวมถึงข้อมูลเพิ่มเติมเพื่อช่วยในการแจง บางครั้งไวยากรณ์เหล่านั้นก็อาจจะถูกเขียนขึ้นในรูปแบบที่ก่อให้เกิดความกำกวมในการแจงทั้งที่ความจริงอาจไม่ได้ต้องการเช่นนั้น ทั้งนี้เพราะว่าไวยากรณ์เหล่านั้นไม่ได้เขียนโดยใช้รูปแบบที่ไม่กำกวมและมีความชัดเจนเพียงพอ

ในวิทยานิพนธ์เล่มนี้เราได้เสนอเทคนิคการแจงเชิงกำหนดจากล่างขึ้นบนแบบใหม่ที่เรียกว่า ตัวแจงแบบนำด้วยข้อมูลซึ่งจะดำเนินการในความซับซ้อนของเวลาเท่ากับ $O(n)$ และต้องการเพียงกฎไวยากรณ์ที่ถูกกำหนดขึ้นในรูปแบบไวยากรณ์แบบใหม่ที่เรียกว่า Chulalongkorn University Normal Form (CUNF) เท่านั้น CUNF ถูกพัฒนามาจาก Chomsky Normal Form (CNF) มันมีความสามารถเทียบเท่ากับไวยากรณ์ที่ไม่อิงบริบท (Context-Free Grammar) และสามารถให้นำเสนอไวยากรณ์จำนวนมากในรูปแบบที่ไม่กำกวมได้ ในวิทยานิพนธ์เรายังได้สาธิตวิธีการประยุกต์ใช้ CUNF กับไวยากรณ์ที่กำกวมประเภทต่างๆ รวมไปถึงการประยุกต์ใช้กับไวยากรณ์ของภาษาเอสทีเอ็มแอลรุ่น 3.2 ไว้ด้วย

ภาควิชา.....วิศวกรรมคอมพิวเตอร์.....ลายมือชื่อนิสิต.....
สาขาวิชา.....วิศวกรรมคอมพิวเตอร์.....ลายมือชื่ออาจารย์ที่ปรึกษา.....
ปีการศึกษา2544.....ลายมือชื่ออาจารย์ที่ปรึกษาร่วม.....

4270497021 : MAJOR COMPUTER ENGINEERING

KEY WORD: GRAMMAR / PARSER

META CITSAWAT : A DEVELOPMENT OF DETERMINISTIC PARSER USING DATA-DIRECTED PARSER . THESIS ADVISOR : ASSOC. PROF. WANCHAI RIVEPIBOON, Ph.D., 95 pp. ISBN 974-030-278-5.

There are two distinct concepts in parsing techniques; top-down and bottom-up paradigms. Both of them reflect two important insights; the rationalist tradition or goal-directed which focuses on the prior knowledge, and the empirical tradition or data-directed which focuses on the data. Generally, each parsing technique will have its own grammar formalisms including additional information to help in parsing. Sometimes, those grammars may have been written in the form that could cause ambiguous results in parsing despite in fact, they were not intended to be. This is because those grammars were not written in the form which is explicitly enough to avoid ambiguous results.

In this thesis, we present a new deterministic bottom-up parsing technique called data-directed parser which runs in time complexity equals to $O(n)$. It requires only grammar rules defined in a new grammar formalism called Chulalongkorn University Normal Form (*CUNF*) grammars. *CUNF* is derived form Chomsky Normal Form (*CNF*). It has a generative power equals to context-free grammars, and can be used to represent a large class of grammars in an unambiguous form. In the thesis, we have also demonstrated how to apply *CUNF* with various ambiguous grammars including the syntax of Hyper Text Markup Language (*HTML*) version 3.2.

Department...Computer..Engineering.... Student's signature.....*Muta Citawat*
 Field of study..Computer..Engineering... Advisor's signature.....*R. Wanchai*
 Academic year2001..... Co-advisor's signature.....

ACKNOWLEDGEMENT

First, I am grateful to Assoc. Prof. Wanchai Rievpiboon, Ph.D., my thesis advisor, for his guidance and suggestions that helped this thesis possible. Moreover, I would like to thank following people for their perspective comments and advice: Assist. Prof. Somchai Prasitjutrakul, Ph.D.; Assoc. Prof. Booncharoen Sirinaovakul, Ph.D.; Boonserm Kijisirikul, Ph.D.; and lecturer Wirote Aroonmanakun, Ph.D. I am also indebted to all of whom reviewed this thesis. Their comments and suggestions have made this a much better thesis.

Thank also to Software Engineering Laboratory and Machine Learning Laboratory, Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, which supplied me with technical information and various of the software products. Finally, the biggest thank-you of all go to my family, especially to my parents.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CONTENTS

	page
บทคัดย่อวิทยานิพนธ์.....	iv
ABSTRACT.....	v
ACKNOWLEDGEMENT.....	vi
CONTENTS.....	vii
CONTENT OF TABLES.....	x
CONTENT OF FIGURES.....	xi
1 INTRODUCTION.....	1
1.1 Background.....	1
1.2 Purpose of Research.....	2
1.3 Scope of Research.....	3
1.4 Expected Outcome.....	3
1.5 Research Plan.....	3
2 THEORY AND LITERATURE REVIEW.....	5
2.1 Grammar Formalisms.....	5
2.1.1 Unrestricted Grammars.....	5
2.1.2 Context Sensitive Grammars.....	5
2.1.3 Context-free Grammars.....	6
2.1.4 Regular Grammars.....	6
2.1.5 BNF Grammars.....	7
2.2 Parsing Techniques.....	7
2.2.1 Parser as Search.....	7
2.2.1.1 Top-Down Parsing.....	8
2.2.1.2 Bottom-Up Parsing.....	9
2.2.1.3 Comparing Top-down and Bottom-up Parsing.....	9
2.2.2 The Earley's Algorithm.....	9
2.2.3 Cocke-Younger-Kasami (CYK).....	12
2.2.4 Finite-State Parsing Methods.....	13
2.2.4.1 Transition Network (TN).....	13
2.2.4.2 Recursive Transition Network (RTN).....	14
2.2.4.3 Augmented Transition Network (ATN).....	14

2.2.4.4 Modified-Augmented Transition Network (M-ATN)	15
2.2.5 LR Parser	16
3 CHULALONGKORN UNIVERSITY NORMAL FORM GRAMMAR	18
3.1 Languages	18
3.2 Languages and Chomsky's Hierarchy of Grammars.....	19
3.3 Derivation and Languages	20
3.4 Derivation Tree	21
3.5 Goal-directed and Data-directed Paradigm	22
3.6 Chomsky Normal Form (CNF).....	23
3.7 Chulalongkorn University Normal Form (CUNF)	24
3.8 Proof of CUNF Grammars.....	26
3.9 Derivation Trees and CUNF Grammars.....	28
3.10 Converting other grammar formalisms to CUNF grammars	29
3.10.1 Context-free grammars	29
3.10.2 Regular grammars	30
3.11 Problematic Grammars and CUNF	31
3.11.1 The shift/reduce Ambiguity	31
3.11.2 The reduce/reduce ambiguity	32
3.11.3 Problem of Special-Case Productions	33
3.11.4 Problem of precedence-conflicts	35
4 DATA-DIRECTED PARSER	37
4.1 Data-Directed Parser.....	37
4.2 Tree	38
4.3 Binary Tree	39
4.4 Recursion	39
4.5 Recursive Tree Traversal.....	41
4.6 Simple Recursive Tree Construction	41
4.7 Conditional Recursive Tree Construction	43
4.8 Algorithm of Data-Directed Parser.....	45
4.9 Proving Algorithm of Data-Directed Parser	46
5 EXPERIMENT	52

5.1 Languages used in Experiment.....	52
5.1.1 Element Attributes	52
5.1.2 Structured Languages	53
5.2 CUNF Grammars for HTML	53
5.2.1 Type I Rules	54
5.2.2 Type II Rules	54
5.2.3 Type III Rules	55
5.3 Output Format	55
5.4 Error-Insensitive Solution for HTML	56
5.5 Common Parsing using Data-Directed Parser.....	57
5.6 Error-Recovery Parsing using Data-Directed Parser	59
6 CONCLUSION AND PERSPECTIVE.....	61
6.1 Conclusion.....	61
6.2 Perspective.....	64
REFERENCES	66
APPENDICES.....	69
APPENDIX A : HTML ELEMENT SPECIFICATIONS	70
APPENDIX B : CUNF GRAMMARS FOR HTML.....	83
APPENDIX C : DATA-DIRECTED PARSER USER MANUAL	93
BIOGRAPHY	95

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CONTENT OF TABLES

Page

Table 5.1 Error-Recovery Table for HTML 57



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CONTENT OF FIGURES

	Page
Figure 3.1 Goal-directed Paradigm	22
Figure 3.2 Data-directed Paradigm	23
Figure 3.3 Expression tree	29
Figure 4.1 Tree Structure	38
Figure 4.2 Linked List.....	42
Figure 4.3 Parse Tree.....	42
Figure 4.4 Input symbols represented by linked list	47
Figure 4.5 Case of one input symbol	47
Figure 4.6 Case of two input symbols.....	48
Figure 4.7 Case of two reducible symbols.....	48
Figure 4.8 Case of two irreducible symbols.....	48
Figure 4.9 Case of three input symbols	49
Figure 4.10 Case of three irreducible input symbols	49
Figure 4.11 Case of shifting right after left reduction	49
Figure 4.12 Case of shifting right after right reduction.....	50
Figure 4.13 Case of shifting self after left reduction.....	50
Figure 4.14 Case of shifting self after right reduction	50
Figure 5.1 Parse tree.....	56

CHAPTER 1

INTRODUCTION

1.1 Background

Syntactic analysis is an important part for analyzing languages generated by grammar. Due to information described by grammar which indicates the way how symbols in strings are related to each other, syntactic analysis is responsible to order those symbols and group them together into constituents to form the syntactic structures. In addition, these structures may identify type of relationships that exist between constituents and can store other information about the particular structures that may be needed for later processing in semantic analysis.

There are mainly two components in the syntactic analysis which are used to examine how the syntactic structure of a sentence can be computed. The first one is grammar which is a formal specification of the structures allowable in the language, and the second one is a parser which is the method of analyzing a sentence to determine its structure according to the grammar. Many grammar formalisms and parsing techniques have been developed which each one has its strength and weakness. The weakness of an existing one always leads to the development of a new one and this becomes the traditional style of development process. However, the strength features still exist and influence another new born researches. For example, the context-free grammar since it was proposed by (Chomsky, 1956), has influenced most of grammar formalisms including parsing techniques, and is an important tool in syntactic analysis at present.

According to this concept, many parsers have been developed such as the top-down and bottom-up parser which could require up to C^n operation to parse the sentence of length n , where C is a constant that depends on the specific algorithm used in the system. To improve the efficiency of the parser, the CYK algorithm, which was originally proposed by J.Cocke, but its first publication was due independently to (Kasami, 1965) and (Younger, 1967), was developed and can parse sentences in time complexity equals to $O(n^3)$. However, the most practical, general, context-free

recognition and parsing algorithm is (Earley, 1970) which is established for $O(n^3)$ in general, where n is the length of the sentence. Moreover, it takes only $O(n^2)$ on any unambiguous *CFG* and is actually linear on a wide variety of useful grammars. Besides, there are the algorithm of (Valiant, 1975) which is asymptotically the most efficient, taking $O(n^{2.8})$ steps, while the algorithm of (Graham et al., 1976) takes $O(n^3/\log n)$ steps. A related result, that membership for unambiguous *CFG*'s can be tested in $O(n^2)$ time, is due to (Kasami and Torii, 1969) and (Earley, 1970). Later, *GLR* parsing with time complexity equals to $O(n^3)$ is proposed by (Tomita, 1991).

However, the most famous technique used in compilers today as described in (Aho, 1986) is *LR* parser which is an efficient and bottom-up syntax analysis technique, and can run in time complexity equals to $O(n)$, where n is the length of input symbols. Due to the properties of *LR* tables, some problems in the past like shift/shift conflicts will never occur, but reduce/reduce and shift/reduce conflicts still remain. In fact, *LR* parser does not deal with number of input symbols exactly but number of states that occur in parsing process. Even though input symbols are used up, the parser continues running until the last state is reached. Referring to *LR* parser's algorithm as shown in chapter 2, we can say that time complexity of *LR* parser is sum of numbers of shift and reduce operation. Then, *LR* parser always runs more than or equals to n loops, where n is length of input symbols.

Generally, the question of concern in this thesis is that "Is it possible to develop a parser that performs better than any other parsers at present?" An ideal parser that should deal with input symbols instead of states. Moreover, there should be a way to eliminate all conflicts and parse input symbols in time complexity equals to $O(n)$ within syntactic boundary. In addition, not only parser techniques but also grammar formalisms that need to be improved in developing a better parser.

1.2 Purpose of Research

1. To develop syntactic rules and additional information called *CUNF* (Chulalongkorn University Normal Form) grammar that is an unambiguous

grammar formalism for analyzing language.

2. To develop an effective and qualitative parser that can parse language in $O(n)$ based on *CUNF* grammar.

1.3 Scope of Research

1. To develop syntactic rules that are appropriate to context-free grammars by deriving from *CNF* (Chomsky Normal Form) to *CUNF* grammar.
2. To develop an effective and qualitative parser with these qualifications.
 - Can parse input symbols correctly according to grammar generated in *CUNF*.
 - Can parse input symbols in time complexity equals to $O(n)$, where n is number of input symbols.
3. Tool used in this research is *Perl*.
4. The developed system will work and run under Windows operating system.

1.4 Expected Outcome

1. An appropriate grammar formalism called *CUNF* which is unambiguous for language generated by context-free grammar.
2. An effective and qualitative parser that can parse language generated by context-free grammar in time complexity equals to $O(n)$ based on *CUNF* grammar.

1.5 Research Plan

1. To explore problem and limitation of various grammar formalisms.
2. To explore problem and limitation of various parsers and algorithms.
3. To develop an appropriate syntax for language generated by context-free grammar.
4. To develop an appropriate methodology for converting grammar from *CFG* to the other grammar formalism called *CUNF*.
5. To develop an appropriate algorithm of parser based on information in

CUNF.

6. To design system and related component.
7. To implement and test system.
8. To evaluate and adjust system



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER 2

THEORY AND LITERATURE REVIEW

2.1 Grammar Formalisms

There are mainly four kinds of grammar formalisms which are different in generative power. A grammar has a greater generative power or be more complex than another if it can define a language that the others cannot define. It is possible to construct a hierarchy of grammars, where the set of languages described by grammars of greater power subsume the set of languages described by grammars of less power. The most common hierarchy used in computational linguistics is Chomsky's hierarchy (Chomsky, 1959), which consists of type 0 or unrestricted grammar, context-sensitive grammar, context-free grammar and regular grammar. Generally, all grammars will conform to the general definition as $G = (V, T, P, S)$, where V is a set of nonterminal symbols, T is a set of terminals symbol, P is a set of productions and S is a start symbol. However, grammars can be classified according to the form of their productions in P .

2.1.1 Unrestricted Grammars

The largest family of grammars in the Chomsky's hierarchy is unrestricted grammar which has no restrictions on the form of their rules, except that the left-hand side can not be the empty string ' ϵ '. Any string which is not string ' ϵ ' can be written as any strings including string ' ϵ '. Unrestricted grammars characterizes the recursively enumerable language. The rule are in the form $\alpha \rightarrow \beta$, where α and β are arbitrary strings of grammar symbols, with $\alpha \neq \epsilon$. These grammars are also known as semi-tue, type 0, phrase structure or unrestricted grammars.

2.1.2 Context Sensitive Grammars

Context-sensitive grammars have rules that rewrite a non-terminal symbol A in the context $\phi A \psi$ as any nonempty string. The production form of

context-sensitive grammars can be written in the form $\phi A \psi \rightarrow \phi \gamma \psi$. This form of restriction on a production prevents the right-side of productions from being empty. Besides, it can be stated in another manner. This form involves letting $\phi A \psi$ and $\phi \gamma \psi$ be expressed as $\alpha = \phi A \psi$ and $\beta = \phi \gamma \psi$ where γ must be nonempty string as given in the following definition.

$\alpha \rightarrow \beta$, where $|\alpha| \leq |\beta|$ and $|\alpha|, |\beta|$ denotes the length of α and β respectively.

2.1.3 Context-free Grammars

Context-free grammars consist of a set of rules of productions, each of which expresses the ways symbols are grouped and ordered. Context-free grammars have four parameters as

- a set of nonterminal symbols V .
- a set of terminal symbols T (disjoint from V).
- a set of productions P in the form of $A \rightarrow \gamma$, where A is a non-terminal symbol and γ is a string of symbols from the infinite set of strings $(V \cup T)^*$.
- a designated start symbol S .

In this formalism, a language is defined via the concept of derivation. One string derives another one if it can be rewritten as the second one via some series of rule applications regardless of the other symbols in its vicinity or context.

2.1.4 Regular Grammars

Regular grammars are a special case of context-free grammars which their power are equivalent to the finite state automata (FSA) and regular expression. This grammar formalism has rules in the form of:

$$A \rightarrow \gamma B \text{ or } A \rightarrow \gamma$$

Regular grammars can either be right-linear or left-linear. A rule in a right-linear grammar has a single terminal symbol on the left and at most one nonterminal symbol on the right-hand side. If there is a terminal symbol on the right-hand side, it must be the last symbol in the string. On the other hand, the right-hand-side of left-linear grammars is reversed.

2.1.5 BNF Grammars

BNF (Backus-Naur form) was developed for the syntactic definition of *ALGOL* by (Backus, 1960) as the way to express grammar formalism for programming languages. At about the same time, a similar grammatical form, the context-free grammar, was developed by linguist (Chomsky, 1959) for the definition of natural language syntax. The *BNF* and context-free grammar are equivalent in power, but the difference is in notation. The definition of *BNF* is in the form:

$$S ::= A B$$

2.2 Parsing Techniques

2.2.1 Parser as Search

In syntactic parsing, parsing can be viewed as searching through the space of all possible parse trees to find the correct parse tree for the input sentence. The search space of possible parse trees is defined by the grammar. The goal of a parsing search is to find all trees whose root is start symbol S , which cover exactly all words in the input sentence. There are two kinds of constraints that help guide the search. These two constraints give rise to the two search strategies underlying most parsers: top-down or goal-directed search and bottom-up or data-directed search. Besides, they also reflect two important insights in the western philosophical tradition: the rationalist tradition (focus on the prior knowledge) and the empirical tradition (focus on the data). Parsing as search algorithm is rather simple but it could require up to C^n operation to parse

a sentence of length n , where C is a constant that depends on the specific algorithm used in system.

2.2.1.1 Top-Down Parsing

Top-down parsers search for a parse tree by trying to build from the root node S down to the leaves. The algorithm starts by looking for grammar rules with S on the left-hand side and build all possible tree in parallel. The next step is to expand these trees as originally expanded S . At each level of search space, this algorithm uses the right-hand-sides of the rules to provide new sets of expectations for the parser and then used to recursively generate the rest of the trees. Trees are built downward until they eventually reach the part-of-speech categories at the bottom of the tree. Finally, tree whose leaves fail to match all the words in the input can be rejected, leaving behind those that represent successful. The algorithm of top-down parser was shown below.

Algorithm of Top-Down parsing algorithm

```

function Top-Down-Parser (input, grammar) return a parse tree
agenda ← (initial S tree, Beginning of input)
current-search-state ← Pop(agenda)
loop
if Successful-Parse(current-search-state) then
    return Tree(current-search-state)
else
    if Cat(Node-To-Expand(current-search-state)) is a POS then
        if Cat(node-to-expand) ⊆ Pos(Current-Input(current-search-state)) then Push
        (Apply-Lexical-Rule(current-search-state), agenda)
    else
        return reject
    else
        Push(Apply-Rules(current-search-state, grammar), agenda)
if agenda is empty then
    return reject
else
    current-search-state ← Next(agenda)
end

```

2.2.1.2 Bottom-Up Parsing

Bottom-up parsing was first suggested by (Yngve, 1959). In bottom-up parsing, the parser starts with the symbols of the input, and tries to build trees from the input symbols by applying rules from the grammar one at a time. The parser will look up each symbol and try to build partial trees. Each of the trees are then expanded if the parser can recognize places in the parse-in-progress where one of the topmost nonterminal symbols is the right hand side of some rule in the grammar. The parse is successful if the parser succeeds in building a tree with the start symbol S at root that covers all of the input.

2.2.1.3 Comparing Top-down and Bottom-up Parsing

Each of these two parsing techniques has its own advantages and disadvantages. Top-down parsers never waste time exploring trees that cannot result in a start symbol S , since they begin by generating just those trees. This means that they also never explore subtrees that cannot find a place in some S -rooted trees. In the bottom-up strategy, by contrast, trees that have no hope of leading to an S , or fitting in with any of their neighbors, are generated with wild abandon.

However, the top-down approach has its own inefficiencies as well. While it does not waste time with trees that do not lead to an S , it does spend considerable effort on S trees that are not consistent with the input. This weakness in top-down parsers arises from the fact that they can generate trees before ever examining the input. Bottom-up parsers, on the other hand, never suggest trees that are not at least locally grounded in the actual input.

2.2.2 The Earley's Algorithm

The Earley's algorithm (Earley, 1970) uses a top-down dynamic programming approach to efficiently implement a parallel dynamic programming solutions, this algorithm reduces an apparently exponential-time problem to a

polynomial-time one by eliminating the repetitive solution of sub-problems inherent in backtracking approaches. The dynamic programming approach leads to a worst-case behavior of $O(n^3)$, where n is the number of words in the input. The core of the Earley's algorithm consists of a single left-to-right pass that fills an array called a chart that has $n + 1$ entries. For each word position i in the sentence, the chart contains a lists of states representing the partial parse trees that have been generated so far. By the end of the sentence, the chart compactly encodes all the possible parses of the input. Importantly, each possible subtree is represented only once and then can be shared by all the parses that need it.

There are three operators used in Earley's algorithm to process states in the chart: the **PREDICTOR** and the **COMPLETER** that add states to the chart entry being processed, and the **SCANNER** which adds a state to the next chart entry. Each one takes a single state as input and derives new states from it. These new states are then added to the chart as long as they are not already present. For the **PREDICTOR**, its job is to create new states representing top-down expectations generated during the parsing process. It is applied to any state that has a nonterminal symbol to the right of the dot that is not a terminal symbol results in the creation of one new state for each alternative expansion of that nonterminal provided by the grammar. The second one, **SCANNER**, it is called to examine the input and incorporate a state corresponding to the predicted terminal symbol into the chart when a state has a terminal symbol to the right of the dot. This is accomplished by creating a new state from the input state with the dot advanced over the predicted input category. The last one, **COMPLETER**, is applied to a state when its dot has reached the right end of the rule. The purpose of the **COMPLETER** is to find and advance all previously created states that were looking for this grammatical category at this position in the input. New states are then created by copying the older state, advancing the dot over the expected category and installing the new state in the current chart entry. Algorithm of Earley's parsing was shown in section below.

Algorithm of Earley's parsing

```

function Earley-Parse(words, grammar) return chart
  Enqueue( $\gamma \rightarrow \cdot S$ , [0,0]), chart[0])
  for  $i \leftarrow$  from 0 to Length(words) do
    for each state in chart[i] do
      if Incomplete(state) and Next-Cat(state) is not a part of speech then
        Predictor(state)
      else if Incomplete(state) and Next-Cat(state) is a part of speech then
        Scanner(state)
      else
        Completer(state)
    end
  end
  return(chart)

  procedure Predictor( $(A \rightarrow \alpha \cdot B \beta$ , [i,j])
    for each ( $B \rightarrow \gamma$ ) in Grammar-Rule-For(B, grammar) do
      Enqueue( $(B \rightarrow \cdot \gamma$ , [j, j]), chart[j])
    end

  procedure Scanner( $(A \rightarrow \alpha \cdot B \beta$ , [i,j])
    if  $B \subset$  Parts-Of-Speech(word[j]) then
      Enqueue( $(B \rightarrow word[j]$ , [j, j + 1]), chart[j + 1])

  procedure Completer( $(B \rightarrow \gamma \cdot$ , [j, k])
    for each ( $A \rightarrow \alpha \cdot B \beta$ , [i, j]) in chart[j] do
      Enqueue( $A \rightarrow \alpha B \cdot \beta$ , [i, k]), chart[k])
    end

  procedure Enqueue(state, chart-entry)
    if state is not already in chart-entry then
      Push(state, chart-entry)
    end
  
```

The version of the Earley's algorithm just described above is actually a recognizer not a parser. After processing, valid sentences will leave the state $S \rightarrow \alpha \cdot, [0, N]$ in the chart. To turn this algorithm into a parser, we must be able to extract individual parses from the chart. To do this, the representation of each state must be augmented with an additional field to store information about the completed states that generated its constituents. This information can be gathered by making a simple change to the **COMPLETER**. Recall that the constituent following the dot is discovered. The only change necessary is to have **COMPLETER** push a pointer to the newly discovered state onto the list of children of the new state. Retrieving a parse tree from the chart is then merely a recursive retrieval starting with the states representing a complete S in the final chart entry.

2.2.3 Cocke-Younger-Kasami (CYK)

The CYK algorithm (Kasami, 1965), (Younger, 1967) uses a bottom-up dynamic programming approach to efficiently implement a parallel dynamic programming solutions as Earley's algorithm. This algorithm reduces an apparently exponential-time problem to a polynomial-time one when given a sentence of length n and a grammar G , which is in Chomsky Normal Form. That is all productions used in this parsing technique must have one or two symbols on the right side of each rule. Unlike Early's algorithm, the grammar rules used in algorithm are not restricted. By eliminating the repetitive solution of sub-problems inherent in backtracking approaches, the dynamic programming approach leads to a worst-case behavior of $O(n^3)$, where n is the number of words in the input.

The core of the CYK algorithm consists of a single left-to-right pass that fills an array called a chart that has n entries. For each cell in chart position V_{ij} , the cell contains a list of corresponding left side of product from a list in the upper cells and diagonal cells extending from V_{ij} to the right. Algorithm of CYK' parsing was shown below.

Algorithm of CYK parsing algorithm

```

function CYK-Parse(words, grammar) returns chart
begin
for  $i := 1$  to  $n$  do
   $V(i, j) := \{A \mid A \rightarrow a \text{ is a production and the } i\text{th symbol of } x \text{ is } a\}$ ;
for  $j := 2$  to  $n$  do
  for  $i := 1$  to  $n - j + 1$  do
    begin
       $V(i, j) := \text{empty}$ ;
      for  $k := 1$  to  $j - 1$  do
         $V(i, j) := V(i, j) \cup \{A \mid A \rightarrow BC \text{ is a production, } B \text{ is in } V(i, k) \text{ and } C \text{ is in } V(i+k, j-k)\}$ 
      end
    end
  end
end

```

2.2.4 Finite-State Parsing Methods

From the basis of graph theory, network is composed of nodes and arcs, which each node has unique name and linked together by arcs or groups of arcs which have specific conditions. Many parsing techniques were developed based on those concepts such as Transition Network, Recursive Transition Network, Augmented Transition Network and Modified-Augmented Transition Network.

2.2.4.1 Transition Network (TN)

Transition network as described in (Gilbert, 1999) consists of nodes or group of nodes combined with arcs or links for defining the transition from one state to another following to conditions of grammar. It can be defined with the definition of a nondeterministic transition network, M , which is a 5-tuple with $M = (Q, \Sigma, K, q_0, F)$, where

- Q is a finite set of nodes or states;
- Σ is a finite set of acceptable input symbols;
- K is a finite set of rules representing the state transition function or transitions from one state to another;

- q_0 in Q is the initial state in the network; and
- F is the set of accepting or final states.

The problem of transition network is complexity of network when the conditions according to syntax increase which leads to a development of the Recursive Transition Network.

2.2.4.2 Recursive Transition Network (RTN)

Recursive transition network as described in (Allen, 1995) and (Gilbert, 1999), has a structure and style like transition network which has nodes, arcs and conditions following to grammar. It can be defined as a 7-tuple with $M = (Q, \Sigma, \Gamma, K, S_0, Z_0, F)$, where

- Q is the set of states;
- Σ is the set of terminal symbols;
- Γ is the set of symbols that may appear on the stack used as an auxiliary form of control;
- K is the set of moves or rules that control the transitions from one state to the next;
- S_0 is the initial state;
- Z_0 is a unique symbol that serves to initialize the stack; and
- F is the set of accepting states.

The *RTN* is flexible to edit or create a new network, but the problem is lacking ability of remembering whether a node has already been examined. When node does not have the suitable condition, network will backtrack 2 nodes and verify condition of arc again which wastes time. Besides, *RTN* cannot support complex sentences. This drawback leads to a development of the Augmented Transition Network.

2.2.4.3 Augmented Transition Network (ATN)

ATN as described in (Gilbert, 1999), works like *TN* and *RTN* but has more conditions of node movement. It registers each process of

analysis, and its definition can be defined as an 8-tuple as $A = (Q, \Sigma, \Gamma, T, O, R, q_0, Z_0)$, where

- Q is a finite set of state symbols representing the possible states;
- Σ is a finite set of input symbol;
- Γ is a finite alphabet of symbols;
- T is the set of transition rules that will subsequently be defined;
- O is a finite set of operations for storing information in a finite system of registers;
- R is a finite set of operations for reading information stored in one or more register;
- q_0 is the initial state in the primary network; and
- Z_0 is a single finite state

Although *ATN* is very efficiency, some part of networks cannot specify the correct function of word in sentence such as arc that displays function of verb which leads to a development of the Modified-Augmented Transition Network.

2.2.4.4 Modified-Augmented Transition Network (M-ATN)

M-ATN has an analysis style form left to right and from top to down which has the method of verifying arc, backtracking and remembering step of working like *ATN*. Besides, *M-ATN* has 3 basises arc.

- Seek Arc (S/ name node) is the arc that jumps to a specific arc.
- Jump Arc is the arc that jumps to a next node or a specific node.
- Send Arc is the arc that indicates the end of network or sub network.

The advantage of *M-ATN* is ability to find the main verb of sentence and its position which leads to the correct deep structure of

language as described in (Varakulsiripunth, 1988) and (Varakulsiripunth, 1989).

2.2.5 LR Parser

LR parser is the most famous technique used in compiler today as described in (Aho, 1986) which is an efficient, bottom-up syntax analysis technique that can be used to parse a large class of context-free grammars. Its time complexity equals to $O(n)$, where n is the length of input sentence. This technique is called *LR(k)* parsing; the *L* is for left-to-right scanning of input, the *R* for construction a rightmost derivation in reverse, and the *k* for the number of input symbols of lookahead that are used in making parsing decisions. When (*k*) is omitted, *k* is assumed to be 1.

The schematic form of *LR* parser consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (*action* and *goto*). The driver program is the same for all *LR* parsers; only the parsing table is different. There are three techniques for constructing *LR* parsing table for a grammar. The first method, called simple *LR (SLR)*, is the easiest to implement, but the least powerful of the three. It may fail to produce a parsing table for certain grammars on which the other methods succeed. The second method, called canonical *LR*, is the most powerful, and the most expensive, The third method, called lookahead *LR (LALR)*, is intermediate in power and cost between the other two. Algorithm of *LR* parser was shown below.

Algorithm of LR parser

set *ip* to point to the first symbol of *w*;

repeat forever begin

 let *s* be the state on top of the stack and *a* the symbol pointed to by *ip*;

 if *action*[*s*, *a*] = shift *s'* then

 begin

 push *a* then *s'* on top of the stack;

 advance *ip* to the next input symbol

 end

```

else if  $action[s, a] = \text{reduce } A \rightarrow B$  then
begin
    pop  $2 * |B|$  symbols off the stack;
    let  $s'$  be the state now on top of the stack;
    push  $A$  then  $goto[s', A]$  on top of the stack;
    output the production  $A \rightarrow B$ 
end
else if  $action[s, a] = \text{accept}$  then
    return
else
    error()
end

```

From algorithm of *LR* parser described above, even though all input symbols are used up, *LR* parser still continues running until the last state is reached. Although *LR* parser's time complexity equals to $O(n)$ where n is a number of input symbols, it does not mean that numbers of looping occur in parsing will equal to n . In fact, the time complexity of *LR* parser always depends on states while those states are dependent to grammar rules.

Starting with some syntactic rules, for *LR* parser, we must derive sets of *LR* items for augmented grammar and construct *LR* table respectively. Then, we need grammar, *LR* items and *LR* table to parse input strings with *LR* parser. Moreover, editing or changing grammars will always lead to rebuilding *LR* items and *LR* table. Even though someone will argue that these two processes can be done automatically by computer, we often encounter with ambiguous grammar causing shift/reduce conflicts and reduce/reduce conflicts that computer cannot solve because of lacking appropriate information, and require human to solve them. In addition, these also mean that only syntactic rules are not enough in some case. Besides, *LR* parser is difficult to write unless we have a tool such as yacc or occs to help us construct the tables used by parser. Also, *LR* error recovery is difficult to do in yacc-generated parsers.

CHAPTER 3

CHULALONGKORN UNIVERSITY NORMAL FORM GRAMMAR

3.1 Languages

A language is made up of sentences that can be used in communication. Nevertheless, not any sequence of words could be characterized as a sentence; only some of which we would judge as grammatical. Whenever we judge which sentence is acceptable, what knowledge do we use as the basis for our judgment? One possible approach to our judgements about sentences is based upon past knowledge, assume that we had accumulated a dictionary-like list of sentences. Then, the set of grammatical or well-formed sentences could be defined as a list of facts, like words in a dictionary, and the process of using that knowledge would be by exhaustive search through that list. However, the list would have to be very long and might easily include several billion entries. More important, the process of searching through that list would be time consuming at best. If this assumption is correct, what about new sentences that we had never seen before? How could we possibly conclude that those also might be grammatical?

In this regard, a line of argument was emphasized by (Chomsky, 1957) with his observation that our knowledge of syntax is *productive*. Chomsky's approach to this dilemma was to shift from an emphasis on the *language* that for all practical purposes includes an infinite number of members to an emphasis on the *grammar* or *rules* that could generate the language. By definition, the grammar must be finite: therefore, our knowledge, as incorporated in the grammar, must also be finite. It is the process of using this knowledge that will be *productive*. In effect, the grammar can define an infinite set of possibilities, including many that have never been encountered before.

However, not any sequences of words are grammatical would have meaning. For example, we can define grammar rules for "1a2b3" and "4b5a7", and we can say that both of these are grammatical, but we can not tell what they mean. We cannot also

answer question like “what is the difference between both of them except the symbols and their order?” Let us change ‘a’ to ‘+’ and ‘b’ to ‘*’, we can define grammar rules for “1+2*3” and “4*5+7” as well. Moreover, we can tell that “1+2*3” equals to “7”, “4*5+7” equals to ‘27’, and “4*5+7” is greater than “1+2*3”. Hence, we should be careful to think that sentence which is grammatical according to defined grammar is really acceptable. For sometimes, nobody even grammar writer knows the meaning of language generated by his grammars. In such case, what we can discuss is only in level of weak equivalence or symbols and order of them; set of possible languages that could be generated by a grammar.

3.2 Languages and Chomsky’s Hierarchy of Grammars

Due to Chomsky’s hierarchy described in Chapter 2, we can classify grammars which conform to the general definition as $G = (V, T, P, S)$ by the form of their productions in P . Referring to Chomsky’s hierarchy, there are four classes of grammars; type 0 or unrestricted grammars, context-sensitive grammars, context-free grammars and regular grammars respectively. A grammar that has a greater generative power can define a language that the less one cannot.

Generally, any communication is composed of 2 agents; a sender who generates languages by his grammars and a receiver who receives the sentences and tries to understand them by using his own grammars. The sentences we got, are in sentential forms or sequence of words, not in structural forms or parse trees. We do not know even what grammar generates such languages. In order to understand such languages, our own grammar rules must be generated first. For, we do not know which grammar that the sender uses in generating such language, so it is possible that our grammar may differ from sender’s. Due to concept of Chomsky’s hierarchy, we can claim that if we understand language of the sender, our grammar should be greater generative power or equals to the sender’s.

These assumptions remind us two important insights in the western philosophical tradition: the rationalist tradition or goal-directed (focus on the prior knowledge) and the

empirical tradition or data-directed (focus on the data). To understand the difference between these two paradigms, let us consider the way that we try to write a sentence and the way we try to read it. In reading a sentence, we have to see all words in sentence before we know its meaning, while in writing a sentence, we need to know the meaning we want to write first. Then, we begin to write each word to form a sentence.

According to Chomsky's hierarchy, it is possible for grammars of greater power to describe languages that subsume set of languages generated by grammars of less power. From this view, we can say that whoever can understand our language must have the grammar that is equal or more powerful than ours. From such assumption, we will design a new grammar formalism that based on data-directed paradigm which differs from grammar formalisms based on goal-directed paradigm.

3.3 Derivation and Languages

Due to Chomsky's observation that our knowledge of syntax is productive, the languages are formally defined by a grammar $G = (V, T, P, S)$, where V is a set of non-terminal symbol, T is a set of terminal symbol, P is a set of productions and S is a start symbol respectively. As described in (John and Jeffrey, 1979), we can develop notation to represent a derivation. First we define two relations \Rightarrow_G and \Rightarrow^*_G between strings in $(V \cup T)^*$. If $A \rightarrow \beta$ is a production of P and α and γ are any strings in $(V \cup T)^*$, then $\alpha A \gamma \Rightarrow_G \alpha \beta \gamma$. We say that the production $A \rightarrow \beta$ is applied to the string $\alpha A \gamma$ to obtain $\alpha \beta \gamma$ or $\alpha A \gamma$ that directly derives $\alpha \beta \gamma$ in grammar G . Two strings are related by \Rightarrow_G exactly when the second is obtained from the first by one application of some production.

Suppose that $\alpha_1, \alpha_2, \dots, \alpha_m$ are strings in $(V \cup T)^*$, $m \geq 1$, and

$$\alpha_1 \Rightarrow_G \alpha_2, \alpha_2 \Rightarrow_G \alpha_3, \dots, \alpha_{m-1} \Rightarrow_G \alpha_m$$

Then we say $\alpha_1 \Rightarrow^*_G \alpha_m$ or α_1 derives α_m in grammar G . That is, \Rightarrow^*_G is the reflexive and transitive closure of \Rightarrow_G . Alternatively, $\alpha_1 \Rightarrow^*_G \beta$ if β follows from α_1 by application of zero or more productions of P . Note that $\alpha \Rightarrow^*_G \alpha$ for each string α .

Usually, if it is clear which grammar G is involved, we use \Rightarrow for \Rightarrow_G and \Rightarrow^* for \Rightarrow^*_G . If α derives β by exactly i steps, we say $\alpha \Rightarrow^i \beta$.

The language generated by G [denoted $L(G)$] is $\{w | w \text{ is in } T^* \text{ and } S \Rightarrow^*_G w\}$.

That is, a string is in $L(G)$ if:

- 1) The string consists solely of terminals.
- 2) The string can be derived from S .

We call L a context-free language (CFL) if it is $L(G)$ for some CFG G . A string of terminals and variables α is called a sentential form if $S \Rightarrow^* \alpha$. We define grammars G_1 and G_2 to be equivalent if $L(G_1) = L(G_2)$.

3.4 Derivation Tree

It is useful to display derivations as trees. These pictures called derivation or parse trees superimpose a structure on the words of a language that is useful in applications such as the compilation of programming languages. The vertices of a derivation tree are labeled with terminal or variable symbols of the grammar or possibly with ϵ . If an interior vertex n is labeled A , and the sons of n are labeled X_1, X_2, \dots, X_k from the left, then $A \rightarrow X_1 X_2 \dots X_k$ must be a production. Figure 3.1 shows the parse tree for derivation, and if we read the leaves, in left-to-right order, we get the last line of "E + E * E". More formally, let $G = (V, T, P, S)$ be a CFG. A tree is a derivation or parse tree for G if:

- 1) Every vertex has a label, which is a symbol of $V \cup T \cup \{\epsilon\}$.
- 2) The label of the root is S .
- 3) If a vertex is interior and has label A , then A must be in V .
- 4) If n has label A and vertices n_1, n_2, \dots, n_k are the sons of vertex n , in order from the left, with labels X_1, X_2, \dots, X_k , respectively, then $A \rightarrow X_1 X_2 \dots X_k$ must be a production in P .
- 5) If vertex n has label ϵ , then n is a leaf and is the only son of its father.

We need one additional concept, that of a subtree. A subtree of a derivation tree is a particular vertex of the tree together with all its descendants, the edges connecting them, and their labels. It looks just like a derivation tree, except that the label of the root may not be the start symbol of the grammar. If variable A labels the root, then we call the subtree an A -tree. Thus "S-tree" is a synonym for "derivation tree" if S is the start symbol.

3.5 Goal-directed and Data-directed Paradigm

Data-directed grammar is a grammar defined by a receiver in order to understand languages generated by sender's goal-directed grammar. While The rationalist tradition or goal-directed grammars focus on the prior knowledge or set of product rules in order to generate languages, the empirical tradition or data-directed grammars focus on the data or input symbols in order to generate parse tree. For the difference of these two paradigms, let us consider context-free grammar $G = (V, T, P, S)$ is the grammar of sender, where P consists of

$$E \rightarrow E + E \mid E * E \mid id$$

For the strings "1 + 2 * 3", in goal-directed paradigm when scan input from left to right, we must look backward to the start symbols in grammar and start derivation from there until we have all external nodes of parse tree that match our input symbols as in figure 3.1. On the other hand, in data-directed paradigm, we start from input symbols and try to construct a parse tree as in figure 3.2. Although this sequence of input symbols generated by context-free grammar, language itself is context-sensitive from data-directed view because we cannot reduce "1 + 2" before "2 * 3", and this is the difference between goal-directed and data-directed paradigm.

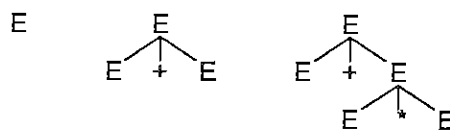


Figure 3.1 Goal-directed Paradigm

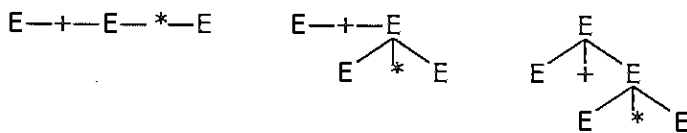


Figure 3.2 Data-directed Paradigm

Generally, we could say that whatever languages that could be written in the form of binary tree can be rewritten in *CUNF* grammar. Referring to Chomsky Normal Form, we could rewrite grammar G_1 to G_2 where languages generated by G_2 are equivalent to languages generated by G_1 . Before going into *CUNF* grammar, it should be useful to review Chomsky Normal Form.

3.6 Chomsky Normal Form (CNF)

Chomsky Normal Form is one of two normal-form theorem which states that all context-free grammars are equivalent to grammars with restrictions on the forms of productions. From the theorem of Chomsky normal form (*CNF*) as described in (John and Jeffrey, 1979), any context-free language without ϵ is generated by a grammar in which all productions are of the form $A \rightarrow BC$ or $A \rightarrow a$, where, A , B and C , are variable and a is a terminal.

Suppose that G be a context-free grammar generating a language not containing ϵ . We can find an equivalent grammar, $G_1 = (V, T, P, S)$, such that P contains no unit productions or ϵ -productions. Thus, if a production has a single symbol on the right, that symbol is a terminal, and the production is already in an acceptable form. Now consider a production in P , of the form $A \rightarrow X_1 X_2 \dots X_m$, where $m \geq 2$. If X_i is a terminal symbol, a , introduce a new variable C_a and a production $C_a \rightarrow a$, which is of an allowable form. Then replace X_i by C_a . Let the new set of variables be V' and the new set of productions be P' . Consider the grammar $G_2 = (V', T, P', S)$. If $\alpha \Rightarrow_{G_1} \beta$, then $\alpha \Rightarrow_{G_2} \beta$. Thus $L(G_1) \subseteq L(G_2)$. Now we show by induction on the number of steps in a derivation that if $A \Rightarrow_{G_2} w$, for A in V and w in T^* , then $A \Rightarrow_{G_1} w$. The result is trivial

for one-step derivations. Suppose that it is true for derivations of up to k steps. Let $A \Rightarrow_{G_2}^* w$ be a $(k + 1)$ - step derivation. The first step must be of the form $A \rightarrow B_1 B_2 \dots B_m$, $m \geq 2$. we can write $w = w_1 w_2 \dots w_m$, where $B_i \Rightarrow_{G_2}^* w_i$, $1 \leq i \leq m$.

If B_i is C_{a_i} for some terminal a_i , then w_i must be a_i . By the construction of P' , there is a production $A \rightarrow X_1 X_2 \dots X_m$ of P where $X_i = B_i$ if B_i is in V and $X_i = a_i$ if B_i is in $V' - V$. For those B_i in V , we know that the derivation $B_i \Rightarrow_{G_2}^* w_i$ takes no more than k steps, so by the inductive hypothesis, $X_i \Rightarrow_{G_2}^* w_i$. Hence $A \Rightarrow_{G_2}^* w$. Now, We have proved the intermediate result that any context-free language can be generated by a grammar for which every production is either of the form $A \rightarrow a$ or $A \rightarrow B_1 B_2 \dots B_m$, for $m \geq 1$, where A and B_1, B_2, \dots, B_m are variables, and a is a terminal. Consider such a grammar $G_2 = (V', T, P', S)$. We modify G_2 by adding some additional symbols to V' and replacing some productions of P' . For each production $A \rightarrow B_1 B_2 \dots B_m$ of P' , where $m \geq 3$, we create new variables D_1, D_2, \dots, D_{m-2} and replace $A \rightarrow B_1 B_2 \dots B_m$ by the set of productions

$$\{A \rightarrow B_1 D_1, D_1 \rightarrow B_2 D_2, \dots, D_{m-3} \rightarrow B_{m-2} D_{m-2}, D_{m-2} \rightarrow B_{m-1} B_m\}$$

3.7 Chulalongkorn University Normal Form (CUNF)

CUNF is denoted by $G = (V, T, P, R, S, \$)$ where both V and T are finite sets of variables and terminals (without empty string) respectively when V and T are disjoint. P is a finite set of productions; each production is of the form either $S \rightarrow A\$$ or $A \rightarrow BC$ or $A \rightarrow a$ where $A, B,$ and C are the string of symbols from V and a is a string of symbol from T . R is a finite set of rules describing the priority of productions in P . S is a special variable called the start symbol and $\$$; a special variable called the end symbol. For example, let us consider a context-free grammar $G = (V, T, P, S)$ as $G(A) = (\{E\}, \{+, *, id\}, P, E)$ where P consists of

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow id$$

We can generate grammar G in CNF as

$$E \rightarrow EE' | EE''$$

$$E' \rightarrow + E$$

$$E'' \rightarrow * E$$

$$E \rightarrow \text{id}$$

From *CNF* we could generate *CUNF* as $G = (\{A, B, C, D, E, F, \text{plus}, \text{mul}, \text{id}\}, \{+, *, P, R, S, \$\})$ where P consists of

$$S \rightarrow \text{id } \$$$

$$A \rightarrow \text{plus id}$$

$$B \rightarrow \text{mul id}$$

$$\text{plus} \rightarrow +$$

$$\text{mul} \rightarrow *$$

$$\text{id} \rightarrow \text{digit} \mid \text{id } A \mid \text{id } B$$

and R consists of

$$\{ \text{plus id} < \text{id } A, \text{plus id} < \text{id } B, \text{mul id} > \text{id } A, \text{mul id} < \text{id } B \}$$

Generally, set P could be viewed as 2 parts; P' and P'' where $P = P' \cup P''$. For P' , it collects production in form $A \rightarrow BC$, where A, B, C are non-terminal symbols and are elements in set V . P' will be used in reduction steps or syntactic analysis while P'' collects productions of form $A \rightarrow a$, where A is a non-terminal symbol and a is a terminal symbol. P'' will be used in lexical analysis. For set R or set of priority rules, it accumulates any pairs of product rules that overlaps with each another such as $A \rightarrow BC$ and $E \rightarrow CD$. We will see that both of the rules share the same symbol C , and must add an appropriate priority rule like $BC > CD$ to set R if we prefer the first rule for a particular parse tree. Although *CUNF* is rather complex compared to other grammar formalism, it will never fail during parsing. Besides, using data-directed parser designed for *CUNF*, we can control every process of parsing during construction of the parse tree. Moreover, its time complexity equals to $O(n)$, and parser will stop running when input symbols are used up, unlike *LR* parser which deals with states; data-directed parser deals with input symbols only.

Many problematic grammars could be represented by *CUNF* which is unambiguous and more powerful. We can define many ambiguous grammars in unambiguous form using *CUNF* as we will see in later section for its power to define

unambiguous grammar. Even though *CUNF* grammar may look like deterministic grammar in which it produces a specific parse tree for the same input symbols, we can apply it to generate non-deterministic results as well. By considering set R or the priority set as the choices that parser must choose either left or right direction. Each times we access set R and find the same item $X_1X_2X_3$ there, we will have 2 possible patterns of parse trees that could be built from the current one. Suppose that we access set R k times, a number of parse trees that would be produces will equal to 2 power k , where the maximum k equal to $n-2$ when n is a number of all symbols. So, the worst case for this process would be $O(2^n)$. However, this boundary cannot be compared with dynamic programming technique's which always require $O(n^3)$ because the value of k is uncertain. In practical, n or $n-2$ differs from k for it depends on numbers of input symbols while value of k depends on grammars especially set R .

3.8 Proof of *CUNF* Grammars

Due to the definition of *CUNF* grammars which have the restriction of numbers of symbols on the right side of product rules described in previous section, we will see that *CUNF* grammar looks like context-free grammar defined in Chomsky Normal Form (*CNF*), except the symbol $\$$ or end marker for input symbols including set R or context-sensitive rules. Proving of *CUNF* will be divided into 2 parts. First, we will prove that it is equivalent to *CNF* in reverse order or bottom-up direction. Next, we will prove that with this non-deterministic form and information defined in context-sensitive rules, it can be used as a deterministic grammar for a specific purpose as well.

In part one, to prove that *CUNF* equivalents to *CNF* we will refer to *CNF* which states that any context-free language without empty string can be generated by a grammar in which all productions are of the form $A \rightarrow BC$ or $A \rightarrow a$, where A, B, C are variables and a is a terminal. Then, if we have productions in P of the form $A \rightarrow X_1 X_2 \dots X_m$ where $m \geq 2$, we can replace them in form $A \rightarrow X_1 D_1, D_1 \rightarrow X_2 D_2, \dots D_{m-3} \rightarrow X_{m-2} D_{m-2}, D_{m-2} \rightarrow X_{m-1} X_m$ which are equivalent. From this concept, suppose that G is context-free grammar not containing empty string as $G = (V, T, P, S)$. According to Chomsky Normal Form theorem, we can define grammar $G_2 = (V, T, P', S)$ in *CNF* which is equivalent to

grammar G . Then we define grammar $G_3 = (V', T, P', S, \$)$ following to concept of *CUNF* grammar. To define grammar G_3 from G_2 , we will add augmented rules such as $S \rightarrow S^* \$$ in P' where S^* is start symbol S in *CNF* and $S^* \rightarrow X Y$ is in P' . Besides, we need to add the rule $S \rightarrow X \$$ in P' where in *CNF*, X could be the reduced to start symbol as well. Refer to *CNF*, now we will have rules defined in P' as

$S \rightarrow A \$$ and $A \rightarrow X_1 D_1, D_1 \rightarrow X_2 D_2, \dots, D_{m-3} \rightarrow X_{m-2} D_{m-2}, D_{m-2} \rightarrow X_{m-1} X_m$, where all productions are unique subtrees of a parse trees according to grammar. From Chomsky Normal Form theory, we will see that $L(G) \subset L(G_2)$ and $L(G_2) \subset L(G_3)$. Then we can conclude that $L(G) \subset L(G_3)$.

For the second part, we will prove that *CUNF* could be use as a deterministic context-free grammar using context-sensitive rules or set R which defined in *CUNF* as grammar $G = (V, T, P, R, S, \$)$. From sequence of terminal symbols $w_1 w_2 w_3 \dots w_m$ generated by grammar $G = (V, T, P, S)$, it is possible to parse them from bottom-up direction in $O(n)$ using only information defined in *CUNF* grammar.

Following to grammar G , any terminal symbol w_i where $X_i \rightarrow w_i$ could be reduced to X_i . So, we will have sequence of non-terminal symbols $X_1 X_2 X_3 \dots X_m$ instead. For any non-terminal symbol X_i where $X_{i-1} X_i X_{i+1}$ is substring of $X_1 X_2 X_3 \dots X_m$, either $X_{i-1} X_i$ or $X_i X_{i+1}$ could be reduce in deterministic grammar. However, In *CNF*, it is possible for two productions that could share either left or right symbol of their right side of productions. For example, we may have the rules both $A \rightarrow X_1 X_2$ and $B \rightarrow X_2 X_3$ for the input symbols like $X_1 X_2 X_3$. But, it not clear whether A or B is preferred and result in the different parse tree. In such case, $X_1 X_2 X_3$ will be added in context-sensitive rules with its priority defined as 'L' for left direction or 'R' for right direction. In syntactic analysis, we have 2 choices each times whenever the overlap occurring between two rules is found. Without context-sensitive rules, suppose that the parser finds these overlaps k times. it means that parser will generate 2^k parse trees as the results. By using context-sensitive rules, we can retrieve only one parse tree which is grammatical following to grammar G at the end because 1 power k equals to 1.

3.9 Derivation Trees and CUNF Grammars

CUNF grammar is a data-directed grammar. It differs from other grammar formalisms in which it concentrates on input symbols and relation among them in sentential form. As we know that some context-free grammar will produce language such as "1+2*3" which cannot be reduced freely, but depends upon precedence and associativity.

In fact, *CUNF* is a model of whole derivation trees in formalism of grammars defined as $G = (V, T, P, R, S, \$)$. Each rule in P is a unique subtree in derivation trees, and priorities among those rules control the appearance of parse trees. However, priorities of rules are not something that just happened in *CUNF* grammar. Its existence remains in other grammar formalisms as well. Let us consider grammar $G = (V, T, P, S)$ and its productions as

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow \text{id}$$

Suppose that we are going to parse "id + id * id". When the parser scans syntax rules downwards, and it get $E \rightarrow E + E$ first, next is $E \rightarrow E * E$, and $E \rightarrow \text{id}$ respectively. Then we will have the parse tree as figure 3.3 a). On the other hand, if we define grammar $G = (V, T, P, S)$ -as

$$E \rightarrow E * E$$

$$E \rightarrow E + E$$

$$E \rightarrow \text{id}$$

When parser scans syntactic rules downwards with productions $E \rightarrow E * E$, $E \rightarrow E + E$, and $E \rightarrow \text{id}$ respectively, it produces the parse tree as shown in figure 3.3 b). This parse tree and the prior one is not the same, even though both of them are weak equivalence. Main cause of this problem is the way that we define grammars, and the way that parser reads them. The rule which parser finds first in parsing is always less important than the next downward one.

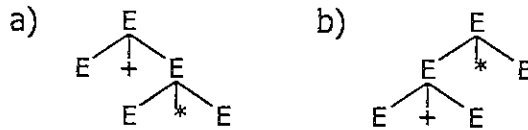


Figure 3.3 Expression tree

Not only precedence plays an important role in grammar formalisms, but it also may cause a problem in parsing if we are careless. The existence of context-sensitive rules or set R defined in *CUNF* grammar is designed for this problem. To solve this problem, we need to break the way parser scans the grammar rules downwards. In order to do this, we have to break the way that grammars are written first. Then, *CUNF* grammar was designed with a separated set of priority among productions.

3.10 Converting other grammar formalisms to *CUNF* grammars

3.10.1 Context-free grammars

A context-free grammar (*CFG*) is denoted as $G = (V, T, P, S)$, where V and T are a finite sets of variables and terminals, respectively. We assume that V and T are disjoint. P is a finite set of productions; each production is of the form $A \rightarrow \alpha$, where A is a variable and α is a string symbol from $(V \cup T)^*$. Finally, S is a special variable called the start symbol. For example, suppose that we use E instead of <expression> for the variable in the grammar. Then we could formally express this grammar as $(\{E\}, \{+, *, (,), id\}, P, E)$, where P consists of

$$E \rightarrow E + E$$

$$E \rightarrow E * E$$

$$E \rightarrow (E)$$

$$E \rightarrow id$$

From *CFG* shown above we could generate *CUNF* grammar as $G = (\{A, B, C, plus, mul, id, lp, rp\}, \{+, *, (,)\}, P, R, S, \$)$ where P consists of

$$S \rightarrow id \$$$

$$A \rightarrow plus id$$

$$B \rightarrow mul id$$

$$\begin{aligned}
 C &\rightarrow \text{id rp} \\
 \text{plus} &\rightarrow + \\
 \text{mul} &\rightarrow * \\
 \text{lp} &\rightarrow (\\
 \text{rp} &\rightarrow) \\
 \text{id} &\rightarrow \text{digit} \mid \text{id A} \mid \text{id B} \mid \text{lp C}
 \end{aligned}$$

and R consists of

$$\{ \text{plus id} > \text{id rp}, \text{mul id} > \text{id rp}, \text{plus id} < \text{id A}, \text{plus id} < \text{id B}, \\
 \text{mul id} > \text{id A}, \text{mul id} < \text{id B} \}$$

3.10.2 Regular grammars

Refer to Chomsky's hierarchy, CFG is more powerful than regular grammar. That means we can rewrite regular grammar in $CUNF$ grammar as well. If all productions of a CFG are of the form $A \rightarrow wB$ or $A \rightarrow w$, where A and B are variables and w is a (possibly empty) string of terminals, then we say the grammar is right-linear. If all productions are of the form $A \rightarrow Bw$ or $A \rightarrow w$, we call it left-linear. A right-linear or left-linear grammar is called a regular grammar. For example, the language $0(10)^*$ is generated by the right-linear grammar

$$\begin{aligned}
 S &\rightarrow 0A \\
 A &\rightarrow 10A \mid \epsilon
 \end{aligned}$$

and by the left-linear grammar

$$S \rightarrow S10 \mid 0$$

From the definition of grammar shown above we can rewrite it into $CUNF$ grammar as $G = (\{A, B, C\}, \{0, 1\}, P, R, S, \$)$ where P is set of products as

$$\begin{aligned}
 S &\rightarrow A\$ \\
 B &\rightarrow CA \\
 A &\rightarrow 0 \mid AB \\
 C &\rightarrow 1
 \end{aligned}$$

and R which is set of context-sensitive rules as

$$\{ CA > A \$, CA < AB \}$$

3.11 Problematic Grammars and CUNF

There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration which the parser, knowing the entire stack contents and the next input symbol, cannot decide whether to shift or to reduce (a shift/reduce conflict), or cannot decide which of several reductions to make (a reduce/reduce conflict). We now give some examples of syntactic constructs that give rise to such grammars. Technically, these grammars are not in the $LR(k)$ class of grammars; we refer to them as non-LR grammars. The k in $LR(k)$ refers to the number of symbols of lookahead on the input. Grammars used in compiling usually fall in the $LR(1)$ class, with one symbol lookahead.

3.11.1 The shift/reduce Ambiguity

An ambiguous grammar can never be LR . For example, consider the dangling-else grammar

$$\begin{aligned} stmt &\rightarrow \text{if } expr \text{ then } stmt \\ &\quad | \text{if } expr \text{ then } stmt \text{ else } stmt \\ &\quad | \text{other} \end{aligned}$$

if we have a shift-reduce parser in configuration

Stack	Input
... if <i>expr</i> then <i>stmt</i>	else ... \$

We cannot tell whether if *expr* then *stmt* is the handle, no matter what appears below it on the stack. Here there is a shift/reduce conflict. Depending on what follows the else on the input, it might be correct to reduce if *expr* then *stmt* to *stmt*, or it might be correct to shift else and then to look for another *stmt* to complete the alternative if *expr* then *stmt* else *stmt*. Thus, we cannot tell whether to shift or reduce in this case, so the grammar is not $LR(1)$. More generally, no ambiguous grammar, as this one certainly is, can be $LR(k)$ for any k .

We should mention, however that shift reduce parsing can be easily adapted to parse certain ambiguous grammars, such as the if-then-else grammar above. When we construct such a parser for a grammar containing the

two productions above, there will be a shift/reduce conflict: on else, either shift, or reduce by $stmt \rightarrow \text{if } expr \text{ then } stmt$. if we resolve the conflict in favor of shifting, the parser will behave naturally:

We could generate *CUNF* grammar as $G = (V, T, P, R, S, \$)$ for shift/reduce solution where P consists of

$$S \rightarrow IT \$ \mid ITE \$ \mid \text{other } \$$$

$$ITE \rightarrow IT E$$

$$IT \rightarrow I T$$

$$I \rightarrow \text{if } expr$$

$$E \rightarrow \text{else } IT \mid \text{else } ITE$$

$$T \rightarrow \text{then } IT \mid \text{then } ITE$$

$$\text{if} \rightarrow \text{if}$$

$$\text{then} \rightarrow \text{then}$$

$$\text{else} \rightarrow \text{else}$$

$$\text{other} \rightarrow \text{other}$$

and set R or context-sensitive rules consists of

$$\{ \text{else } IT > IT \$, \text{else } ITE > ITE \$, \text{then } IT > IT \$, \text{then } ITE > ITE \$ \\ \}$$

3.11.2 The reduce/reduce ambiguity

Suppose we have grammar such as:

$$S \rightarrow AZ \mid BY$$

$$Z \rightarrow CD$$

$$Y \rightarrow CD$$

$$A \rightarrow a$$

$$B \rightarrow b$$

$$C \rightarrow c$$

$$D \rightarrow d$$

A statement beginning with “a c d” would appear as the token stream “A C D” to the parser. After shifting the first three tokens onto the stack, a shift-reduce parser would be in configuration

Stack	Input
-------	-------

... A C D	\$...
-----------	--------

CUNF grammar for this reduce/reduce solution will be defined as grammar $G = (\{A, B, C, D, E, F\}, \{a, b, c, d\}, P, R, S, \$)$ where all productions are:

$$S \rightarrow F \$$$

$$F \rightarrow A E \mid B E$$

$$E \rightarrow C D$$

$$A \rightarrow a$$

$$B \rightarrow b$$

$$C \rightarrow c$$

$$D \rightarrow d$$

3.11.3 Problem of Special-Case Productions

Sometimes, the ambiguous grammars is useful if we introduce an additional production to specify a special case of a syntactic construct generated in a more general fashion by the rest of the grammar. When we add the extra production, we generate a parsing action conflict. We can often resolve the conflict satisfactorily by a disambiguating rule that says reduce by the special-case production. The semantic action associated with the additional production then allows the special case to be handled by a more specific mechanism.

An interesting use of special-case productions was made by (Kernighan and Cherry, 1975) in their equation-typesetting preprocessor *EQN*. In *EQN*, the syntax of a mathematical expression is described by a grammar that uses a subscript operator *sub* and a superscript operator *sup*, as shown in the grammar fragment below. Braces are used by the preprocessor to bracket compound expressions, and *c* is used as a token representing any string of text.

$$(1) E \rightarrow E \text{ sub } E \text{ sup } E$$

$$(2) E \rightarrow E \text{ sub } E$$

$$(3) E \rightarrow E \text{ sup } E$$

$$(4) E \rightarrow \{ E \}$$

$$(5) E \rightarrow c$$

Grammar above is ambiguous for several reasons. The grammar does not specify the associativity and precedence of the operators `sub` and `sup`. Even if we resolve the ambiguities arising from the associativity and precedence of the `sub` and `sup`, by making these two operators of equal precedence and right associative, the grammar will still be ambiguous. This is because production (1) isolates a special case of expressions generated by productions (2) and (3), mainly expressions of the form $E \text{ sub } E \text{ sup } E$. The reason for treating expressions of this form specially is that many typesetters would prefer to typeset an expression like $a \text{ sub } i \text{ sup } 2$ as a_i^2 rather than as a_i^2 . By merely adding a special case production, Kernighan and Cherry were able to get *EQN* to produce this special case output.

We could generate *CUNF* grammar G for special-case products solution as $(\{A, B, C, D, E, \text{char}, \text{sub}, \text{sup}, \text{rb}, \text{lb}\}, \{\{, \}, \text{sub}, \text{sup}, \text{alphanumeric}\}, P, R, S, \$)$ where P consists of

$$S \rightarrow \text{char } \$ \mid C \$$$

$$A \rightarrow \text{sub char}$$

$$B \rightarrow \text{sup char} \mid \text{sup } C$$

$$C \rightarrow \text{char } B$$

$$D \rightarrow \text{sub } C$$

$$E \rightarrow C \text{ rb} \mid \text{char rb}$$

$$\text{char} \rightarrow \text{alphanumeric} \mid \text{char } D \mid \text{lb } E \mid \text{char } A$$

$$\text{sub} \rightarrow \text{sub}$$

$$\text{sup} \rightarrow \text{sup}$$

$$\text{lb} \rightarrow \{$$

$$\text{rb} \rightarrow \}$$

and set R consists of

$$\{ \text{sub char} < \text{char } A, \text{sub char} < \text{char } B, \text{sub char} < \text{char } D, \\ \text{sup char} < \text{char } A, \text{sup char} < \text{char } B, \text{sup char} < \text{char } D, \\ \text{sub char} > \text{char rb}, \text{sup char} > \text{char rb}, \text{sup } C > C \text{ rb} \}$$

3.11.4 Problem of precedence-conflicts

This kind of ambiguity reflects the problem of goal-directed grammars that try to represent precedences and associativities within grammar. To define grammars that specify the associativities or precedences is not wrong as in the following grammar for arithmetic expressions with operators + and *.

$$E \rightarrow E + E \mid E * E \mid \text{id}$$

This grammar is ambiguous because it does not specify the precedence of the operators + and *. We can define an unambiguous grammar that generates the same language, but gives + a lower precedence than *, and makes both operators left-associative as

$$E \rightarrow E + T \mid T$$

$$T \rightarrow T * F \mid F$$

$$F \rightarrow (E) \mid \text{id}$$

To demonstrate the problem of precedence conflicts, let us define a grammar for arithmetic expression with operator +, * and ^ in ambiguous form as

$$E \rightarrow E + E \mid E * E \mid E \wedge E \mid \text{id}$$

We add 3 additional conditions into grammar by giving + a lower precedence than *, * a lower precedence than ^ and ^ a lower precedence than +. Then "1 + 2 * 3 ^ 4" or "1 + (2 * (3 ^ 4))" equals to 163, and "1 + 2 ^ 3" or "(1 + 2) ^ 3" equals to 27 following to all conditions above. In goal-directed grammars, writing unambiguous grammars for grammars with precedence conflicts is impossible. But, it could be defined in an unambiguous form using *CUNF* grammars.

We could generate *CUNF* grammar *G* for precedence conflicts solution as $\{A, B, C, \text{plus, mul, pow, id}\}, \{+, *, \wedge\}, P, R, S, \$$ where *P* consists of

$$S \rightarrow \text{id } \$$$

$$A \rightarrow \text{plus id}$$

$$B \rightarrow \text{mul id}$$

$C \rightarrow \text{pow id}$

$\text{plus} \rightarrow +$

$\text{mul} \rightarrow *$

$\text{pow} \rightarrow ^$

$\text{id} \rightarrow \text{digit} \mid \text{id A} \mid \text{id B} \mid \text{id C}$

and R consists of

$\{ \text{plus id} < \text{id A}, \text{plus id} < \text{id B}, \text{plus id} > \text{id C},$

$\text{mul id} > \text{id A}, \text{mul id} < \text{id B}, \text{mul id} < \text{id C},$

$\text{pow id} < \text{id A}, \text{pow id} > \text{id B}, \text{pow id} < \text{id C} \}$



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER 4

DATA-DIRECTED PARSER

4.1 Data-Directed Parser

A data-directed parser consists of two important parts; tree structure that represents derivation tree in computational process and recursive programming to operate on tree. Generally, a data-directed parser is a bottom-up parser. It generates a parse tree from input symbols or external nodes upward to start symbol or root. What distinguishes data-directed parsers from other bottom-up parsers is that a data-directed parsers use a data-directed grammars called *CUNF* grammars instead of other grammar formalisms. In fact, *CUNF* grammars and data-directed parser are two sides of the same coin. We can define *CUNF* grammars for any context-free language because of data-directed parser's power in recursion, and we can parse context-free language in $O(n)$ due to power of *CUNF* grammars for its unambiguous forms and rich information that helps in parsing input symbols.

Although, data-directed parsers and *LR* parsers can parse context-free language in $O(n)$, a data-directed parsers are more rigid because they can eliminate all conflicts; shift/shift conflicts, shift/reduce conflicts and reduce/reduce conflicts using only *CUNF* grammars. While users need to derive *LR* items and *LR* table respectively from grammar in order to use *LR* parser, ones use only *CUNF* grammars to parse the same input in time complexity equals to $O(n)$. Besides, its time complexity does not depend on a number of products or a number of states like *LR* parser but only a number of input symbols. Data-directed parser is a bottom-up parser that can scan input symbols both in forward and backward style. It will process input data following to grammar based on data-directed paradigm such as *CUNF* grammars. In fact, its power comes from the power of *CUNF* grammars which can define the priority of product rules to determine which one will be selected in order to build a parse tree. Due to the properties of tree, we can implement data-directed parser that can run in time complexity equals to $O(n)$.

4.2 Tree

Trees are data structures that are generally suitable for the representation of hierarchical data, and lie at the heart of many important algorithms and have been studied extensively as mathematical objects. In data-directed parser, tree plays an important role in representing derivation tree. Due to the properties of hierarchical data, we have an ancestor-descendant, superior-subordinate, whole-part, or similar relationship among the data elements including relations among symbols in syntactic rules. Then, in this section, we will consider the basic definitions and terminology associated with trees and the ways of representing trees as a model in parsing for data-directed grammar as *CUNF* grammars.

By definition, a tree is a finite nonempty set of elements. One of these elements is called the root, and the remaining elements are partitioned into trees which are called the subtrees as in figure 4.1. In addition, it is a collection of vertices and edges that satisfies certain requirements. A vertex is a simple object called a node that can have a name and can carry other associated information; an edge is a connection between two vertices. A path in a tree is a list of distinct vertices in which successive vertices are connected by edges in the tree, and one node in the tree is designated as the root. The defining property of a tree is that there is exactly one path between the root and each of the other nodes in the tree.

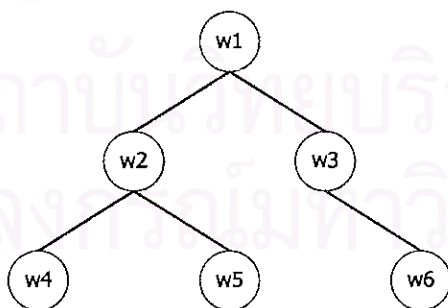


Figure 4.1 Tree Structure

4.3 Binary Tree

In particular, the simplest type of tree is the binary tree. A binary tree is an ordered tree consisting of two types of nodes: external nodes with no children and internal nodes with exactly two children of each internal node are ordered. Every internal node must have both a left and right child, though one or both of them might be an external node. Due to the definition of *CUNF* grammars in which right hand side of each production must have two nonterminal symbols, except those rules used in lexical analysis, binary tree is the most appropriate data structure for the model of parsing effectively using data-directed grammar as *CUNF* grammars.

The purpose of the binary tree is to structure the internal nodes; the external nodes serve only as place holders. We include them in the definition because the most commonly used representations for binary trees must account for each external node. A binary tree could be *empty*, consisting of no internal nodes and one external node. The full binary tree is one in which internal nodes completely fill every level, except possibly the last. A complete binary tree is a full binary tree where the internal nodes on the bottom level all appear to the left of the external nodes on that level.

The essential differences between a binary tree and a tree are a binary tree can be empty, whereas a tree cannot. Each element in a binary tree has exactly two subtrees, but each element in a tree can have any number of subtrees. The subtrees of each element in a binary tree are ordered. That is, we distinguish between the left and the right subtrees while the subtrees in a tree are unordered.

4.4 Recursion

Recursion is a fundamental concept in mathematics and computer science. The simple definition is that a recursive program is one that calls itself and a recursive function is one that is defined in terms of itself. However, a recursive program can't call itself always, or it would never stop. As well as a recursive function that can't be defined in terms of itself always, or the definition would be circular. Another essential ingredient

is that there must be a termination condition when the program can cease to call itself, and when the function is not defined in terms of itself.

A recursive function is a function that invokes itself. In direct recursion the code for function Z contains a statement that invokes function Z , whereas in indirect recursion, function Z invokes a function X , which invokes a function Y , and so on until function Z is again invoked. Recursive definitions of functions are quite common in mathematics in which we often define a function in terms of itself. For example, the most familiar recursive function is the factorial function, defined by the formula

$$n! = n * (n - 1)! \text{ for } n \geq 1 \text{ with } 0! = 1.$$

This definition corresponds directly to the following simple recursive program:

```
int factorial(int n)
{
    if (n == 0) return 1;
    return(n * factorial (n - 1));
}
```

This program illustrates the basic features of a recursive program. It calls itself with a smaller value of its argument, and it has a termination condition in which it directly computes its result. However, there is no masking the fact that this program is nothing more than a glorified for loop, so it is hardly a convincing example of the power of recursion. Also, it is important to remember that it is a program, not an equation. For example, neither the equation nor the program above works for negative N , but the negative effects of this oversight are perhaps more noticeable with the program than with the equation. The call `factorial (-1)` results in an infinite recursive loop. This is in fact a common bug that can appear in more subtle forms in more complicated recursive programs.

Recurrence relations often arise when we try to determine performance characteristics of recursive programs. Thus, the relationship between recursive programs and recursively defined functions is often more philosophical than practical. Actually, the problems pointed out above are associated not with the concept of recursion itself, but with the implementation.

4.5 Recursive Tree Traversal

Perhaps the simplest way to traverse the nodes of a tree is with a recursive implementation. For example, the following program visits the nodes of a binary tree in inorder style.

```

traverse(struct node *t)
{
    if (t != z)
    {
        traverse(t->l);
        visit(t);
        traverse(t->r);
    }
}

```

The implementation precisely mirrors the definition of inorder: "if the tree is nonempty, first traverse the left subtree, then visit the root, then traverse the right subtree." Obviously, preorder can be implemented by putting the call to visit before the two recursive calls, and postorder can be implemented by putting the call to visit after the two recursive calls. This recursive implementation of tree traversal is more natural than a stack-based implementation because trees are recursively defined structures and because preorder, inorder, and postorder are recursively defined processes.

4.6 Simple Recursive Tree Construction

In last section, we know how to traverse the nodes of the tree with recursive implementation. However, not only traversing the tree can be implemented using recursive paradigm but also building the tree that can be implemented due to the fact that tree is the recursively defined structures. This idea is a basic algorithm for implementing data-directed parser, so we will describe it in this section as shown below.

Simple Recursive Defined Tree Algorithm

```

# List consists of many nodes linked together (struct node)
Node* BuildParseTree (List *l)
{
    Node *p_node = l->lastNode() ;
    p_node = ParseTree(p_node);
}

```

```

return(p_node);
}
Node* ParseTree(Node *p_right)
{
    Node *p_left = p_right->leftNode();
    if (p_left == NULL)
        return(p_right);

    Node *p_new = new Node();
    Combine nodes pointed by p_left and p_right;
    p_new = ParseTree(p_new);
    return(p_new);
}

```

Suppose that we have link list of input symbols as figure 4.2

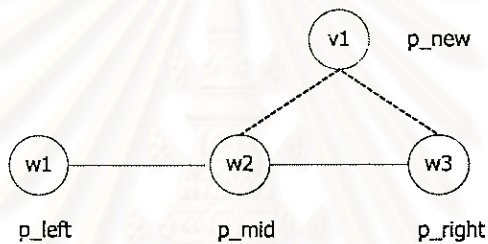


Figure 4.2 Linked List

After processed by algorithm above, we will get the tree structure as figure 4.3

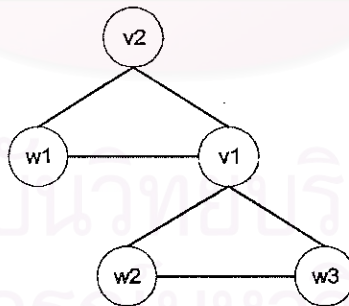


Figure 4.3 Parse Tree

Time complexity required for building tree structure of this case equals to $O(n)$, where n is a number of nodes in list. Suppose that number of nodes in list is 3. After processing 2 new nodes will be created. Finally, we will have tree structure that contains

5 nodes. Generally, we can say that giving link list of n nodes to build tree structure; $n-1$ new nodes will be created. Eventually, we will have a binary tree that contains $2n - 1$ nodes.

4.7 Conditional Recursive Tree Construction

Sometimes, building the tree structure may not be easy as an example above. In case that there are conditions among nodes, we have to consider these conditions before any nodes will be combined together. For instance, if we have grammar $G = (V, T, P, S)$ and set of productions as shown below:

$$A + B \rightarrow C$$

$$A + C \rightarrow D$$

$$E + D \rightarrow F$$

$$E + A \rightarrow G$$

$$A + F \rightarrow H$$

Besides, we know that relation between nonterminal symbol E and A is less powerful than the relation of nonterminal symbol E and A as $E + A < A + C$ when conflicts occur between these two rules.

Suppose that we need to parse "A E A C". Then, we will construct two hashes to store all information above. For the first hash called hash of rules, we will store all production rules in it. So, if we give two inputs as A and B to it, we will get C as a result. The other hash called hash of priority, we will store priority of product rules. When we give E , A and C to this hash, the result we will receive is either 'L' or 'R' which means left pair ($E + A$) or right pair ($A + C$) is selected respectively. From supporting of these two hashes, we will present algorithm as following one.

Conditional Recursive Defined Tree Algorithm

List consists of many nodes linked together (struct node)

Node* BuildParseTree (List *)

```
{
    Node *p_node = l->lastNode() ;
    p_node = ParseTree(p_node);
    return(p_node);
}
```

```

Node* ParseTree(Node *p_right)
{
    Node *p_mid = p_right->leftNode();
    Node *p_left = p_mid->leftNode();
    if ((p_right != NULL)&&(p_mid != NULL)&&(p_left != NULL))
    {
        String strCase = Priority_Hash(p_left->value(), p_mid->value(), p_right->value());
        if (strCase == 'L')
        {
            Node *p_new = new Node();
            Combine nodes pointed by p_left and p_mid as p_new ;
            p_new = ParseTree(p_right);
            return(p_new);
        }
        else if (strCase == 'R')
        {
            Node *p_new = new Node();
            Combine nodes pointed by p_mid and p_right as p_new;
            if (p_new->rightNode() != NULL)
            {
                p_new = ParseTree(p_new);
            }
            return(p_new);
        }
        else // data does not exist in hash
        {
            if can combine node pointed by p_left and p_mid
            {
                Node *p_new = new Node();
                Combine nodes pointed by p_left and p_mid as p_new;
                p_new = ParseTree(p_right);
                return(p_new);
            }
            else if can combine node pointed by p_mid and p_right
            {
                Node *p_new = new Node();
                Combine nodes pointed by p_mid and p_right as p_new;
                if (p_new->rightNode() != NULL)
                {
                    p_new = ParseTree(p_new->rightNode());
                }
                return(p_new);
            }
            else
            {
                Node *p_new = ParseTree(p_left->leftNode());
                return(p_new);
            }
        }
    }
}
else if ((p_right != NULL)&&(p_mid != NULL))

```

```

{      if can combine node pointed by p_mid and p_right
      {      Node *p_new = new Node();
            Combine nodes pointed by p_right and p_mid as p_new;
            return(p_new);
      }
      else
      {      return(p_mid);
      }
}
else
return(p_right);
}

```

4.8 Algorithm of Data-Directed Parser

Generally, data-directed parser's algorithm is not different from the prior one in last section except function *CanCombine()*. This function plays an important role in algorithm for the priorities among production rules in grammar. It determines whether parser could reduce some symbols at specific position in strings or not. It also considers which rule will be use in reduction steps when conflicts occur in parsing process. Algorithm of data-directed parser in backward style implemented using recursive programming was shown in section below.

Data-directed Parse Algorithm

```

function DataDirectedParser
{
    set pRight = last node in list;
    pRight = DdpParse(pRight);
    return(pRight);
}

function DdpParse(Node* pRight)
{
    Node* pMid = pRight->Left();
    Node* pLeft = pMid->Left();

    if ((pLeft != NULL)&& (pMid != NULL)&& (pRight != NULL))
    {
        if (CanCombine(pRight, intDirection, strValue))
        {
            if ($intDirection == 1) # reduce left pair of symbols
            {
                Node *pNew = new Node(strValue);
            }
        }
    }
}

```



```

        Replace pLeft and pMid with pNew;
        If (pRight->Right() != NULL)
        {      return( DdpParse(pRight->Right()));      }
        else
        {      return( DdpParse(pRight));      }
    }
    elseif ($intDirection == -1) # reduce right pair of symbols
    {      Node *pNew = new Node(strValue);
        Replace pMid and pRight with pNew;
        If (pNew->Right() != NULL)
        {      return(DdpParse(pNew->Right()));      }
        else
        {      return(DdpParse(pNew));      }
    }
    else
    {      return(DdpParse(pRight->Left()));      }
}
else if ((pMid != NULL)&& (pRight != NULL))
{      if (CanCombine(pRight, intDirection, strValue))
    {      Node *pNew = new Node(strValue);
        Replace pMid and pRight with pNew;
        return(pNew);
    }
    else
    {      return( DdpParse(pMid));      }
}
else if (pRight != NULL)
{      return(pRight);      }
else
{      return('NULL');      }
}

```

4.9 Proving Algorithm of Data-Directed Parser

Consider languages written in the books or newspaper. Have we ever seen the corresponding grammars attached with them? If there are nothing but the sentential input symbols, why most of us think that analyzing language should start with start symbol downward to input symbols? At least, we need to see those symbols first to

choose the appropriate grammar because there are many grammars for various languages.

Data-directed parser is a bottom-up parse that scans input symbols from right to left. It attempts to construct a parse tree for an input string beginning at the leaves and working up towards the root. The purpose of this process is to reduce a string to the start symbol of a grammar. At each reduction step a particular substring matching the right side of a production is replaced by the symbol on the left of that production. Hence, in each reduction step, one symbol will be removed from string. That means we need n reduction steps to reach the start symbol or the root of parse tree following to *CUNF* grammars, where n is a number of input symbols in string. Suppose that we have the input symbols as in figure 4.4, we will proof this algorithm by dividing problems into 3 cases below.

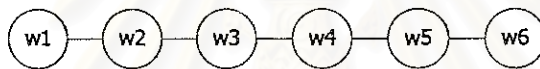


Figure 4.4 Input symbols represented by linked list

Case I

This case occurs when an input symbol is sent to parser as in figure 4.5. There is nothing for us to do with this case because all syntactic rules that used in reduction steps always need two symbols on the right of productions according to *CUNF* grammars. So, the pointer that points to current symbol will be returned.



Figure 4.5 Case of one input symbol

Case II

When there are 2 input symbols in string as in figure 4.6, we must consider if these two symbols could be reduced by a syntactic rule in grammar. If we can find the rule that its right side of production matches these two

symbols, then they will be replaced with the symbol on the left side of that product. It is certain that we have a number of input symbols equal to 2 at the beginning. Then, after this reduction step, the rest of input symbols in list will be 1 element as in figure 4.7, and will be sent to case I. However, if we cannot find one as in figure 4.8, the pointer pointing to current node will be shifted to the left, and it will be sent to case I as well.

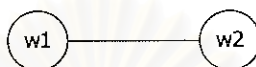


Figure 4.6 Case of two input symbols

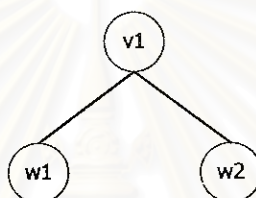


Figure 4.7 Case of reducible two symbols

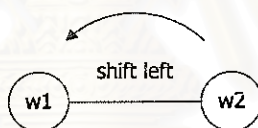


Figure 4.8 Case of irreducible two symbols

Case III

When there are 3 input symbols in consideration as in figure 4.9, two choices that could occur are whether the left pair of symbols or the right pair of symbols will be reduced following to grammar. In each reduction for this case, two symbols will be replaced with one symbol according to syntactic rule. So, if we have k symbols for input, there will be only $k - 1$ input symbols after each reduction step. Otherwise, current pointer will be shift to the left and sent to another case if we cannot find the production that its right side matched left pair or right pair of these 3 symbols as in figure 4.10.

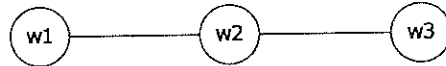


Figure 4.9 Case of three input symbols

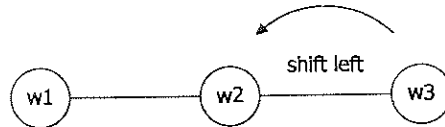


Figure 4.10 Case of three irreducible input symbols

In fact, the behaviors of this parser are quite different after reduction step depend on the position that these 3 symbols appear in string. Suppose that 3 pointers pointing to 3 current symbols are $pLeft$, $pMid$ and $pRight$ from left to right respectively, where $pLeft$ points to w_1 , $pMid$ point to w_2 , and $pRight$ to w_3 as in figure 4.9. If either left pair or right pair of symbols are reduced, we have to consider whether the right node of $pRight$ is empty or not. If there is a node behind $pRight$, we must shift right before going to another case in next recursion as shown in figure 4.11 and figure 4.12.

Sometimes, it is possible for $pRight$ as the right most element in list which means there is no node next to it on the right. After reduction step, the current right most element in list will be sent to the next recursion. This right most element could be either w_3 pointed by $pRight$ as in figure 4.13 or v_1 pointed by $pNew$ as in figure 4.14, where $pNew$ will be sent to the next recursion as $pRight$.

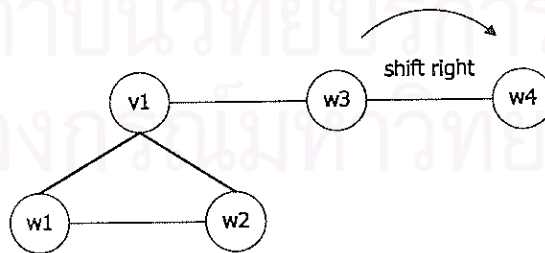


Figure 4.11 Case of shifting right after left reduction

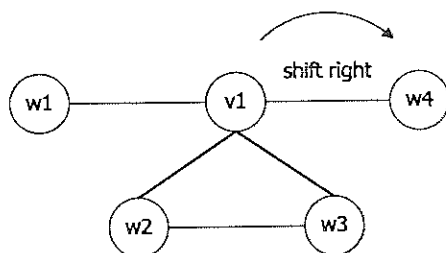


Figure 4.12 Case of shifting right after right reduction

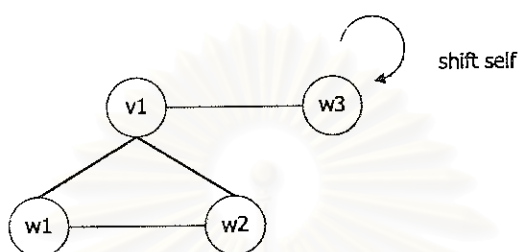


Figure 4.13 Case of shifting self after left reduction

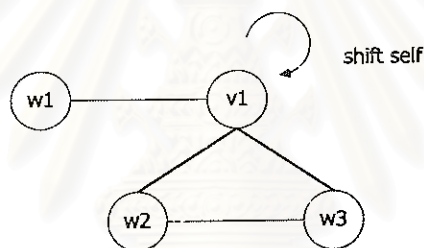


Figure 4.14 Case of shifting self after right reduction

Three cases described above are all cases in function *DdpParse* that we must encounter while parsing. Besides, there is something that is important in function *CanCombine* that we should understand. Suppose that we have three input symbols sent into *CanCombine* as in figure 4.9. We need to examine is whether left pair or right of symbols could be reduced. Begin with searching sequence of $w_1w_2w_3$ in context-sensitive rules. If we can find one in hash that require $O(1)$, that means both left pair and right pair of symbols could be reduce, but only one will be selected according to value that we retrieve from hash of context-sensitive rules.

However, it is possible that we will find item $w_1w_2w_3$ in context-sensitive rules but cannot reduce any one neither left pair nor right pair because of priority flow in context. Finally, if there is no item $w_1w_2w_3$ in hash of context-sensitive rules, hash of production will be searched for the value w_1w_2 or w_2w_3 respectively for either left pair or right pair of symbols may be chosen in reduction step.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER 5

EXPERIMENT

5.1 Languages used in Experiment

In this experiment, we demonstrated how to apply *CUNF* grammar with *HTML* (Hyper Text Markup Language) version 3.2 as described in (Ian, 1997) including a simple recovery solution. By the experiment, we will prove what we claimed for *CUNF* grammar and data-directed parser. Generally, a markup language is a way of describing, using instructions embedded within a document. Those instructions or markup elements used in *HTML* called tags which are the sections of text enclosed by a less-than and greater-than sign (<...>), and are the markup instructions that explain what each part of the document means, where <...> indicates start tag and </...> indicates end tag. For example, the tag <TITLE> indicates the start of a title, while the </TITLE> tag marks the end of a title. Thus, the text string

```
<TITLE> Hyper Text Markup Language </TITLE>
```

marks the string "Hyper Text Markup Language" as a title.

5.1.1 Element Attributes

Due to the definition of *HTML*, some elements can take attributes that define properties or special information about the element. Generally, attributes are much like variables. They always appear in the start tag of an element, and are usually assigned values that define these special properties. For example, the element

```
<H1 ALIGN="center"> Element Attributes </H1>
```

takes an ALIGN attribute, which states that the heading should be centered on the display. Another example, used to include an image within an HTML document, is found in the IMG element. An IMG element appears via the tag:

```
<IMG SRC = "picture.gif">
```

The SRC attribute specifies the name of the image file to be included in the document. The attribute name, like the element name, is case-insensitive. Thus the above line could equally well be written as either of:

```
<img SRC="picture.gif">
```

```
<img SrC="picture.gif">
```

However, the value assigned to an SRC attribute is case-sensitive; case-sensitivity can be preserved by enclosing the string in quotation marks. As we may have noticed, the IMG element is empty like the BR element, since it merely inserts an image and does not affect a block of text.

5.1.2 Structured Languages

HTML is a structured language which means that there are rules for where elements can and cannot go. These rules are presented to enforce an overall logical structures upon the document. For example, a heading element like `<H1> ... </H1>` can contain text, text marked for emphasis, line breaks, inline images, and hypertext anchors, but it cannot contain any other HTML element. As a result, the markup

```
<H1><H2> ... text ... </H2></H1>
```

is invalid. Obviously, it does not make sense for a heading to “contain” another heading, and the *HTML* rules reflect this reality. In addition, elements can never overlap; this means that tag placement like

```
<H1><H2> Structure Language </H1></H2>
```

is illegal. There are many such structural rules.

5.2 CUNF Grammars for HTML

We can divide the markup tags of *HTML* into 5 categories as ‘S’ for a start tag, ‘E’ for an end tag, ‘T’ for a block of text, ‘C’ for a comment tag and ‘D’ for a declaration tag respectively. First, ‘S’ or a group of start tags which are tags like `<html>`, `<head>` or `<body>` that each one consists of section of text enclosed by a less-than and greater-than sign (`< ... >`). Second, ‘E’ or a group of end tags such as `</html>`, `</head>` or

`<body>` which are section of text enclosed in (`</ ... >`). Third, 'T' or the section of text that is not written within a less-than and a greater-than sign like "hyper text markup language". Fourth, 'C' or a group of comment tags like `<!-- comment -->` which are any text written within (`<!-- ... -->`). And the last one, 'D' or a group of declaration tags which are section of text written in (`<! ... >`) such as `<! declaration >`. Besides, we need to classify them into 3 types in order to generate an appropriate syntactic rules defined in *CUNF* grammars as type I rules, type II rules and type III rules respectively.

5.2.1 Type I Rules

Type I rules are rules for tags that have both the start tag and end tag like `<html> ... </html>` or `<head> ... </head>`. We define symbol 'Sx' for any start tag `<x>` where x is a tag name, and we define symbol 'Ex' for any end tag `</x>` where x is a tag name. Then symbols that represent `<html>` and `</html>` will be 'Shtml' and 'Ehtml' respectively. In reduction step, both 'Sx' and 'Ex' will be reduced to 'x'. That means both 'Shtml' and 'Ehtml' will be reduced to 'html' which characterize as ordinary text in syntactic rules. Hence, tags as `<head>` and `</head>` will be represented as 'Shead' and 'Ehead' which can be replaced with 'head' as well. Suppose that there are "`<html> <head> example </head> </html>`" in html file, where section of text 'example' will be represented with 't'. We will define grammar rules according to *CUNF* grammars for these tags as

$$\begin{aligned} \text{Shtml} + \text{Ehtml} &\rightarrow \text{html} \\ \text{head} + \text{Ehtml} &\rightarrow \text{Ehtml} \\ \text{Shead} + \text{Ehead} &\rightarrow \text{head} \\ \text{t} + \text{Ehead} &\rightarrow \text{Ehead} \end{aligned}$$

5.2.2 Type II Rules

Type II rules are rules for tags that have only start tag such as `<base>` or `<isindex>`. Sometimes, it means that these tags such as `<base>`, `<isindex>`, `<link>`, or `<meta>` are complete at first. So, we will define 'Sx' for `<x>` where x is a tag name that has only a start tag. However, it is possible to write text after

these tags like '<link> link'. Then we shall define rules based on CUNF grammar for these tags as

$$\text{Slink} + t \rightarrow \text{Slink}$$

5.2.3 Type III Rules

Type III rules are rules for tags that have start tag but end tag is optional such as ... or We define symbol 'Sx' for start tag <x> and symbol 'Ex' for end tag because end tag is optional. This fact also means only a start tag is enough, so we define symbol 'Sx' for these kinds of tag such as <dt>, <dd> or . Suppose that we have " unodered list list item " and " unodered list list item" in html files, we shall define grammar rules for them as

$$\text{Sul} + \text{Eul} \rightarrow \text{ul}$$

$$t + \text{Eul} \rightarrow \text{Eul}$$

$$\text{li} + \text{Eul} \rightarrow \text{Eul}$$

$$\text{Sli} + \text{Eli} \rightarrow \text{li}$$

$$\text{Sli} + t \rightarrow \text{Sli}$$

$$t + \text{Eli} \rightarrow \text{Eli}$$

5.3 Output Format

In the experiment, input sentences will be analyzed by lexical module and syntactic module respectively to produce parse tree for each file of HTML. To represent the parse tree which consists of nodes and arcs or links between nodes, we will use flat tree form instead. For example, (S (A B)) as figure 5.1 a) means the parse tree that has root S and two children; A and B. Besides, the tree could have subtree like (S (A B (C D))) as figure 5.1 b) which means the child B has two children; C and D.

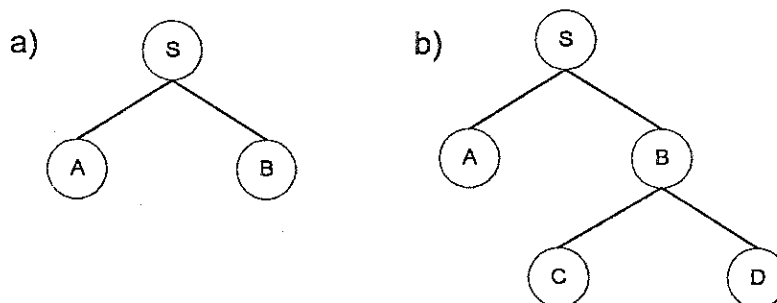


Figure 5.1 Parse tree

Results from the experiment in this stage, a parser will parse input sentences and displaying its structure in a flat tree form. The parse tree will be considered as a successful one if its root labeled S, and will be an incomplete parse tree for its root with the other symbols.

5.4 Error-Insensitive Solution for HTML

It is quite useful for us to understand the concept of solution for incomplete tags in HTML file. Due to the nature of HTML which is structured language, we can divide tags into 5 categories as 'S' for a start tag, 'E' for an end tag, 'T' for a block of text, 'C' for a comment, and 'D' for a declaration tag. For tags that have both the start tag and end tag like `<body> ... </body>`, we can reduce them to body. Otherwise, we cannot reduce them according to *CUNF* grammars such as `<body> ... </head>`. In ordinary parsing, we can reject this input data as an incomplete parse tree, but in error-insensitive parsing we need to repair them in parsing process to form the complete parse tree. Hence, we will define three virtual categories of tags for an error-insensitive parsing as 'S' for a start tag, 'E' for an end tag, and 'X' for other tags. Besides, we need to construct the transition table as shown in table 5.1 for any cases of tags that could occur in parsing including the way to solve them. The transition table shown in table 5.1 is an example of error-recovery implementation according to *HTML* rules. It is a solution with 9 states and some constraints. In error-recovery mode, tag BODY and HTML will be added to input file automatically if they are missing.

Table 5.1 Error-Recovery Table for HTML

State	Label	Next State
1	SS	1, 2, 3
2	SE	1, 2, 3, 5, 8, 9
3	SX	1, 2, 3
4	ES	4, 5, 6
5	EE	2, 4, 5, 6, 8
6	EX	2, 5, 8
7	XS	7, 8, 9
8	XE	3, 6, 9
9	XX	3, 6, 9

5.5 Common Parsing using Data-Directed Parser

Data-Directed Parser can be applied to various works that require time complexity equals to $O(n)$ in parsing while languages used in parsing must be defined correctly in *CUNF* grammars. Sometimes, we can think that grammar is a programming language that controls all processes of parser. So, it is not important whether our problems will relate to grammar theory and languages or not. The only important one is that if we can transform those of them into *CUNF*, nothing can prohibit us from our goal. To apply it with parsing, the example of HTML file was shown below as,

Html File

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<!-- saved from url=(0031)http://www.w3.org/TR/REC-html32 -->
<HTML><HEAD><TITLE>HTML 3.2 Reference Specification</TITLE>
<META content="text/html; charset=iso-8859-1" http-equiv=Content-Type>
<META content="MSHTML 5.00.2614.3500" name=GENERATOR></HEAD>
<BODY aLink=#ff00ff
background="HTML 3_2 Reference Specification_files/recbg.jpg" bgColor=#fff6f0
link=#c00000 text=#000000 vLink=#800000><!--basefont size=3-->
<H2 align=right>132</H2>
<H1 align=center>HTML 3.2 Reference Specification</H1>
<H3 align=center>Abstract</H3>
```

```

<P>The HyperText Markup Language (HTML) is a simple markup language used to
create hypertext documents that are portable from one platform to another. HTML
documents are SGML documents with generic semantics that are appropriate for
representing information from a wide range of applications. This specification
defines HTML version 3.2. HTML 3.2 aims to capture recommended practice as of
early '96 and as such to be used as a replacement for HTML 2.0 (<A
href="http://www.w3.org/TR/REC-html32#refs">RFC 1866</A>).
<HR>
<H2>Contents</H2>
<UL>
<LI><A href="http://www.w3.org/TR/REC-html32#intrc">introduction to HTML 3.2</A>
<LI><A href="http://www.w3.org/TR/REC-html32#sgm">HTML as an SGML application</A>
<LI><A href="http://www.w3.org/TR/REC-html32#html">The Structure of HTML documents</A>
<LI><A href="http://www.w3.org/TR/REC-html32#head">The HEAD element and its
children</A>
<LI><A href="http://www.w3.org/TR/REC-html32#body">The BODY element and its
children</A>
<LI><A href="http://www.w3.org/TR/REC-html32#acks">Acknowledgements</A>
<LI><A href="http://www.w3.org/TR/REC-html32#refs">Further Reading ...</A> </LI>
</UL>
<HR>
<H2>&nbsp;</H2>
</BODY></HTML>

```

We will display the result of parse tree in the form of flat tree with the root that labeled *html* symbol, which means the process is successful. The result of the experiment will be shown as

Parse Tree

```

(html (d html (c html (Shtml Ehtml (head (Shead Ehead (title (Stitle Etitle (t Etitle )))Ehead (Smeta Ehead (Smeta Ehead
))))Ehtml (body (Sbody Ebody (c Ebody (h2 (Sh2 Eh2 (t Eh2 )))Ebody (h1 (Sh1 Eh1 (t Eh1 )))Ebody (h3 (Sh3 Eh3 (t Eh3
)))Ebody (Sp Ebody (t Ebody (t Ebody (a (Sa Ea (t Ea )))Ebody (t Ebody (Shr Ebody (h2 (Sh2 Eh2 (t Eh2 )))Ebody (ul (Sul
Eul (Sli (Sli a (Sa Ea (t Ea )))Eul (Sli (Sli a (Sa Ea (t Ea )))Eul (Sli (Sli a (Sa Ea (t Ea )))Eul (Sli (Sli a (Sa Ea (t Ea )))Eul (Sli
(Sli a (Sa Ea (t Ea )))Eul (Sli (Sli a (Sa Ea (t Ea )))Eul (li (Sli Eli (a (Sa Ea (t Ea )))Eli )))Eul ))))))))Ebody (Shr Ebody (h2
(Sh2 Eh2 (t Eh2 )))Ebody ))))))))Ehtml ))))
<Structure High Level> ( html )

```

5.6 Error-Recovery Parsing using Data-Directed Parser

We can also develop the original concept of parser to parse any HTML files that are incomplete as well by depending on clues or existing tags in context. By applying an error-recovery table described in previous section, we will have an error insensitive parser that can roughly removes errors that occur in parsing process. It can also assign the appropriate tags in order to build a parse tree with the symbol S at the root. The experiment with the incomplete HTML file with the error-free parser will be demonstrated below as:

Error-Recovery Result

```

S E -> S X :: recommend to add tag: <h2> for tag: </h2> at line 9
S E -> ? E :: recommend to remove tag: <meta> before tag: </html> at Line 5
S E -> ? E :: recommend to remove tag: <meta> before tag: </html> at Line 4
E E -> ? E :: recommend to remove </title> before tag: </html> at line 3
S E -> ? E :: recommend to remove tag: <head> before tag: </html> at Line 3

```

Html File

```

1:      <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
2:      <!-- saved from url=(0031)http://www.w3.org/TR/REC-html32 -->
3:      <HTML><HEAD>HTML 3.2 Reference Specification</TITLE>
4:      <META content="text/html; charset=iso-8859-1" http-equiv="Content-Type">
5:      <META content="MSHTML 5.00.2614.3500" name="GENERATOR">
6:      <BODY aLink=#ff00ff
7:      background="HTML_3_2 Reference Specification_files/recbg.jpg" bgColor=#fff6f0
8:      link=#c00000 text=#000000 vLink=#800000><!--basefont size=3-->
9:      l32</H2>
10:     <H1 align=center>HTML 3.2 Reference Specification</H1>
11:     <H3 align=center>Abstract</H3>
12:     <P>The HyperText Markup Language (HTML) is a simple markup language used to
13:     create hypertext documents that are portable from one platform to another. HTML
14:     documents are SGML documents with generic semantics that are appropriate for
15:     representing information from a wide range of applications. This specification
16:     defines HTML version 3.2. HTML 3.2 aims to capture recommended practice as of
17:     early '96 and as such to be used as a replacement for HTML 2.0 (<A
18:     href="http://www.w3.org/TR/REC-html32#refs">RFC 1866</A>).
19:     <HR>

```

```
20: <H2>Contents</H2>
21: <UL>
22: <LI><A href="http://www.w3.org/TR/REC-html32#intro">Introduction to HTML 3.2</A>
23: <LI><A href="http://www.w3.org/TR/REC-html32#sgml">HTML as an SGML application</A>
24: <LI><A href="http://www.w3.org/TR/REC-html32#html">The Structure of HTML documents</A>
25: <LI><A href="http://www.w3.org/TR/REC-html32#head">The HEAD element and its
26: children</A>
27: <LI><A href="http://www.w3.org/TR/REC-html32#body">The BODY element and its
28: children</A>
29: <LI><A href="http://www.w3.org/TR/REC-html32#acks">Acknowledgements</A>
30: <LI><A href="http://www.w3.org/TR/REC-html32#refs">Further Reading ...</A> </LI>
31: </UL>
32: <HR>
33: </BODY></HTML>
```



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER 6

CONCLUSION AND PERSPECTIVE

6.1 Conclusion

Since the concept of grammars and the hierarchy of grammars was proposed by Chomsky in (Chomsky, 1956) and (Chomsky, 1959), many parsing techniques have been developed in 2 distinct concepts; top-down and bottom-up paradigm. Both of these paradigms reflect 2 important insights in the philosophical tradition; the rationalist tradition or goal-directed which focuses on the prior knowledge, and the empirical tradition or data-directed which focuses on the data. Even though these two parser paradigms are different, both of them can share the same grammar and reflect different problems. Top-down parser searches for a parse tree by trying to build from the root node S down to the leaves, but it could fail to match all the symbols in the input. While bottom-up parser starts from the symbols of the input and tries to build tree until the root S is found, but it could fail as well.

To reduce an apparently exponential-time problem to polynomial-time one by eliminating the repetitive solution of sub-problems inherent in backtracking approaches, two dynamic programming approach was proposed; Earley's algorithm and CYK algorithm. Earley's algorithm is a top-down parser that was proposed by (Earley, 1970) while CYK algorithm is a bottom-up parser that was proposed by Kasami (Kasami, 1965) and Younger (Younger, 1967). The dynamic programming approach leads to a worst-case behavior of $O(n^3)$, where n is the number of words in the input while the original top-down and bottom-up requires time complexity up to C^k . However, these four parser techniques are parser that can work using only grammars unlike LR parser.

LR parser; the most famous technique used in compiler today which proposed by Aho (Aho, 1986), needs additional information called LR parsing table that has two parts; *action* and *goto*, in order to parse deterministic context-free grammar in time complexity equals to $O(n)$. This technique is formally called as $LR(k)$ parsing where L is for left-to-right scanning of input, R for construction a rightmost derivation in reverse,

and k for the number of input symbols of lookahead that are used in making parsing decisions. From the experiment, we found that time complexity of *LR* parsing does not depend on a number of input symbols but depends on states generated from grammars especially when those grammars have precedences and associativities. *LR* parser is also said that it could use an ambiguous grammar as a seed to generate *LR* parsing table; two advantages for using ambiguous grammar is shorter, more natural specification and for special case optimization. Although this may be interesting, we often encounter with resolving conflicts occurring in process of generating *LR* parsing table such shift/reduce conflicts or reduce/reduce conflicts.

In this research, we try to propose two important parts for syntactic analysis. Firstly, we propose a new data-directed grammar formalism called *CUNF* grammar, which is unambiguous and could be applied to various problems. *CUNF* is defined as $G = (V, T, P, R, S, \$)$ where V is a set of non-terminal symbols, T is a set of terminal symbols, P is a set of productions, R is a set of context-sensitive rules, S is a start symbol, and $\$$ is an end symbol. It reflects two important parts in grammar formalism; productions and priorities in using those productions. Languages with precedence conflicts which are described in chapter 3 demonstrating the existence of these priorities that plays important roles in parsing. As a result, many ambiguous grammars could be described easily in unambiguous form using *CUNF* grammars. Problems such as shift/shift conflicts, shift/reduce conflicts and reduce/reduce conflict will be resolved by this grammar formalism and a number of productions defined in *CUNF* grammars are often compact following to Chomsky Normal Form.

We present two classes of grammars; context-free grammars and regular grammars could be written in *CUNF* grammar. For regular grammar, it is not complex and there is nothing special for writing them in *CUNF* grammars. However, for context-free grammars which consist of two subclasses; non-deterministic and deterministic context-free grammars, there are some advantages for writing them in *CUNF* grammar instead of writing them in other grammar formalisms. Those advantages are that we can write both ambiguous grammars and unambiguous grammars in unambiguous form. Besides, for some language, It is possible to produce both non-deterministic and

deterministic results from *CUNF* grammars. For special purpose, when some deterministic results are preferred, we could defined R_1, R_2, \dots, R_{n-k} for those results in grammars G as the form $G_1 = (V, T, P, R_1, S, \$)$, $G_2 = (V, T, P, R_2, S, \$)$ and so on. In case that all non-deterministic results are necessary, we can use a bottom-up parser like *CYK* parser to parse all possible parse trees using grammar G by omitting context-sensitive rules or set R . By removing set R or context-sensitive rules, *CUNF* grammars will resemble the definition of grammars defined in Chomsky Normal Form, and will be compatible with *CYK*'s parser.

Secondly, we propose a data-directed parser which is a bottom-up parser as well as *CYK*'s parser. Data-directed parser works with *CUNF* grammars and uses only information defined in *CUNF* grammars. Its time complexity equals to $O(n)$ and does not depend on states like *LR* parser but depends on input symbols only. Precedences and associativities do not affect data-directed parser as in *LR* parser, so its time complexity will always equals to $O(n)$. Due to the fact that there are many grammars as well as there are many start symbol S 's from top-down view, it is quite difficult to answer why we choose one start symbol instead of others despite we do not know what input symbols are at first. As in bottom-up view, there could be many nonterminal symbols from various grammars which an input symbol could be reduced to. However, we can use some clues such contextual information that may be useful in decisions to produce the better results.

Generally, a data-directed parser may be more attractive than *LR* parser for its k lookahead symbols both for the left and right direction where k is an infinite value following to syntactic rules defined in grammars. Due to power of recursive programming which plays the important role in the data-directed parser, we can link priorities of all productions together in parsing while it can maintain time complexity equals to $O(n)$. With these two concepts; one for a new grammar formalism and the other one for a new parser, it could be useful to write grammars both ambiguous and unambiguous forms in unambiguous forms using only *CUNF* grammars only. Each deterministic result generating from *CUNF* grammars could be received from the data-

directed parser. Besides, all non-deterministic results could be also produced by *CYK's* parser.

6.2 Perspective

Although those information written in each chapter concentrates on context-free grammars and regular grammars, *CUNF* grammars are powerful enough to deal with natural languages in deterministic style depends on all information we have. However, natural language is rather complex and ambiguous, so using only syntactic rules is not a good idea. Many techniques especially probabilistic models have been introduced to natural languages processing tasks based on increasing availability of text corpora. In syntactic parsing, probabilistic techniques are utilized to rank the potentially high numbers of parses generated for natural language applications, and several attempts have been made to prune meaningless parse trees and aid in the selection of the most likely parse from multiple parse candidates.

Fujisaki et al. (Fujisaki et al., 1989) introduced the notion of a probabilistic context-free grammar (*PCFG*), with probabilities trained in the Forward/Backward manner. Wright and Wrigley (Wright and Wrigley, 1991) formalized a method of mapping *PCFG* onto *LR* parsing tables by way of distributing the probabilities originally associated with a given *CFG* to each corresponding *LR* parsing action. Briscoe and Carroll (Briscoe and Carroll, 1993) proposed the simpler way of incorporating trained probabilities into each parsing action of the *LR* table. Probabilities are computed directly from the frequency of application of each action while it is parsing the training corpus. Their method seems to be able to exploit the advantages offered by the context-sensitivity of *GLR* parsing.

In addition, it is possible for applying probabilistic models to a data-directed parser as well as *LR* parser. By using data-directed parser, we will have unique structures for any sequences of input strings that share the same patterns. In additional experiment with natural languages, we notice some sentences which are really

ambiguous even for human. For example, the sentence "I like the red ball and flower." which means (red (ball and flower)). But, it may also means (red ball and (flower)). In this case, selecting an appropriate grammar for a specific target should be a better method. In the future, the problem that should be concerned is a semantic problem. Sometimes, collocation or knowledge-based approach should have been applied to produce a better result.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

REFERENCES

- Aho, A. V., and Ullman, J. D. The Theory of Parsing, Translation, and Compiling. Vol.1. (n.p.) : Prentice-Hall, 1972.
- Aho, A. V., Sethi, R., and Ullman, J. D. Compilers: Principles, Techniques, and Tools. (n.p.) : Addison-Wesley, 1986.
- Allen, I. H. Compiler Design in C. New Jersey : Prentice Hall, 1990.
- Allen, J. Natural Language Understanding. (n.p.) : Benjamin/Cummings, 1995.
- Backus, J. The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference. Information Processing (1960) : 125-132.
- Briscoe, T., and Carroll, J. Generalized Probabilistic LR Parsing of Natural Language (Corpora) with Unification-Based Grammars. Computational Linguistic 19,1, (1993) : 25-59.
- Chomsky, N. Three Models for the Description of Language. IRI Transactions on Information Theory 2,3, (1956) : 113-124.
- Chomsky, N. On certain formal properties of grammars. Information and Control 2, (1959) : 137-167.
- Dick, G. et al. Modern Compiler Design. New York : John Wiley and Sons, 2000.
- Earley, J. An Efficient Context-Free Parsing Algorithm. Communications of the ACM 6,8, (1970) : 451-455.
- Fujisaki, T. et al. A Probabilistic Parsing Method for Sentence Disambiguation. Proceedings of 1st International Workshop on Parsing Techniques (1989) : 85-94.
- Gilbert, K. K. Computer Processing of Natural Language. New Jersey. : Prentice-Hall, 1999.
- Graham, S. L., Harrison, M. A., and Ruzzo, W. L. On-line context-free Language Recognition in less than cubic time. Proc. Eighth Annual ACM Symposium on

Switching and Automata Theory (1976) : 175-180.

Grune, D., and Jacobs, C. J. H. Parsing Techniques (A practical Guide). (n.p.) : Ellis Horwood, 1990.

Ian, S. G. HTML Sourcebook. Third Edition. (n.p.) : John wisley & sons, 1997.

John, E. H., and Jeffrey, D. U. Introduction to Automata Theory, Language, and Computation. (n.p.) : Addison-Wesley, 1979.

Kasami, T. An Efficient Recognition and Syntax Algorithm for Context-free Languages. Scientific Report AFCRL-65-785 (1965).

Kasami, T., and Tori, K. . A Syntax Analysis Procedure for Unambiguous Context-free Grammars. J. ACM 16,3, (1969) : 423-431.

Kernighan, B. W., and Cherry, L. L. . A System for Typesetting Mathematics. Comm. ACM 18,3, (1975) : 151-157.

Lee, K-H. Table-driven Parsing with Non-LR and Underspecified Grammars. NLPRS' 97 (Incorporating SNLP' 97) Proceeding of the Natural Language processing Pacific Rim Symposium (1997) : 175-180.

Shiina, H., and Masuyama, S. Proposal of the UGLR Parser for Phrase Structure Grammars. NLPRS' 97 (Incorporating SNLP' 97) Proceeding of the Natural Language processing Pacific Rim Symposium (1997) : 529-532.

Tomita, M. Generalized LR parsing. Kluwer Academic Publisher, 1991.

Varakulsiripunth, R., and Junwun, S. The Analysis on Sentence Structure by M-ATN. การประชุมทางวิชาการ วิศวกรรมไฟฟ้า ๑ สถาบัน ครั้งที่ 11 เล่ม 1. สถาบันเทคโนโลยีราชมงคล (1988) : 1-18-1 – 1-18-13.

Varakulsiripunth, R., Junwun S., and Maneenate, N. Thai Syntactical Analysis by M-ATN. การประชุมวิชาการทาง วิศวกรรมไฟฟ้า ครั้งที่ 12. มหาวิทยาลัยเกษตรศาสตร์ (1989).

Variant, L. G. Regularity and Related Problem for deterministic pushdown automata. J. ACM 22,1, (1975) : 1-10.

Wright, J. H., and Wrigley, E. N. GLR Parsing with Probability. Kluwer Academic

Publishers, 1991.

Yngve, V. H. Syntax and the problem of multiple meaning. Machine Translation of Languages (1955).

Younger, D. H. Recognition and parsing of context-free languages in time n^3 . Information and Control 10,2, (1967) : 189-208.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย



APPENDICES

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

APPENDIX A

HTML ELEMENT SPECIFICATIONS

Illustration of the abbreviations used to describe elements

<i>Element</i>	<i>Description</i>
HTML	An HTML Document
HEAD	Document Meta-Information
BASE	Base URL of the document
ISINDEX	Searchable document
LINK	Relationships to other resources
META	Meta-Information
SCRIPT	Program scripts
STYLE	Document stylesheet
TITLE	Document title
BODY	Document Body
ADDRESS	Address information
BLOCKQUOTE	Block quotation
CENTER	Centered text
DIV	Block division of a document
H1-H6	Heading (1-6)
HR	Horizontal divider
MULTICOL	Multicolumn text
P	Paragraphs
PRE	Preformatted text
FORM	User input form
FIELDSET	Group related input elements
KEYGEN	Generate encrypted keys
LABEL	Label for input elements
INPUT	Input fields
SELECT	Selectable fields
OPTION	Option in selectable field
TEXTAREA	Text input region
DL	Description/Glossary list
DT	Term
DD	Description
OL	Ordered list
LI	List item
UL	Unordered list
LI	List item
MENU	Menu list
LI	List item

<i>Element</i>	<i>Description</i>
DIR	Directory list
LI	List item
TABLE	Table
CAPTION	Table Caption
COL	Column properties specifier
COLGROUP	Column group specification
THEAD	Table header grouping
TBODY	Table body grouping
TFOOT	Table footer grouping
TR	Table row
TD	Table data cell
TH	Table header cell

Semantic Phrase Markup

<i>Element</i>	<i>Description</i>
CITE	Citation
CODE	Typed computer code
DFN	Definition
EM	Emphasized text
KBD	Keyboard input
Q	Inline quotation
SAMP	Sample text
STRIKE	Struck-out text
STRONG	Strong emphasis
VAR	A variable

Physical Phrase Markup

<i>Element</i>	<i>Description</i>
B	Bold
BDO	Bidirectional override
DIG	Bigger text
BLINK	Blinking text
FONT	Font size/face/color
I	Italics
MARQUEE	Scrolling marquee text
NOBR	No line breaks
S	Strike through
SMALL	Smaller text
SPAN	Stylesheet-specified styling
SUB	Subscript
SUP	Superscript

<i>Element</i>	<i>Description</i>
TT	Fixed-width font
U	Underline

Character Level Elements

<i>Element</i>	<i>Description</i>
BR	Line break
WBR	Optional Word Break
SPACER	Horizontal or vertical space

Inclusion Elements

<i>Element</i>	<i>Description</i>
APPLET	Embedded applet
PARAM	Parameter for applet
IMG	Inline image
IFRAME	Insert floating document frame
EMBED	Embed arbitrary data
NOEMBED	HTML alternative to EMBED
NOSCRIPT	HTML alternative to SCRIPT
OBJECT	Embed data and handler
PARAM	Parameter for object/handler
SERVER	Server-side scripting

Hypertext Relationships

<i>Element</i>	<i>Description</i>
A	Hypertext anchor

Meta-Information Elements

<i>Element</i>	<i>Description</i>
BASEFONT	Base font for document
BGSOUND	Background audio/sound
MAP	Client-side imagemap
AREA	Imagemap data

Netscape FRAME Documents

<i>Element</i>	<i>Description</i>
FRAMESET	Declare framed regions
FRAME	Specify frame contents
NOFRAMES	Markup for non-frame browsers

% Basic Structures of HTML

HTML Element: An HTML Document

Usage: <HTML> ... </HTML>

Can Contain: HEAD, BODY

Can Be Inside: nothing

HEAD Element: Document Meta-information

Usage: <HEAD> ... </HEAD>

Can Contain: BASE, ISINDEX, LINK, META, SCRIPT, STYLE, TITLE

Can Be Inside: HTML

BODY Element: Document Text Body

Usage: <BODY> ... </BODY>

Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, ISINDEX, INPUT, SELECT, TEXTAREA, DIR, DL, MENU, OL, UL, P, HR, Hn, ADDRESS, BLOCKQUOTE, CENTER, DIV, FORM, PRE, TABLE

Can Be Inside: HTML

BASE Element: Base URL

Usage: <BASE>

Can Contain: empty

Can Be Inside: HEAD

ISINDEX Element: Searchable Document

Usage: <ISINDEX>

Can Contain: empty

Can Be Inside: HEAD, BLOCKQUOTE, BODY, CENTER, DD, DIV, FORM, LI, TD, TH

LINK Element: Relationship to Other Documents

Usage: <LINK>

Can Contain: empty

Can Be Inside: HEAD

META Element: Document Meta-Information

Usage: <META>

Can Contain: empty

Can Be Inside: HEAD

SCRIPT Element: Include a Program Script

Usage: <SCRIPT> ... </SCRIPT>

Can Contain: script program code (characters)

Can Be Inside: HEAD, BODY, any BODY element that allows content

STYLE Element: Stylesheet or Rendering Information

Usage: <STYLE> ... </STYLE>

Can Contain: characters

Can Be Inside: HEAD

TITLE Element: Document Title

Usage: <TITLE> ... </TITLE>

Can Contain: characters

Can Be Inside: HEAD

Body Text Block and Heading Elements

ADDRESS Element: Address Information

Usage: <ADDRESS> ... </ADDRESS>

Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, INPUT, SELECT, TEXTAREA, P

Can Be Inside: BLOCKQUOTE, BODY, CENTER, DIV, FORM, TD, TH

BLOCKQUOTE Element: Block Quotations

Usage: <BLOCKQUOTE> ... </BLOCKQUOTE>

Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, ININDEX, INPUT, SELECT, TEXTAREA, DIR, DL, MENU, OL, UL, P, HR, Hn, ADDRESS, BLOCKQUOTE, CENTER, DIV, FORM, PRE, TABLE

Can Be Inside: BLOCKQUOTE, BODY, CENTER, DD, DIV, FORM, LI, TD, TH

CENTER Element: Center the Enclosed Text Horizontally

Usage: <CENTER> ... </CENTER>

Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, ISINDEX, INPUT, SELECT, TEXTAREA, DIR, DL, MENU, OL, UL, P, HR, Hn, ADDRESS, BLOCKQUOTE, CENTER, DIV, FORM, PRE, TABLE

Can Be Inside: BLOCKQUOTE, BODY, CENTER, DD, DIV, FORM, LI, TD, TH

DIV Element: A Block Division of the BODY

Usage: <DIV> ... </DIV>

Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, ISINDEX, INPUT, SELECT, TEXTAREA, DIR, DL, MENU, OL, UL, P, HR, Hn, ADDRESS, BLOCKQUOTE, CENTER, DIV, FORM, PRE, TABLE

Can Be Inside: BLOCKQUOTE, BODY, CENTER, DD, DIV, FORM, LI, TD, TH

Hn Elements: Headings

Usage: <Hn> ... </Hn>

Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, INPUT, SELECT, TEXTAREA

Can Be Inside: BLOCKQUOTE, BODY, CENTER, DIV, FORM, TD, TH

HR Element: Horizontal Rule

Usage: <HR>

Can Contain: empty

Can Be Inside: BLOCKQUOTE, BODY, CENTER, DD, DIV, FORM, LI, TD, TH

P Element: Paragraphs

Usage: <P> ... (</P>)

Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, ISINDEX, INPUT, SELECT, TEXTAREA

Can Be Inside: ADDRESS, BLOCKQUOTE, BODY, CENTER, DD, DIV, FORM, LI, TD, TH

PRE Element: Preformatted Text

Usage: <PRE> ... </PRE>

Can Contain: characters, B, CITE, CODE, DFN, EM, I, KBD, S, SAMP, STRIKE, STRONG, TT, U, VAR, A, APPLET, BR, MAP, SCRIPT, INPUT, SELECT, TEXTAREA

Can Be Inside: BLOCKQUOTE, BODY, CENTER, DD, DIV, FORM, LI, TD, TH

FORM Element: Fill-in Forms

Usage: <FORM> ... </FORM>

Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, ISINDEX, INPUT, SELECT, TEXTAREA, DIR, DL, MENU, OL, UL, P, HR, Hn, ADDRESS, BLOCKQUOTE, CENTER, DIV, PRE, TABLE

Can Be Inside: BLOCKQUOTE, BODY, CENTER, DD, DIV, FORM, LI, TD, TH

INPUT Element: Text Boxes, Checkboxes, and Radio Buttons

Usage: <INPUT>

Can Contain: empty

Can Be Inside: ADDRESS, BLOCKQUOTE, BODY, CENTER, DD, DIV, DT, FORM, LI, PRE, TD, TH, P, Hn, A, CAPTION, character highlighting

SELECT Element: Select from among Multiple Options

Usage: <SELECT> ... </SELECT>

Can Contain: OPTION

Can Be Inside: ADDRESS, BLOCKQUOTE, BODY, CENTER, DD, DIV, DT, FORM, LI, PRE, TD, TH, P, Hn, A, CAPTION, character highlighting

OPTION Element: List of Options for SELECT

Usage: <OPTION> ... </OPTION>

Can Contain: characters

Can Be Inside: SELECT

TEXTAREA Element: Text Input Region

Usage: <TEXTAREA> ... </TEXTAREA>

Can Contain: characters

Can Be Inside: ADDRESS, BLOCKQUOTE, BODY, CENTER, DD, DIV, DT, FORM, LI, PRE, TD, TH, P, Hn, A, CAPTION, character highlighting

KEYGEN Element: Generate Encrypted Keys (Netscape Navigator Only)

Usage: <KEYGEN>

Can Contain: empty

Can Be Inside: ADDRESS, BLOCKQUOTE, BODY, DD, DIV, DT, FORM, LI, PRE, TD, TH, CENTER, Hn, P, A, APPLET, CAPTION, character highlighting

% Lists and List-Related Elements

DL Element: Glossary List

Usage: <DL> ... </DL>

Can Contain: DT, DD

Can Be Inside: BLOCKQUOTE, BODY, CENTER, DD, DIV, FORM, LI, TD, TH

DT Element: Term in a Glossary List

Usage: <DT> ... (</DT>)

Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, INPUT, SELECT, TEXTAREA

Can Be Inside: DL

DD Element: Description in a Glossary List

Usage: <DD> ... </DD>

Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, ISINDEX, INPUT, SELECT, TEXTAREA, DIR, DL, MENU, OL, UL, P, HR, BLOCKQUOTE, CENTER, DIV, FORM, PRE, TABLE

Can Be Inside: DL

OL Element: Ordered List

Usage: ...

Can Contain: LI
 Can Be Inside: BLOCKQUOTE, BODY, CENTER, DD, DIV, FORM, LI, TD, TH

UL Element: Unordered List

Usage: ...
 Can Contain: LI
 Can Be Inside: BLOCKQUOTE, BODY, CENTER, DD, DIV, FORM, LI, TD, TH

DIR Element: Directory List

Usage: <DIR> ... </DIR>
 Can Contain: LI
 Can Be Inside: BLOCKQUOTE, BODY, CENTER, DD, DIV, FORM, LI, TD, TH

MENU Element: Menu List

Usage: <MENU> ... </MENU>
 Can Contain: LI
 Can Be Inside: BLOCKQUOTE, BODY, CENTER, DD, DIV, FORM, LI, TD, TH

LI Element: List Item

Usage: ...
 Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, ISINDEX, INPUT, SELECT, TEXTAREA, DIR, DL, MENU, OL, UL, P, HR, BLOCKQUOTE, CENTER, DIV, FORM, PRE, TABLE
 Can Be Inside: DIR, MENU, UL, OL

% Tables and Tabular Structures

TABLE Element: Tables and Tabular Structures

Usage: <TABLE> ... </TABLE>
 Can Contain: CAPTION, COL, COLGROUP, TBODY, TFOOT, THEAD, TR
 Can Be Inside: BLOCKQUOTE, BODY, CENTER, DD, DIV, FORM, LI, TD, TH

CAPTION Element: Table Caption

Usage: <CAPTION> ... </CAPTION>
 Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, INPUT, SELECT, TEXTAREA
 Can Be Inside: TABLE

COL Element: Specify Properties of a Column

Usage: <COL>
 Can Contain: empty
 Can Be Inside: COLGROUP, TABLE

COLGROUP Element: Properties of a Collection of Columns

Usage: <COLGROUP> ... </COLGROUP>
 Can Contain: COL
 Can Be Inside: TABLE

THEAD Element: Table Header

Usage: <THEAD> ... (</THEAD>)
 Can Contain: TR
 Can Be Inside: TABLE

TBODY Element: Table Body

Usage: <TBODY> ... (</TBODY>)
 Can Contain: TR
 Can Be Inside: TABLE

TFOOT Element: Table Footer

Usage: <TFOOT> ... (</TFOOT>)
 Can Contain: TR
 Can Be Inside: TABLE

TR Element: Table Row

Usage: <TR> ... (</TR>)
 Can Contain: TH, TD
 Can Be Inside: TABLE, TBODY, TFOOT, THEAD

TH Element: Table Headers

Usage: <TH> ... (</TH>)
 Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, ISINDEX, INPUT, SELECT, TEXTAREA, DIR, DL, MENU, OL, UL, P, HR, Hn, ADDRESS, BLOCKQUOTE, CENTER, DIV, FORM, PRE, TABLE
 Can Be Inside: TR

TD Element: Table Data

Usage: <TD> ... (</TD>)
 Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, ISINDEX, INPUT, SELECT, TEXTAREA, DIR, DL, MENU, OL, UL, P, HR, Hn, ADDRESS, BLOCKQUOTE, CENTER, DIV, FORM, PRE, TABLE
 Can Be Inside: TR

% Inclusion Elements**APPLET Element: Include an Embedded Applet**

Usage: <APPLET> ... </APPLET>
 Can Contain: characters, character highlighting, A, BR, IMG, BASEFONT, MAP, SCRIPT, INPUT, SELECT, TEXTAREA, PARAM
 Can Be Inside: ADDRESS, BLOCKQUOTE, BODY, CENTER, DIV, FORM, PRE, DD, DT, LI, P, TD, TH, Hn, A, CAPTION, character highlighting

PARAM Element: Define an Applet Parameter

Usage: <PARAM>
 Can Contain: empty
 Can Be Inside: APPLET, OBJECT

IMG Element: Inline Images

Usage:
 Can Contain: empty
 Can Be Inside: ADDRESS, BLOCKQUOTE, BODY, CENTER, DIV, FORM, DD, DT, LI, P, TD, TH, Hn, A, CAPTION, character highlighting

A Element: Hypertext Anchors

Usage: <A> ...
 Can Contain: probably: characters, character highlighting, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, INPUT, SELECT, TEXTAREA, SPACER, WBR, EMBED, NOEMBED, OBJECT
 Can Be Inside: probably: ADDRESS, BLOCKQUOTE, BODY, CENTER, DIV, FORM, PRE, DD, DT, LI, P, TD, TH, Hn, CAPTION, APPLET, NOEMBED, OBJECT, character highlighting

% Text/Phrase Markup Elements**Content Model for Highlighting Elements**

The context model for all the character highlighting elements is largely the same as the format below while *NAME* is one of CITE, CODE, DFN, EM, KBD, SAMP, STRIKE, STRONG, VAR, B, BIG, FONT, I, S, SMALL, SPAN, SUB, SUP, TT, or U.

Usage: <NAME> ... </NAME>
 Can Contain: characters, character highlighting, A, APPLET, BR, IMG, BASEFONT, MAP, SCRIPT, INPUT, SELECT, TEXTAREA
 Can Be Inside: ADDRESS, BLOCKQUOTE, BODY, CENTER, DIV, FORM, PRE, DD, DT, LI, P, TD, TH, Hn, A, CAPTION, character highlighting

Logical Highlighting Elements and Their Recommended Formatting

<i>Element</i>	<i>Meaning</i>	<i>Recommended Formatting</i>
CITE	A citation	italics
CODE	An example of typed code	fixed-width font

DFN	A definition	italics
EM	Emphasized text	italics
KBD	Keyboard input-for example	fixed-width
SAMP	A sequence of literal characters	fixed-width
STRIKE	Struck-out text	text with line
STRONG	Strong emphasis	boldface
VAR	A variable name	italics

Physical Highlighting Elements and Their Recommended Formatting

<i>Element</i>	<i>Meaning</i>
B	boldface
BIG	bigger text
FONT	font size, face, or color
I	italics
S	strike-through
SMALL	smaller-text
SPAN	stylesheet-specified formatting information
SUB	subscript
SUP	superscript
TT	fixed-width font
U	underlined

% Character-like Elements

BR Element: Line Break

Usage:	
Can Contain:	empty
Can Be Inside:	ADDRESS, BLOCKQUOTE, BODY, CENTER, DIV, FORM, PRE, DD, DT, LI, P, TD, TH, Hn, A, CAPTION, character highlighting

% Meta-Information Elements

BASEFONT Element: Set Default Font Characteristics

Usage:	<BASEFONT>
Can Contain:	empty
Can Be Inside:	ADDRESS, BLOCKQUOTE, BODY, CENTER, DIV, FORM, PRE, DD, DT, LI, P, TD, TH, Hn, A, CAPTION, character highlighting

MAP Element: Client-Side Imagemap Database

Usage:	<MAP> ... </MAP>
Can Contain:	AREA
Can Be Inside:	ADDRESS, BLOCKQUOTE, BODY, CENTER, DIV, FORM, PRE, DD, DT, LI, P, TD, TH, Hn, A, CAPTION, character highlighting

AREA Element: Client-Side Imagemap Mapping Areas

Usage: <AREA>
 Can Contain: empty
 Can Be Inside: MAP

% FRAME and Framed Documents

FRAME Element: Declare a FRAME Document

Usage: <FRAMESET> ... </FRAMESET>
 Can Contain: FRAME, FRAMESET, NOFRAMES
 Can Be Inside: HTML

FRAME Element: A FRAME within a FRAMESET

Usage: <FRAME>
 Can Contain: empty
 Can Be Inside: FRAMESET

NOFRAMES Element: Markup for FRAME-Incapable Browsers

Usage: <NOFRAMES> ... </NOFRAMES>
 Can Contain: characters, character highlighting, A, APPLET, BR, IMG, MAP, SCRIPT, CENTER, Hn, P, HR, ISINDEX, DIR, DL, MENU, OL, UL, ADDRESS, BLOCKQUOTE, DIV, FORM, PRE, TABLE, BODY
 Can Be Inside: FRAMESET

% Common HTML Extensions

BLINK Element: Blinking Text

Usage: <BLINK> ... </BLINK>
 Can Contain: Unspecified; probably: characters, character highlighting, A, BASEFONT, BR, IMG, MAP, NOSCRIPT, SCRIPT, SPACER, WBR, APPLET, EMBED, NOEMBED, OBJECT
 Can Be Inside: Unspecified; probably: ADDRESS, BLOCKQUOTE, BODY, CENTER, DIV, FORM, MULTICOL, PRE, Hn, DD, DT, LI, P, TD, TH, APPLET, NOEMBED, OBJECT, A, CAPTION, character highlighting

EMBED Element: Embed an Arbitrary Data Object

Usage: <EMBED>
 Can Contain: empty
 Can Be Inside: ADDRESS, BLOCKQUOTE, BODY, CENTER, DIV, FORM, MULTICOL, PRE, Hn, DD, DT, LI, P, TD, TH, NOEMBED, OBJECT, A, CAPTION, character highlighting

MULTICOL Element: Multicolumn Text

Usage: <MULTICOL> ... </MULTICOL>
 Can Contain: Unspecified; probably: characters, character highlighting, A, BASEFONT, BR, IMG, MAP, NOSCRIPT, SCRIPT, SPACER, ISINDEX, WBR, INPUT, SELECT, TEXTAREA, APPLET, EMBED,

NOEMBED, OBJECT, CENTER, Hn, P, HR, DIR, MENU, OL, UL, ADDRESS, BLOCKQUOTE, DIV, FORM, MULTICOL, PRE, TABLE

Can Be Inside: Unspecified; probably: ADDRESS, BLOCKQUOTE, BODY, CENTER, DIV, FORM, MULTICOL, DD, LI, TD, TH, APPLET, NOEMBED, OBJECT

NOBR Element: No Line Break

Usage: <NOBR> ... </NOBR>

Can Contain: Unspecified; probably: characters, character highlighting, A, BASEFONT, BR, IMG, MAP, NOSCRIPT, SCRIPT, SPACER, WBR, INPUT, SELECT, TEXTAREA, APPLET, EMBED, NOEMBED, OBJECT

Can Be Inside: Unspecified; probably: ADDRESS, BLOCKQUOTE, BODY, CENTER, DIV, FORM, MULTICOL, Hn, DD, DT, LI, P, TD, TH, APPLET, NOEMBED, OBJECT, A, CAPTION, character highlighting

NOEMBED Element: HTML Alternative to EMBED

Usage: <NOEMBED> ... </NOEMBED>

Can Contain: Unspecified; probably: characters, character highlighting, A, BR, IMG, MAP, NOSCRIPT, SCRIPT, SPACER, ISINDEX, WBR, INPUT, SELECT, TEXTAREA, APPLET, EMBED, NOEMBED, OBJECT, CENTER, Hn, P, HR, DIR, DL, MENU, OL, UL, ADDRESS, BLOCKQUOTE, DIV, FORM, PRE, TABLE, MULTICOL

Can Be Inside: Unspecified; probably: ADDRESS, BLOCKQUOTE, BODY, CENTER, DIV, FORM, MULTICOL, DD, LI, TD, TH, APPLET, OBJECT

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

APPENDIX B
CUNF GRAMMARS FOR HTML

Rule No.	Grammar Rules	
1	html :	c + html d + html Shtml + Ehtml body + Ehtml c + Ehtml d + Ehtml frameset + Ehtml head + Ehtml t + Ehtml
2	head :	Shead + Ehead Sbase + Ehead c + Ehead d + Ehead Sisindex + Ehead Slink + Ehead Smeta + Ehead script + Ehead style + Ehead t + Ehead title + Ehead
3	body :	Sbody + Ebody a + Ebody address + Ebody applet + Ebody b + Ebody Sbasefont + Ebody big + Ebody blink + Ebody blockquote + Ebody Sbr + Ebody c + Ebody center + Ebody cite + Ebody code + Ebody d + Ebody dfn + Ebody dir + Ebody div + Ebody di + Ebody em + Ebody Sembed + Ebody font + Ebody form + Ebody h1 + Ebody h2 + Ebody h3 + Ebody h4 + Ebody h5 + Ebody h6 + Ebody Shr + Ebody i + Ebody Simg + Ebody Sinput + Ebody Sisindex + Ebody kbd + Ebody Skeygen + Ebody map + Ebody menu + Ebody multicol + Ebody nobr + Ebody noembed + Ebody ol + Ebody p + Ebody Sp + Ebody pre + Ebody s + Ebody samp + Ebody script + Ebody select + Ebody small + Ebody span + Ebody strike + Ebody strong + Ebody sub + Ebody su
4	Sbase :	Sbase + c Sbase + t
5	Sisindex :	Sisindex + c Sisindex + t
6	Slink :	Slink + c Slink + t
7	Smeta :	Smeta + c Smeta + t
8	script :	Sscript + Escript c + Escript t + Escript
9	style :	Sstyle + Estyle c + Estyle t + Estyle
10	title :	Stitle + Etitle c + Etitle t + Etitle
11	address :	Saddress + Eaddress a + Eaddress applet + Eaddress b + Eaddress Sbasefont + Eaddress big + Eaddress blink + Eaddress Sbr + Eaddress c + Eaddress cite + Eaddress code + Eaddress dfn + Eaddress em + Eaddress Sembed + Eaddress font + Eaddress i + Eaddress Simg + Eaddress Sinput + Eaddress kbd + Eaddress Skeygen + Eaddress map + Eaddress multicol + Eaddress nobr + Eaddress noembed + Eaddress p + Eaddress Sp + Eaddress s + Eaddress samp + Eaddress script + Eaddress select + Eaddress small + Eaddress span + Eaddress strike + Eaddress strong + Eaddress sub + Eaddress sup + Eaddress t + Eaddress textarea + Eaddress tt + Eaddress u + Eaddress var + Eaddress

12	blockquote :	Sblockquote + Eblockquote a + Eblockquote address + Eblockquote applet + Eblockquote b + Eblockquote Sbasefont + Eblockquote big + Eblockquote blink + Eblockquote blockquote + Eblockquote Sbr + Eblockquote c + Eblockquote center + Eblockquote cite + Eblockquote code + Eblockquote dfn + Eblockquote dir + Eblockquote div + Eblockquote dl + Eblockquote em + Eblockquote Sembed + Eblockquote font + Eblockquote form + Eblockquote h1 + Eblockquote h2 + Eblockquote h3 + Eblockquote h4 + Eblockquote h5 + Eblockquote h6 + Eblockquote Shr + Eblockquote i + Eblockquote Simg + Eblockquote inindex + Eblockquote Sinput + Eblockquote Sisindex + Eblockquote kbd + Eblockquote Skeygen + Eblockquote map + Eblockquote menu + Eblockquote multicol + Eblockquote nobr + Eblockquote noembed + Eblock
13	center :	Scenter + Ecenter a + Ecenter address + Ecenter applet + Ecenter b + Ecenter Sbasefont + Ecenter big + Ecenter blink + Ecenter blockquote + Ecenter Sbr + Ecenter c + Ecenter center + Ecenter cite + Ecenter code + Ecenter dfn + Ecenter dir + Ecenter div + Ecenter dl + Ecenter em + Ecenter Sembed + Ecenter font + Ecenter form + Ecenter h1 + Ecenter h2 + Ecenter h3 + Ecenter h4 + Ecenter h5 + Ecenter h6 + Ecenter Shr + Ecenter i + Ecenter Simg + Ecenter Sinput + Ecenter Sisindex + Ecenter kbd + Ecenter Skeygen + Ecenter map + Ecenter menu + Ecenter multicol + Ecenter nobr + Ecenter noembed + Ecenter ol + Ecenter p + Ecenter Sp + Ecenter pre + Ecenter s + Ecenter samp + Ecenter script + Ecenter select + Ecenter small + Ecenter
14	div :	Sdiv + Ediv a + Ediv address + Ediv applet + Ediv b + Ediv Sbasefont + Ediv big + Ediv blink + Ediv blockquote + Ediv Sbr + Ediv c + Ediv center + Ediv cite + Ediv code + Ediv dfn + Ediv dir + Ediv div + Ediv dl + Ediv em + Ediv Sembed + Ediv font + Ediv form + Ediv h1 + Ediv h2 + Ediv h3 + Ediv h4 + Ediv h5 + Ediv h6 + Ediv Shr + Ediv i + Ediv Simg + Ediv Sinput + Ediv Sisindex + Ediv kbd + Ediv Skeygen + Ediv map + Ediv menu + Ediv multicol + Ediv nobr + Ediv noembed + Ediv ol + Ediv p + Ediv Sp + Ediv pre + Ediv s + Ediv samp + Ediv script + Ediv select + Ediv small + Ediv span + Ediv strike + Ediv strong + Ediv sub + Ediv sup + Ediv t + Ediv table + Ediv textarea + Ediv
15	h1 :	Sh1 + Eh1 a + Eh1 applet + Eh1 b + Eh1 Sbasefont + Eh1 big + Eh1 blink + Eh1 Sbr + Eh1 c + Eh1 cite + Eh1 code + Eh1 dfn + Eh1 em + Eh1 Sembed + Eh1 font + Eh1 i + Eh1 Simg + Eh1 Sinput + Eh1 kbd + Eh1 Skeygen + Eh1 map + Eh1 nobr + Eh1 s + Eh1 samp + Eh1 script + Eh1 select + Eh1 small + Eh1 span + Eh1 strike + Eh1 strong + Eh1 sub + Eh1 sup + Eh1 t + Eh1 textarea + Eh1 tt + Eh1 u + Eh1 var + Eh1
16	h2 :	Sh2 + Eh2 a + Eh2 applet + Eh2 b + Eh2 Sbasefont + Eh2 big + Eh2

		blink + Eh2 Sbr + Eh2 c + Eh2 cite + Eh2 code + Eh2 dfn + Eh2 em + Eh2 Sembed + Eh2 font + Eh2 i + Eh2 Simg + Eh2 Sinput + Eh2 kbd + Eh2 Skeygen + Eh2 map + Eh2 nobr + Eh2 s + Eh2 samp + Eh2 script + Eh2 select + Eh2 small + Eh2 span + Eh2 strike + Eh2 strong + Eh2 sub + Eh2 sup + Eh2 t + Eh2 textarea + Eh2 tt + Eh2 u + Eh2 var + Eh2
17	h3:	Sh3 + Eh3 a + Eh3 applet + Eh3 b + Eh3 Sbasefont + Eh3 big + Eh3 blink + Eh3 Sbr + Eh3 c + Eh3 cite + Eh3 code + Eh3 dfn + Eh3 em + Eh3 Sembed + Eh3 font + Eh3 i + Eh3 Simg + Eh3 Sinput + Eh3 kbd + Eh3 Skeygen + Eh3 map + Eh3 nobr + Eh3 s + Eh3 samp + Eh3 script + Eh3 select + Eh3 small + Eh3 span + Eh3 strike + Eh3 strong + Eh3 sub + Eh3 sup + Eh3 t + Eh3 textarea + Eh3 tt + Eh3 u + Eh3 var + Eh3
18	h4:	Sh4 + Eh4 a + Eh4 applet + Eh4 b + Eh4 Sbasefont + Eh4 big + Eh4 blink + Eh4 Sbr + Eh4 c + Eh4 cite + Eh4 code + Eh4 dfn + Eh4 em + Eh4 Sembed + Eh4 font + Eh4 i + Eh4 Simg + Eh4 Sinput + Eh4 kbd + Eh4 Skeygen + Eh4 map + Eh4 nobr + Eh4 s + Eh4 samp + Eh4 script + Eh4 select + Eh4 small + Eh4 span + Eh4 strike + Eh4 strong + Eh4 sub + Eh4 sup + Eh4 t + Eh4 textarea + Eh4 tt + Eh4 u + Eh4 var + Eh4
19	h5:	Sh5 + Eh5 a + Eh5 applet + Eh5 b + Eh5 Sbasefont + Eh5 big + Eh5 blink + Eh5 Sbr + Eh5 c + Eh5 cite + Eh5 code + Eh5 dfn + Eh5 em + Eh5 Sembed + Eh5 font + Eh5 i + Eh5 Simg + Eh5 Sinput + Eh5 kbd + Eh5 Skeygen + Eh5 map + Eh5 nobr + Eh5 s + Eh5 samp + Eh5 script + Eh5 select + Eh5 small + Eh5 span + Eh5 strike + Eh5 strong + Eh5 sub + Eh5 sup + Eh5 t + Eh5 textarea + Eh5 tt + Eh5 u + Eh5 var + Eh5
20	h6:	Sh6 + Eh6 a + Eh6 applet + Eh6 b + Eh6 Sbasefont + Eh6 big + Eh6 blink + Eh6 Sbr + Eh6 c + Eh6 cite + Eh6 code + Eh6 dfn + Eh6 em + Eh6 Sembed + Eh6 font + Eh6 i + Eh6 Simg + Eh6 Sinput + Eh6 kbd + Eh6 Skeygen + Eh6 map + Eh6 nobr + Eh6 s + Eh6 samp + Eh6 script + Eh6 select + Eh6 small + Eh6 span + Eh6 strike + Eh6 strong + Eh6 sub + Eh6 sup + Eh6 t + Eh6 textarea + Eh6 tt + Eh6 u + Eh6 var + Eh6
21	Shr:	Shr + c Shr + t
22	p:	Sp + Ep a + Ep applet + Ep b + Ep Sbasefont + Ep big + Ep blink + Ep Sbr + Ep c + Ep cite + Ep code + Ep dfn + Ep em + Ep Sembed + Ep font + Ep i + Ep Simg + Ep Sinput + Ep Sisindex + Ep kbd + Ep Skeygen + Ep map + Ep nobr + Ep s + Ep samp + Ep script + Ep select + Ep small + Ep span + Ep strike + Ep strong + Ep sub + Ep sup + Ep t + Ep textarea + Ep tt + Ep u + Ep var + Ep

23	Sp :	Sp + a Sp + applet Sp + b Sp + Sbasefont Sp + big Sp + blink Sp + Sbr Sp + c Sp + cite Sp + code Sp + dfn Sp + em Sp + Sembed Sp + font Sp + i Sp + Simg Sp + Sinput Sp + Sisindex Sp + kbd Sp + Skeygen Sp + map Sp + nobr Sp + s Sp + samp Sp + script Sp + select Sp + small Sp + span Sp + strike Sp + strong Sp + sub Sp + sup Sp + t Sp + textarea Sp + tt Sp + u Sp + var
24	pre :	Spre + Epre a + Epre applet + Epre b + Epre Sbasefont + Epre blink + Epre Sbr + Epre c + Epre cite + Epre code + Epre dfn + Epre em + Epre Sembed + Epre i + Epre Sinput + Epre kbd + Epre Skeygen + Epre map + Epre s + Epre samp + Epre script + Epre select + Epre span + Epre strike + Epre strong + Epre t + Epre textarea + Epre tt + Epre u + Epre var + Epre
25	form :	Sform + Eform a + Eform address + Eform applet + Eform b + Eform Sbasefont + Eform big + Eform blink + Eform blockquote + Eform Sbr + Eform c + Eform center + Eform cite + Eform code + Eform dfn + Eform dir + Eform div + Eform dl + Eform em + Eform Sembed + Eform font + Eform form + Eform h1 + Eform h2 + Eform h3 + Eform h4 + Eform h5 + Eform h6 + Eform Shr + Eform i + Eform Simg + Eform Sinput + Eform Sisindex + Eform kbd + Eform Skeygen + Eform map + Eform menu + Eform multicol + Eform nobr + Eform noembed + Eform ol + Eform p + Eform Sp + Eform pre + Eform s + Eform samp + Eform script + Eform select + Eform small + Eform span + Eform strike + Eform strong + Eform sub + Eform sup + Eform
26	Sinput :	Sinput + c Sinput + t
27	select :	Sselect + Eselect c + Eselect option + Eselect Soption + Eselect t + Eselect
28	option :	Soption + Eoption c + Eoption t + Eoption
29	Soption :	Soption + c Soption + t
30	textarea :	Stextarea + Etextarea c + Etextarea t + Etextarea
31	Skeygen :	Skeygen + c Skeygen + t
32	dl :	Sdl + Edl b + Edl c + Edl dd + Edl Sdd + Edl dl + Edl dt + Edl Sdt + Edl em + Edl i + Edl p + Edl Sp + Edl samp + Edl t + Edl
33	dt :	Sdt + Edt a + Edt applet + Edt b + Edt Sbasefont + Edt big + Edt blink + Edt Sbr + Edt c + Edt cite + Edt code + Edt dfn + Edt em + Edt Sembed + Edt font + Edt i + Edt Simg + Edt Sinput + Edt kbd + Edt Skeygen + Edt map + Edt nobr + Edt s + Edt samp + Edt script + Edt select + Edt small + Edt span + Edt strike + Edt strong + Edt sub + Edt sup + Edt t + Edt textarea + Edt tt + Edt u + Edt var + Edt
34	Sdt :	Sdt + a Sdt + applet Sdt + b Sdt + Sbasefont Sdt + big Sdt + blink Sdt + Sbr Sdt + c Sdt + cite Sdt + code Sdt + dfn Sdt + em Sdt + Sembed Sdt + font Sdt + i Sdt + Simg Sdt + Sinput Sdt + kbd Sdt +

		Skeygen Sdt+map Sdt+nobr Sdt+s Sdt+samp Sdt+script Sdt+select Sdt+small Sdt+span Sdt+strike Sdt+strong Sdt+sub Sdt+sup Sdt+t Sdt+textarea Sdt+tt Sdt+u Sdt+var
35	dd :	Sdd+Edd a+Edd applet+Edd b+Edd Sbasefont+Edd big+Edd blink+Edd blockquote+Edd Sbr+Edd c+Edd center+Edd cite+Edd code+Edd dfn+Edd dir+Edd div+Edd dl+Edd em+Edd Sembed+Edd font+Edd form+Edd Shr+Edd i+Edd Simg+Edd Sinput+Edd Sisindex+Edd kbd+Edd Skeygen+Edd map+Edd menu+Edd multicol+Edd nobr+Edd noembed+Edd ol+Edd p+Edd Sp+Edd pre+Edd s+Edd samp+Edd script+Edd select+Edd small+Edd span+Edd strike+Edd strong+Edd sub+Edd sup+Edd t+Edd table+Edd textarea+Edd tt+Edd u+Edd ul+Edd var+Edd
36	Sdd :	Sdd+a Sdd+applet Sdd+b Sdd+Sbasefont Sdd+big Sdd+blink Sdd+blockquote Sdd+Sbr Sdd+c Sdd+center Sdd+cite Sdd+code Sdd+dfn Sdd+dir Sdd+div Sdd+dl Sdd+em Sdd+Sembed Sdd+font Sdd+form Sdd+Shr Sdd+i Sdd+Simg Sdd+Sinput Sdd+Sisindex Sdd+kbd Sdd+Skeygen Sdd+map Sdd+menu Sdd+multicol Sdd+nobr Sdd+noembed Sdd+ol Sdd+Sp Sdd+p Sdd+pre Sdd+s Sdd+samp Sdd+script Sdd+select Sdd+small Sdd+span Sdd+strike Sdd+strong Sdd+sub Sdd+sup Sdd+t Sdd+table Sdd+textarea Sdd+tt Sdd+u Sdd+ul Sdd+var
37	ol :	Sol+Eol c+Eol li+Eol Sli+Eol t+Eol
38	ul :	Sul+Eul c+Eul li+Eul Sli+Eul t+Eul
39	dir :	Sdir+Edir c+Edir li+Edir Sli+Edir t+Edir
40	menu :	Smenu+Emenu c+Emenu li+Emenu Sli+Emenu t+Emenu
41	li :	Sli+Eli a+Eli applet+Eli b+Eli Sbasefont+Eli big+Eli blink+Eli blockquote+Eli Sbr+Eli c+Eli center+Eli cite+Eli code+Eli dfn+Eli dir+Eli div+Eli dl+Eli em+Eli Sembed+Eli font+Eli form+Eli Shr+Eli i+Eli Simg+Eli Sinput+Eli Sisindex+Eli kbd+Eli Skeygen+Eli map+Eli menu+Eli multicol+Eli nobr+Eli noembed+Eli ol+Eli p+Eli Sp+Eli pre+Eli s+Eli samp+Eli script+Eli select+Eli small+Eli span+Eli strike+Eli strong+Eli sub+Eli sup+Eli t+Eli table+Eli textarea+Eli tt+Eli u+Eli ul+Eli var+Eli
42	Sli :	Sli+a Sli+applet Sli+b Sli+Sbasefont Sli+big Sli+blink Sli+blockquote Sli+Sbr Sli+c Sli+center Sli+cite Sli+code Sli+dfn Sli+dir Sli+div Sli+dl Sli+em Sli+Sembed Sli+font Sli+form Sli+Shr Sli+i Sli+Simg Sli+Sinput Sli+Sisindex Sli+kbd Sli+

		Skeygen Sli + map Sli + menu Sli + multicol Sli + nobr Sli + noembed Sli + ol Sli + Sp Sli + p Sli + pre Sli + s Sli + samp Sli + script Sli + select Sli + small Sli + span Sli + strike Sli + strong Sli + sub Sli + sup Sli + t Sli + table Sli + textarea Sli + tt Sli + u Sli + ul Sli + var
43	table :	Stable + Etable c + Etable caption + Etable Scol + Etable colgroup + Etable t + Etable tbody + Etable Stbody + Etable tfoot + Etable tfoot + Etable thead + Etable Sthead + Etable tr + Etable Str + Etable
44	caption :	Scaption + Ecaption a + Ecaption applet + Ecaption b + Ecaption Sbasefont + Ecaption big + Ecaption blink + Ecaption Sbr + Ecaption c + Ecaption cite + Ecaption code + Ecaption dfn + Ecaption em + Ecaption Sembed + Ecaption font + Ecaption i + Ecaption Simg + Ecaption Sinput + Ecaption kbd + Ecaption Skeygen + Ecaption map + Ecaption nobr + Ecaption s + Ecaption samp + Ecaption script + Ecaption select + Ecaption small + Ecaption span + Ecaption strike + Ecaption strong + Ecaption sub + Ecaption sup + Ecaption t + Ecaption textarea + Ecaption tt + Ecaption u + Ecaption var + Ecaption
45	Scol :	Scol + c Scol + t
46	colgroup :	Scolgroup + Ecolgroup c + Ecolgroup Scol + Ecolgroup t + Ecolgroup
47	thead :	Sthead + Ethead c + Ethead t + Ethead tr + Ethead Str + Ethead
48	Sthead :	Sthead + c Sthead + t Sihead + Str Sthead + tr
49	tbody :	Stbody + Etbody c + Etbody t + Etbody tr + Etbody Str + Etbody
50	Stbody :	Stbody + c Stbody + t Stbody + Str Stbody + tr
51	tfoot :	Stfoot + Etfoot c + Etfoot t + Etfoot tr + Etfoot Str + Etfoot
52	Stfoot :	Stfoot + c Stfoot + t Stfoot + Str Stfoot + tr
53	tr :	Str + Etr c + Etr t + Etr td + Etr Std + Etr th + Etr Sth + Etr
54	Str :	Str + c Str + t Str + Std Str + td Str + Sth Str + th
55	th :	Sth + Eth a + Eth address + Eth applet + Eth b + Eth Sbasefont + Eth big + Eth blink + Eth blockquote + Eth Sbr + Eth c + Eth center + Eth cite + Eth code + Eth dfn + Eth dir + Eth div + Eth dl + Eth em + Eth Sembed + Eth font + Eth form + Eth h1 + Eth h2 + Eth h3 + Eth h4 + Eth h5 + Eth h6 + Eth Shr + Eth i + Eth Simg + Eth Sinput + Eth Sisindex + Eth kbd + Eth Skeygen + Eth map + Eth menu + Eth multicol + Eth nobr + Eth noembed + Eth ol + Eth p + Eth Sp + Eth pre + Eth s + Eth samp + Eth script + Eth select + Eth small + Eth span + Eth strike + Eth strong + Eth sub + Etr sup + Eth t + Eth table + Eth textarea + Eth tt + Eth u + Eth ul + Eth var + Eth
56	Sth :	Sth + a Sth + address Sth + applet Sth + b Sth + Sbasefont Sth + big Sth + blink Sth + blockquote Sth + Sbr Sth + c Sth + center Sth + cite Sth + code Sth + dfn Sth + dir Sth + div Sth + dl Sth + em Sth +

		Sembed Sth + font Sth + form Sth + h1 Sth + h2 Sth + h3 Sth + h4 Sth + h5 Sth + h6 Sth + Shr Sth + i Sth + Simg Sth + Sinput Sth + Sisindex Sth + kbd Sth + Skeygen Sth + map Sth + menu Sth + multicol Sth + nobr Sth + noembed Sth + ol Sth + Sp Sth + p Sth + pre Sth + s Sth + samp Sth + script Sth + select Sth + small Sth + span Sth + strike Sth + strong Sth + sub Sth + sup Sth + t Sth + table Sth + textarea Sth + tt Sth + u Sth + ul Sth + var
57	td :	Std + Etd a + Etd address + Etd applet + Etd b + Etd Sbasefont + Etd big + Etd blink + Etd blockquote + Etd Sbr + Etd c + Etd center + Etd cite + Etd code + Etd dfn + Etd dir + Etd div + Etd dl + Etd em + Etd Sembed + Etd font + Etd form + Etd h1 + Etd h2 + Etd h3 + Etd h4 + Etd h5 + Etd h6 + Etd Shr + Etd i + Etd Simg + Etd Sinput + Etd Sisindex + Etd kbd + Etd Skeygen + Etd map + Etd menu + Etd multicol + Etd nobr + Etd noembed + Etd ol + Etd p + Etd Sp + Etd pre + Etd s + Etd samp + Etd script + Etd select + Etd small + Etd span + Etd strike + Etd strong + Etd sub + Etd sup + Etd t + Etd table + Etd textarea + Etd tt + Etd u + Etd u' + Etd var + Etd
58	Std :	Std + a Std + address Std + applet Std + b Std + Sbasefont Std + big Std + blink Std + blockquote Std + Sbr Std + c Std + center Std + cite Std + code Std + dfn Std + dir Std + div Std + dl Std + em Std + Sembed Std + font Std + form Std + h1 Std + h2 Std + h3 Std + h4 Std + h5 Std + h6 Std + Shr Std + i Std + Simg Std + Sinput Std + Sisindex Std + kbd Std + Skeygen Std + map Std + menu Std + multicol Std + nobr Std + noembed Std + ol Std + Sp Std + p Std + pre Std + s Std + samp Std + script Std + select Std + small Std + span Std + strike Std + strong Std + sub Std + sup Std + t Std + table Std + textarea Std + tt Std + u Std + ul Std + var
59	applet :	Sapplet + Eapplet a + Eapplet Sbasefont + Eapplet blink + Eapplet Sbr + Eapplet c + Eapplet Simg + Eapplet Sinput + Eapplet Skeygen + Eapplet map + Eapplet multicol + Eapplet nobr + Eapplet noembed + Eapplet Sparam + Eapplet script + Eapplet select + Eapplet span + Eapplet t + Eapplet textarea + Eapplet
60	Sparam :	Sparam + c Sparam + t
61	Simg :	Simg + c Simg + t
62	a :	Sa + Ea applet + Ea b + Ea Sbasefont + Ea big + Ea blink + Ea Sbr + Ea c + Ea cite + Ea code + Ea dfn + Ea em + Ea Sembed + Ea font + Ea i + Ea Simg + Ea Sinput + Ea kbd + Ea Skeygen + Ea map + Ea nobr + Ea noembed + Ea object + Ea s + Ea samp + Ea script + Ea select + Ea small + Ea Sspacer + Ea span + Ea strike + Ea strong + Ea sub + Ea sup + Ea t + Ea textarea + Ea tt + Ea u + Ea var + Ea Swbr + Ea

63	cite :	Scite + Ecite a + Ecite applet + Ecite Sbasefont + Ecite Sbr + Ecite c + Ecite Simg + Ecite Sinput + Ecite map + Ecite script + Ecite select + Ecite t + Ecite textarea + Ecite
64	code :	Scode + Ecode a + Ecode applet + Ecode Sbasefont + Ecode Sbr + Ecode c + Ecode Simg + Ecode Sinput + Ecode map + Ecode script + Ecode select + Ecode t + Ecode textarea + Ecode
65	dfn :	Sdfn + Edfn a + Edfn applet + Edfn Sbasefont + Edfn Sbr + Edfn c + Edfn Simg + Edfn Sinput + Edfn map + Edfn script + Edfn select + Edfn t + Edfn textarea + Edfn
66	em :	Sem + Eem a + Eem applet + Eem b + Eem Sbasefont + Eem Sbr + Eem c + Eem Simg + Eem Sinput + Eem map + Eem samp + Eem script + Eem select + Eem t + Eem textarea + Eem
67	kbd :	Skbd + Ekbd a + Ekbd applet + Ekbd Sbasefont + Ekbd Sbr + Ekbd c + Ekbd Simg + Ekbd Sinput + Ekbd map + Ekbd script + Ekbd select + Ekbd t + Ekbd textarea + Ekbd
68	samp :	Ssamp + Esamp a + Esamp applet + Esamp Sbasefont + Esamp Sbr + Esamp c + Esamp em + Esamp Simg + Esamp Sinput + Esamp map + Esamp script + Esamp select + Esamp t + Esamp textarea + Esamp
69	strike :	Sstrike + Estrike a + Estrike applet + Estrike Sbasefont + Estrike Sbr + Estrike c + Estrike Simg + Estrike Sinput + Estrike map + Estrike script + Estrike select + Estrike t + Estrike textarea + Estrike
70	strong :	Sstrong + Estrong a + Estrong applet + Estrong Sbasefont + Estrong Sbr + Estrong c + Estrong Simg + Estrong Sinput + Estrong map + Estrong script + Estrong select + Estrong t + Estrong textarea + Estrong
71	var :	Svar + Evar a + Evar applet + Evar Sbasefont + Evar Sbr + Evar c + Evar Simg + Evar Sinput + Evar map + Evar script + Evar select + Evar t + Evar textarea + Evar
72	b :	Sb + Eb a + Eb applet + Eb Sbasefont + Eb Sbr + Eb c + Eb font + Eb Simg + Eb Sinput + Eb map + Eb script + Eb select + Eb t + Eb textarea + Eb
73	big :	Sbig + Ebig a + Ebig applet + Ebig Sbasefont + Ebig Sbr + Ebig c + Ebig Simg + Ebig Sinput + Ebig map + Ebig script + Ebig select + Ebig t + Ebig textarea + Ebig
74	font :	Sfont + Efont a + Efont applet + Efont Sbasefont + Efont Sbr + Efont c + Efont Simg + Efont Sinput + Efont map + Efont script + Efont select + Efont t + Efont textarea + Efont
75	i :	Si + Ei a + Ei applet + Ei b + Ei Sbasefont + Ei Sbr + Ei c + Ei Simg + Ei Sinput + Ei map + Ei samp + Ei script + Ei select + Ei sub + Ei t + Ei textarea + Ei
76	s :	Ss + Es a + Es applet + Es Sbasefont + Es Sbr + Es c + Es Simg + Es Sinput + Es map + Es script + Es select + Es t + Es textarea + Es

77	small :	Ssmall + Esmall a + Esmall applet + Esmall Sbasefont + Esmall Sbr + Esmall c + Esmall Simg + Esmall Sinput + Esmall map + Esmall script + Esmall select + Esmall t + Esmall textarea + Esmall
78	span :	Sspan + Espan a + Espan applet + Espan Sbasefont + Espan Sbr + Espan c + Espan Sembed + Espan Simg + Espan Sinput + Espan map + Espan noembed + Espan noscript + Espan object + Espan script + Espan select + Espan Sspacer + Espan t + Espan textarea + Espan Swbr + Espan .
79	sub :	Ssub + Esub a + Esub applet + Esub Sbasefont + Esub Sbr + Esub c + Esub Simg + Esub Sinput + Esub map + Esub script + Esub select + Esub t + Esub textarea + Esub
80	sup :	Ssup + Esup a + Esup applet + Esup Sbasefont + Esup Sbr + Esup c + Esup Simg + Esup Sinput + Esup map + Esup script + Esup select + Esup t + Esup textarea + Esup
81	tt :	Stt + Ett a + Ett applet + Ett Sbasefont + Ett Sbr + Ett c + Ett Simg + Ett Sinput + Ett map + Ett script + Ett select + Ett t + Ett textarea + Ett
82	u :	Su + Eu a + Eu applet + Eu Sbasefont + Eu Sbr + Eu c + Eu Simg + Eu Sinput + Eu map + Eu script + Eu select + Eu t + Eu textarea + Eu
83	Sbr :	Sbr + c Sbr + t
84	Sbasefont :	Sbasefont + c Sbasefont + t
85	map :	Smap + Emap Sarea + Emap c + Emap t + Emap
86	Sarea :	Sarea + c Sarea + t
87	frameset :	Sframeset + Eframeset c + Eframeset Sframe + Eframeset frameset + Eframeset noframes + Eframeset t + Eframeset
88	Sframe :	Sframe + c Sframe + t
89	noframes :	Snoframes + Enoframes address + Enoframes applet + Enoframes blockquote + Enoframes body + Enoframes Sbr + Enoframes c + Enoframes center + Enoframes dir + Enoframes div + Enoframes dl + Enoframes form + Enoframes h1 + Enoframes h2 + Enoframes h3 + Enoframes h4 + Enoframes h5 + Enoframes h6 + Enoframes Shr + Enoframes Simg + Enoframes Sisindex + Enoframes map + Enoframes menu + Enoframes ol + Enoframes p + Enoframes Sp + Enoframes pre + Enoframes script + Enoframes t + Enoframes table + Enoframes ul + Enoframes
90	blink :	Sblink + Eblink a + Eblink applet + Eblink Sbasefont + Eblink Sbr + Eblink c + Eblink Sembed + Eblink Simg + Eblink map + Eblink noembed + Eblink noscript + Eblink object + Eblink script + Eblink Sspacer + Eblink t + Eblink Swbr + Eblink
91	Sembed :	Sembed + c Sembed + t
92	multicol :	Smulticol + Emulticol a + Emulticol address + Emulticol applet + Emulticol Sbasefont + Emulticol blink + Emulticol blockquote + Emulticol Sbr + Emulticol

		c + Emulticol center + Emulticol dir + Emulticol div + Emulticol Sembed + Emulticol form + Emulticol h1 + Emulticol h2 + Emulticol h3 + Emulticol h4 + Emulticol h5 + Emulticol h6 + Emulticol Shr + Emulticol Simg + Emulticol Sinput + Emulticol Sisindex + Emulticol map + Emulticol menu + Emulticol multicol + Emulticol nobr + Emulticol noembed + Emulticol noscript + Emulticol object + Emulticol ol + Emulticol p + Emulticol Sp + Emulticol pre + Emulticol script + Emulticol select + Emulticol Sspacer + Emulticol span + Emulticol t + Emulticol table + Emulticol textarea + Emulticol ul + Emulticol Swbr + Emultic
93	nobr :	Snobr + Enobr a + Enobr applet + Enobr Sbasefont + Enobr Sbr + Enobr c + Enobr Sembed + Enobr Simg + Enobr Sinput + Enobr map + Enobr noembed + Enobr noscript + Enobr object + Enobr script + Enobr select + Enobr Sspacer + Enobr t + Enobr textarea + Enobr Swbr + Enobr
94	noembed :	Snoembed + Enoembed a + Enoembed address + Enoembed applet + Enoembed blink + Enoembed blockquote + Enoembed Sbr + Enoembed c + Enoembed center + Enoembed dir + Enoembed div + Enoembed dl + Enoembed Sembed + Enoembed form + Enoembed h1 + Enoembed h2 + Enoembed h3 + Enoembed h4 + Enoembed h5 + Enoembed h6 + Enoembed Shr + Enoembed Simg + Enoembed Sinput + Enoembed Sisindex + Enoembed map + Enoembed menu + Enoembed multicol + Enoembed nobr + Enoembed noembed + Enoembed noscript + Enoembed object + Enoembed ol + Enoembed p + Enoembed Sp + Enoembed pre + Enoembed script + Enoembed select + Enoembed Sspacer + Enoembed span + Enoembed t + Enoembed table + Enoembed textarea + Enoembed ul + Enoembed Swbr + Enoembed

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

APPENDIX C

DATA-DIRECTED PARSER USER MANUAL

Application Source Code

The source code of application developed in this thesis is distributed in a single file named DDP.zip which could be unzipped to directory named "C:\DDP" without changing any file's contents. However, if user prefers another directory, one need to change the declaration in file's header as in an example below from 1 to 2 for all perl files (*.pl) .

- 1) @INC = qw(C:\Perl\lib C:\Perl\site\lib C:\DDP);
- 2) @INC = qw(C:\Perl\lib C:\Perl\site\lib C:\NewDirectory);

Perl Interpreter and Module

All files developed in this thesis are perl file format, and require a perl interpreter that works under Windows operating system. Perl interpreter used in an experiment is *Activeperl* which could be downloaded from <http://www.activestate.com>. Default directory for perl interpreter should be "C:\Perl", otherwise user need to change all perl's library files declared in file's header by changing from 1 to 2

- 1) @INC = qw(C:\Perl\lib C:\Perl\site\lib C:\DDP);
- 2) @INC = qw(C:\NewPerl\lib C:\NewPerl\site\lib C:\NewDirectory);

After that, user need to copy file Tokenizer.pm and HtmlTokenizer.pm from directory C:\DDP to directory C:\Perl\site\lib\Html which both of them were derived from module Parser.pm and TokeParser.pm respectively.

Using Application

Under the directory that user unzipped application files, then type

```
perl ddp.pl -i input [-o output] [-r recovery] [-s structure] [-h help]
```

such as

```
perl ddp.pl -i input.html -r 1 -s 1
```

Note:

- i : indicates input file that should be in HTML style.
- o : this parameter is optional, where it indicates output file's name (default name is [input].out).
- r : this parameter is optional, where 0 is common parsing and 1 is error recovery parsing.
- s : display structure, where 1 for all nodes in parse tree, 2 for the highest level only and 3 for both. This parameter is optional as well.
- h : help information.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

BIOGRAPHY

The author was born at Ranong province and had his primary school education at Sriarunothai School. He had a secondary education at Pichairatanakarn School, La Salle College and Samsen College respectively. He received a Bachelor of Engineering degree in computer engineering program from Chulalongkorn University in 1999. For his parent and interesting in natural language processing, he continued learning in master degree of computer engineering. In 2000, the author and his advisor, Assoc. Prof. Wanchai Rivepiboon, Ph.D. had one paper; Attribute-based parser for machine translation, published in SNLP 2000 that was held at Chiangmai province.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย