

A FORMAL LANGUAGE MODEL FOR
DETECTING AMBIGUITY IN SGML

By

RICHARD WALTER MATZEN

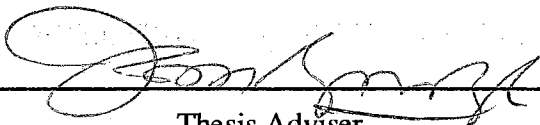
Bachelor of Science
University of Central Arkansas
Conway, Arkansas
1984

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1987

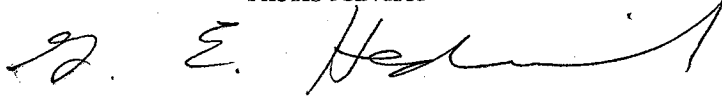
Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
December, 1993

A FORMAL LANGUAGE MODEL FOR
DETECTING AMBIGUITY IN SGML

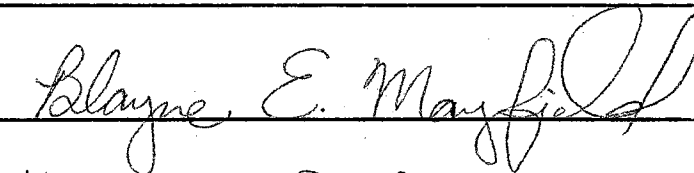
Thesis Approved:




Thesis Adviser



Blayne E. Mayfield



Wayne B. Powell



Dean of the Graduate College

PREFACE

The Standard Generalized Markup Language (SGML) is the first document processing standard to gain widespread acceptance. However, the lack of a formal language model for SGML causes problems. One such problem is detecting ambiguity in the document type definitions (DTDs) that are used to define classes of documents. There are two kinds of ambiguity that are defined and prohibited by the standard (ISO 8879). However, no complete methods have been shown for detecting and preventing these kinds of ambiguity. The definitions of ambiguity in the standard were revised; the revisions formalize the existing definitions and they clearly distinguish the kinds of ambiguity that can occur in a DTD. Then a language model was developed to recognize both DTDs and the classes of documents they define. Based on this language model, algorithms were developed for detecting ambiguity under the revised definitions. The revised definitions of ambiguity and the algorithms for detecting ambiguity resolve open questions regarding the definitions and their implementation.

I wish to express my appreciation to all who assisted me in this research and in my course of study at Oklahoma State University. I wish to thank my major adviser, Dr. G.E. Hedrick, for his contributions to this dissertation, for his expert instruction in language theory, and for his invaluable guidance. I wish to express my gratitude and respect to my dissertation adviser, Dr. K. M. George, for his guidance in this work and for the ideas and the rigor that he has contributed to it. I also wish to thank the other members of my advisory committee, Dr. W. B. Powell and Dr. B. E. Mayfield, for their advisement and their interest.

I am grateful to my coworkers at TMS who have given me support; especially to Dr. J. R. Phillips for his advice and encouragement and to Butch Taylor for the lively

discussions on automata theory. I am also grateful to my parents, to TMS Inc., and to the McAlester Scottish Rite Foundation for providing financial assistance. A special thank-you is due to my wife Jan for her help with the illustrations. I also wish to thank Anne Brueggemann-Klein for providing a copy of her paper on ambiguous content models.

Special thanks are due to all of my family: to my wife and sons for their understanding and patience, to my mother, sister, and brother for their support, and to the Chaney's for their positive encouragement. I wish to express my deepest appreciation to my father for his inspiration, his advice, and for believing in me.

TABLE OF CONTENTS

Chapter	Page
1. Introduction.	1
2. Literature review	5
2.1. Review of Language Theory Concepts	5
2.2. SGML Background.	8
2.2.1. SGML Syntax Productions	10
2.2.2. Ambiguous Content Models.	16
2.2.3. Ambiguity Caused By Omitted Tags.	19
2.2.4. Disambiguating Rules for SGML Parsers	20
2.3. Related Work in Language Theory.	21
2.4. Related Work on Ambiguity in SGML.	23
3. A Language Model for SGML DTDs.	29
3.1. Preliminary Definitions.	29
3.2. Algorithms	33
4. A Parser for DTDs	51
4.1. A System of Regular Expressions for DTDs	51
4.2. A System of Finite Automata for DTDs	52
4.3. A Parser for DTDs.	54
5. Ambiguous Model Groups.	58
5.1. Preliminary Definitions.	58
5.2. Algorithms	61
6. Ambiguous DTDs.	76
6.1. Systems of Finite Automata Recognizing Document Instances	77
6.2. Ambiguity Caused By Omitted Tags	84
7. Exceptions.	121
7.1. Preliminary Definitions.	121
7.2. The Effect of Exceptions on Content Models	124
7.3. A System of Finite Automata for Document Instances.	129
7.4. Exceptions and Ambiguity	134
7.5. Exception Validation	139

Chapter	Page
8. Conclusions and Future Work	140
8.1. Conclusions.	140
8.2. Recommendations for Future Work.	142
References.	143

LIST OF TABLES

Table	Page
2.1. The operators of syntax productions	10
4.1. DTD: parsing actions.	55
4.2. Element_declaration: parsing actions.	55
4.3. Model_group: parsing actions.	56
4.4. Content_token: parsing actions.	56
5.1. Model_group: indexing a model group	61
5.2. Content_token: indexing a model group	62
5.3. Model_group: constructing an equivalent regular expression for an indexed model group	65
5.4. Content_token: constructing an equivalent regular expression for an indexed model group	66
5.5. The NFA for (A AND B AND C) by Construction 2 .	70
5.6. Removing e-transitions and inaccessible states.	72
6.1. DTD: constructing an explicit DTD	79
6.2. Element_declaration: constructing an explicit DTD.	79
6.3. Correspondence for Example 6.4.	92
6.4. Correspondence for Example 6.5.	94
6.5. Correspondence for Example 6.6.	95
6.6. Correspondence for Example 6.7.	97
6.7. Correspondence for Example 6.8.	99

Table	Page
6.8. Correspondence for Example 6.9.	101
7.1. Dynamic content models for Example 7.3.	124
7.2. Construct a DCM list for a DTD.	126

LIST OF FIGURES

Figure	Page
2.1. Graph representations of NFAs.	6
2.2. The NFA for b in Σ	7
2.3. The NFA for $(s t)$	7
2.4. The NFA for (st)	8
2.5. The NFA for (s^*)	8
3.1. The NFA for A in Example 3.2	32
3.2. The NFA for B in Example 3.2	32
3.3. The NFA for A in Example 3.3	35
3.4. The NFA for B in Example 3.3	35
3.5. The NFA for A_1 in Example 3.4.	37
3.6. The NFA for B in Example 3.4	37
4.1. The NFA for DTD.	53
4.2. The NFA for element declaration.	53
4.3. The NFA for model group.	53
4.4. The NFA for content token.	54
5.1. The NFA for r'	67
5.2. The NFA for $M(m_1 \text{ PLUS})$	68
5.3. The NFA for $(C A)^+$ by Construction 1	68
5.4. The NFA for $(C A)^+$ by Thompson's Construction.	68
5.5. The NFA for $(A \text{ AND } B \text{ AND } C)$ by Construction 2.	70

Figure	Page
5.6. Removing e-transitions and inaccessible states .	71
6.1. The NFA for TOP in Example 6.3	83
6.2. The NFA for A in Example 6.3	83
6.3. The NFA for B in Example 6.3	83
6.4. The NFA for A in Example 6.4	90
6.5. The NFA for B in Example 6.4	91
6.6. The NFA for C in Example 6.4	91
6.7. The ALA(1) tree for (A,2) in Example 6.4	91
6.8. The NFA for A in Example 6.5	92
6.9. The NFA for B in Example 6.5	93
6.10. The NFA for C in Example 6.5	93
6.11. The ALA(1) tree for (A,2) in Example 6.5	93
6.12. The NFA for A in Example 6.6	94
6.13. The NFA for B in Example 6.6	95
6.14. The ALA(1) tree for (B,2) in Example 6.6	95
6.15. The NFA for A in Example 6.7	96
6.16. The NFA for B in Example 6.7	96
6.17. The NFA for C in Example 6.7	97
6.18. The ALA(1) tree for (A,2) in Example 6.7	97
6.19. The NFA for A in Example 6.8	98
6.20. The NFA for B in Example 6.8	98
6.21. The ALA(1) tree for (A,1) in Example 6.8	99
6.22. The NFA for A in Example 6.9	100
6.23. The NFA for B in Example 6.9	100
6.24. The ALA(1) tree for (A,3) in Example 6.9	100

Figure	Page
6.25. The NFA for E in Example 6.10.	102
6.26. The NFA for A in Example 6.10.	102
6.27. The NFA for B in Example 6.10.	102
6.28. The ALA(1) tree for (E,3) in Example 6.10. . . .	102
6.29. Thompson's Construction for $((\langle N_i \rangle e)s)$	105
6.30. Thompson's Construction for $(s(\langle /N_i \rangle e))$	107
6.31. Thompson's Construction for $((\langle N_i e)s(\langle /N_i \rangle e))$	108
7.1. The DCM tree for Example 7.4.	128
7.2. The DCM tree for Example 7.5.	129
7.3. The DCM tree for Example 7.6	131
7.4. The NFA for A in Example 7.6.	132
7.5. The NFA for B in Example 7.6.	132
7.6. The NFA for C in Example 7.6.	132
7.7. The NFA for X in Example 7.6.	132
7.8. The NFA for A_1 in Example 7.6	133
7.9. The NFA for A_2 in Example 7.6	133
7.10. The NFA for B_1 in Example 7.6	134
7.11. The ALA(1) tree for (A,1) in Example 7.6. . . .	135
7.12. The ALA(1) tree for $(A_1,1)$ in Example 7.6 . . .	135
7.13. The NFA for A in Example 7.7.	136
7.14. The NFA for B in Example 7.7.	136
7.15. The NFA for C in Example 7.7.	136
7.16. The NFA for X in Example 7.7.	137
7.17. The DCM tree for Example 7.7.	137

Figure	Page
7.18. The NFA for A_1 in Example 7.7	137
7.19. The NFA for B_1 in Example 7.7	138
7.20. The NFA for X_1 in Example 7.7	138
7.21. The ALA(1) tree for $(A_1, 2)$ in Example 7.7 . . .	138

1. *Introduction*

The Standard Generalized Markup Language (SGML) is a flexible framework for defining document structure. It was adopted as an international standard, ISO 8879, in 1986 and it is rapidly becoming accepted in industry and government [7,8,23]. The widespread acceptance of SGML can be attributed to two distinguishing features: 1. It is based on descriptive markup that separates the logical components of the document as opposed to procedural markup that defines the physical appearance (formatting) of the document. 2. It is a meta-language system for document definition rather than a specific markup scheme for document description. Almost any kind of document structure can be defined using SGML; productions called element declarations are used to define arbitrary elements of documents and to formally specify the context in which they can occur. A finite set of element declarations called a document type definition (DTD) defines the high level syntax of a class of documents. Using DTDs, SGML approaches the maximum amount of standardization that can be achieved in document design without sacrificing flexibility.

Although DTDs are elegant and powerful, they can be ambiguous. There are two kinds of ambiguity that are defined and prohibited by the standard [13]. In this dissertation, these definitions are revised and methods are shown for detecting the two kinds of ambiguity while parsing the DTD.

DTDs are similar to context free grammars (CFGs) [2,3] in that they contain a finite set of element declarations (productions), the left hand side of each element declaration is an element name (a nonterminal symbol), and one element is declared as the DOCTYPE element (the start symbol). The right hand sides of element declarations and the productions of CFGs are similar in that they are expressions containing terminal and nonterminal symbols. However, the right hand sides of element declarations are more complex than the productions of CFGs. The primary component of the right hand side of an element declaration is called a content model; each content model consists of a model group and optional exceptions. The model group is an extended regular expression that

defines the content of the element named on the left hand side in terms of text and other elements. Exceptions modify the effect of model groups by allowing elements to be included in or excluded from the element named on the left hand side. The other component of the right hand side is called the omitted tag minimization. Elements that occur in a document are bounded by a begin tag and an end tag; the omitted tag minimization allows for these tags to be omitted optionally.

Clause 11.2.4.3 of the standard defines and prohibits ambiguous content models. However, the scope of the clause is not clear; the standard shows examples of model groups that are ambiguous content models, but it does not state whether or not exceptions and omitted tag minimization affect the clause. Thus, there are questions in the SGML community regarding the definition and its implementation. Following an article on ambiguity [17], the editors comment: "We welcome your comments on this issue, specifically, a paper that mathematically proves or disproves whether it is possible to find all ambiguous content models..." Methods are shown in the literature for detecting ambiguous content models when omitted tags and exceptions are not considered [6,16,18,25].

Clause 7.3.1 of the standard prohibits ambiguity caused by omitted tags. Ambiguous documents are defined as opposed to ambiguous DTDs; this implies that detecting and preventing ambiguities caused by omitted tags must be accomplished at the time of document entry rather than during DTD design. There is no definition in the standard for the term *ambiguity* as it is used in Clause 7.3.1, and there are no adequate methods shown in the literature for defining, detecting, and preventing this kind of ambiguity. The significance of this problem is described in a recent article [9] as follows: "Probably the most misunderstood concept in SGML today is the use of the two characters, the hyphen and the 'o'...that indicate omitted tag minimization..."

Annex H of the standard states that model group notation reduces to regular expression notation and that "...conformance to a content model is essentially equivalent to

recognizing...regular expressions." However, it also states, "No assumptions should be made about the general applicability of automata theory to content models." The standard does not describe a formal language model for DTDs, and there is little work in the literature on this topic. One article [25] shows a method for converting DTDs into LL(1) context free grammars, but inconsistencies are noted in this conversion. The central hypothesis of this dissertation is that the difficulties encountered in defining and detecting ambiguity in DTDs arise from the lack of a suitable formal model for recognizing them.

In Chapter 2, background is provided for language theory and for SGML; the system of syntax productions that define DTDs is reviewed and examples are shown. Then, the relevant literature is reviewed. In Chapter 3, definitions are given for systems of regular expressions and for a class of language recognizers, systems of finite automata. Then an algorithm is given for constructing a system of finite automata, S , from a system of regular expressions, R , such that $L(S)=L(R)$. Useless states are defined for systems of finite automata and an algorithm is shown for removing the useless states. In Chapter 4, the syntax productions that define DTDs are converted to an equivalent system of regular expressions, and then a system of finite automata recognizing DTDs is constructed. Using this system of finite automata, a parser is constructed for DTDs; the parser is a deterministic implementation of the inherently nondeterministic system of finite automata. In Chapter 5, an implementation independent definition is given for ambiguous model groups, and a generalized algorithm is presented for detecting them. In Chapter 6, an algorithm is given for constructing a system of finite automata that recognizes the set of documents defined by a DTD when exceptions are not considered. A definition is given for DTDs that are ambiguous by omitted tags. Then, using the system of finite automata for DTDs an algorithm is given for detecting this kind of ambiguity. In Chapter 7 an algorithm is given for enumerating all possible ways that exceptions can affect model groups in a DTD. Then using this result, a second algorithm is given for constructing a system of finite automata that recognizes the documents defined by a DTD when

exceptions are considered. Examples are shown to illustrate that the methods in Chapter 6 for detecting ambiguity caused by omitted tags still apply when exceptions are considered. Chapter 8 states the conclusions drawn from this research and shows how the results can be applied by the SGML community. Recommendations for future work are also made.

The formal language model derived for SGML (systems of finite automata from systems of regular expressions) is applied in two ways: 1. A system of finite automata is constructed that recognizes DTDs, and then a parser for DTDs is constructed from this system of finite automata. 2. An algorithm is shown for constructing a system of finite automata that recognizes the set of documents defined by a DTD.

Revised definitions are given for the two kinds of ambiguity defined in the standard. The revised definitions formalize the existing definitions, they are consistent with them where feasible, and they distinguish the kinds of ambiguity that can occur in DTDs. Thus, they will resolve questions in the industry regarding ambiguity in DTDs. Algorithms are given for detecting these kinds ambiguity while parsing the DTD. The algorithm for detecting ambiguous model groups is generalized, and thus is an improvement over existing methods. The algorithm for detecting ambiguity caused by omitted tags is the first complete method shown for detecting this kind of ambiguity; it shows that this is a solvable problem, and it will be a useful tool for DTD design [9,11,12,17]. Thus, the language model, the revised definitions of ambiguity, and the methods for detecting ambiguity solve important open problems in SGML.

2. Literature Review

2.1. Review of Language Theory Concepts

In this section, some basic elements of language theory are briefly reviewed: regular sets, regular expressions, and finite state automata (NFAs). The notation and definitions from [2,3] are adopted with a few minor variations. Then a method is described for constructing an NFA from a regular expression [3].

Definition 2.1. Languages. An alphabet, Σ , is a finite set of symbols. A language over Σ is a set of strings consisting of symbols in Σ and possibly the empty string, ϵ . The concatenation of two languages L_1 and L_2 , written as L_1L_2 , is the set of all strings xy , such that x is in L_1 and y is in L_2 . The closure of a language L , L^* , is the empty string, ϵ , plus the set of all strings that can be derived by concatenating any number of strings in L .

Definition 2.2. Regular Expressions and Regular Sets. Regular expressions are used to denote a class of languages called the regular sets. A regular expression over Σ is defined as:

1. \emptyset is a regular expression denoting the regular set \emptyset .
2. ϵ is a regular expression denoting the regular set $\{\epsilon\}$.
3. a in Σ is a regular expression denoting the regular set $\{a\}$.
4. If p and q are regular expressions denoting the regular sets P and Q :
 - a. $(p|q)$ is a regular expression denoting $P \cup Q$.
 - b. (pq) is a regular expression denoting PQ .
 - c. (p^*) is a regular expression denoting P^* .

$L(s)$ is the language denoted by the regular expression, s .

Definition 2.3. Nondeterministic Finite Automata. A nondeterministic finite automaton (NFA) is a 5-tuple, $M = \{Q, \Sigma, \delta, q_0, F\}$, where:

1. Q is a finite set of states.
2. Σ is a finite set of input symbols (the input alphabet).

3. δ is the state transition function that maps $(Q \times (\Sigma \cup \epsilon))$ to subsets of Q .
4. q_0 is a state in Q : the start state of M .
5. F is a subset of Q : the final (or accepting) states of M .

When δ maps a state q and input symbol a to some subset of Q , $\{p, r\}$, the mapping is denoted by: $\delta(q, a) \rightarrow p$ and $\delta(q, a) \rightarrow r$. To illustrate concepts or proofs, NFAs are sometimes represented in graph form as shown in Figure 2.1. The arrow into q indicates that it is the start state, and the double circle around r indicates that it is a final state.

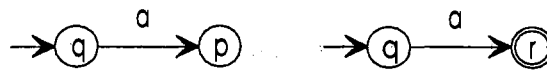


Figure 2.1. Graph representations of NFAs.

If M is an NFA, then a configuration of M is (q, w) , where q is the current state of M and w is the remaining input, a string of symbols in Σ^* . An initial configuration is of the form (q_0, w) and a final configuration is of the form (q, ϵ) , where q is in F . A move by M , denoted by \vdash , is a binary relation (on the set of all possible configurations) such that if $\delta(q, a) \rightarrow p$, then $(q, aw) \vdash (p, w)$. The transitive closure of \vdash is denoted by \vdash^+ , the reflexive-transitive closure by \vdash^* , and \vdash^n denotes n moves. An input string w in Σ^* is recognized (accepted) by M if $(q_0, w) \vdash^* (q, \epsilon)$ for some q in F . $L(M)$ denotes the language recognized by M .

Thompson's construction derives an NFA from a regular expression, r , such that the NFA recognizes $L(r)$ [2]. Although there are other methods in the literature for constructing NFAs from regular expressions [4,5], this method is used because it has properties that are important to some of the algorithms in this dissertation. The regular expression r is first parsed into its constituent subexpressions. Then in a left to right, innermost first order, NFAs are inductively constructed for the subexpressions of r by combining the NFAs of their respective subexpressions. There is one basis rule and four inductive rules for the construction; these are described below. Each component NFA

derived during the course of the construction has important properties: there is exactly one start state, one final state, and no transitions enter the start state or leave the final state. In the illustrations of these constructions in Figures 2.2-2.5, for a subexpression s of r , the NFA $M(s)$ is denoted as an arc from the single start state q of $M(s)$ to the single final state f of $M(s)$, and the label on the arc is $M(s)$.

1. The basis rule constructs a separate NFA each time that a symbol b in Σ occurs in r .

Construct the NFA for b as shown in Figure 2.2.

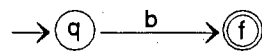


Figure 2.2. The NFA for b in Σ .

2. For the regular expression, slt , construct the composite NFA for $M(slt)$ from the component NFAs $M(s)$ and $M(t)$ as shown in Figure 2.3. In Figure 2.3, states q' and q'' are the start states of $M(s)$ and $M(t)$ and states f' and f'' are the final states of $M(s)$ and $M(t)$. These states are not start and final states of $M(slt)$; the start state of $M(slt)$ is q and the final state is f .

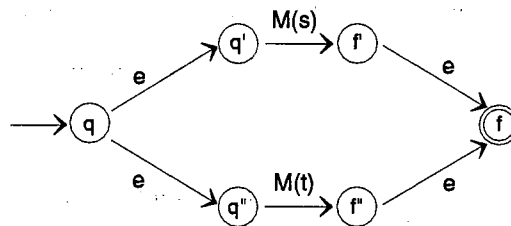


Figure 2.3. The NFA for (slt) .

3. For the regular expression, st , construct the composite NFA for $M(st)$ from the component NFAs $M(s)$ and $M(t)$ as shown in Figure 2.4. The start state q of $M(s)$ becomes the start state of $M(st)$, and the final state f of $M(t)$ becomes the final state of $M(st)$. The start state q'' of $M(t)$ is merged into the final state f' of $M(s)$ as follows:

remove q'' from $M(st)$ and for all transitions in $M(t)$, $\delta(q'', b) \rightarrow p$, for some p in $M(t)$ and b in Σ , add $\delta(f'', b) \rightarrow p$ to $M(st)$.

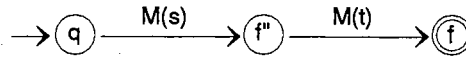


Figure 2.4. The NFA for (st) .

4. For the regular expression, s^* , construct the composite NFA for $M(s^*)$ from the component NFA for $M(s)$ as shown in Figure 2.5. The start state q'' of $M(s)$ is no longer a start state in $M(s^*)$ and the final state f'' of $M(s)$, is no longer a final state in $M(s^*)$.

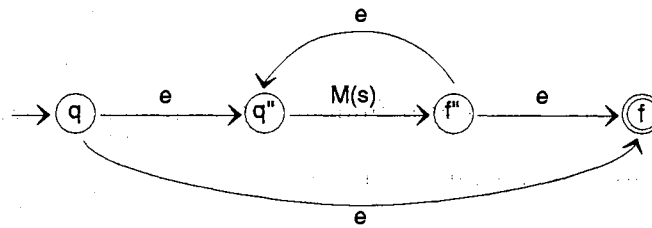


Figure 2.5. The NFA for (s^*) .

5. For (s) , the NFA is the same as the NFA $M(s)$ for s .

2.2. SGML Background

SGML is a meta-language system for defining the structure of documents; there are separate features for defining the high level syntax (parsing) and the low level syntax (token recognition). This paper does not consider token recognition; it is assumed that an SGML parser can distinguish between text and markup. Markup in SGML is the codes that are added to the text of a document to separate the logical components (elements) [13].

A document type declaration, also called a document type definition (DTD) [8,9,11,12,16,17,19,24,25], defines the high level syntax for a class of documents, the

document type. A DTD contains a set of element declarations that define the elements of a document type and the context in which they may occur. One element is declared as the top level or "DOCTYPE" element. There are other components of DTDs that do not affect the high level syntax of document specifications. Only the DOCTYPE declaration and the element declarations are considered in this paper.

A document that conforms to the requirements of the DTD is called a *document instance*. A document instance is a single occurrence of the DOCTYPE element and everything that occurs within it: a stream of tokens consisting of markup (element begin and end tags) and text. An element begin tag is written as "<element_name>" and an element end tag is written as "</element_name>." Example 2.1 shows a DTD and a document instance.

Example 2.1. A DTD and a conforming document instance. Consider the DTD

```
<! DOCTYPE book [
  <! ELEMENT book (header, (header,chapter)+)>
  <! ELEMENT chapter (#PCDATA)>
  <! ELEMENT header (#PCDATA)> ]>
```

The first line defines the document type as a book. The next three lines are element declarations for the elements book, chapter, and header. The parenthesized expressions are model groups, the only required components of content models. These declare that 1. a book contains a header, followed by one or more chapters, each preceded by a header. 2. a chapter contains only text. 3. a header contains only text. A conforming document instance for D is

```
<book>
  <header>This is a header for the book
</header>
  <header>This is a header for the first chapter
</header>
  <chapter>This is the text of a chapter
</chapter>
</book>
```

2.2.1. SGML Syntax Productions

The syntax of DTDs is defined by a hierarchical system of syntax productions. Each component of DTDs is defined by a production of the form "syntactic variable = expression", where the syntactic variable names the component and the expression defines the component in terms of tokens and operators. The tokens are: 1. syntactic variables (each of which has a corresponding syntax production), 2. keywords, and 3. other tokens. The operators determine the ordering and selection of the tokens and are defined in Table 2.1.

Table 2.1. The operators of syntax productions.

Operator Class	Operator Symbol	Operation
Occurrence- Indicators	?	Optional (0 or 1 time)
	+	Required and repeatable (1 or more times)
	*	Optional and repeatable (0 or more times)
Connectors	&	All must occur in any order
"	,	All must occur in the order shown
"		One and only one must occur
Parenthesis	()	Precedence of operations

Although they are defined with a different terminology, the operators '*', '|', ',', and parenthesis correspond to the regular expression operators, '*', '|', concatenation, and parenthesis respectively. Expressions containing '?', '+', and '&' also have equivalent regular expressions [13]. The following conventions are used to represent syntax productions: syntactic variables are in lower case, keywords are in quotes, all other tokens are in upper case, and the operators are unquoted characters.

There are over 75 syntax productions in the system of syntax productions defining DTDs [13]. To eliminate unnecessary detail, the syntax productions shown below are derived from the syntax productions in the standard as follows:

1. White space, some delimiters, and other items that do not affect the high level syntax of document instances defined by DTDs are omitted.
2. Some occurrences of syntactic variables in the expressions of syntax productions are replaced by their corresponding expressions. This replacement is similar to removing single productions in context free grammars and does not affect the syntax defined for DTDs.
3. This paper does not consider the optional SGML features: RANK, DATATAG, and SHORTTAG. All tokens related to these features are removed from the syntax productions.

The result of these transformations is a simplified system of syntax productions for DTDs that is relevant to the scope of this research and for the purposes of this dissertation is equivalent to the system of syntax productions defined in the standard. This simplified system of syntax productions is defined in the following paragraphs.

Document type declarations, also called *document type definitions* or *DTDs*, define the high level syntax of the document instances. The syntax production for DTDs is:

DTD = "DOCTYPE", GI, element declaration*

A DTD consists of the keyword, "DOCTYPE", followed by a GI (element name), followed by one or more element declarations. The GI names the DOCTYPE (or top level) element of the document. Although it is not shown in the syntax productions, the standard requires that there must be at least one element declaration for the DOCTYPE element. The following syntax production implements this syntactic requirement:

DTD = "DOCTYPE", GI, element declaration+

Element declarations specify the high level syntax of document types by naming the structural elements and defining the content of these elements. The syntax production for element declarations is:

```
element declaration = MDO, "ELEMENT", element type, (omitted tag
                    minimization)?, (declared content | content model), MDC
```

MDO and MDC are delimiters for the declaration. Thus, an element declaration consists of a keyword followed by the element type, an optional omitted tag minimization, and either declared content or a content model. The syntax productions for element type, content models, declared content, and omitted tag minimization are described below.

The *element type* is a list of elements that are defined by the element declaration. The syntax production for element type is:

```
element type = GI | name group
```

The element type is either a GI (element name) or a parenthesized list of (GIs). The content of elements named by the GIs is defined by the remainder of the element declaration.

Content models define the content of the elements named in the element type. The syntax production for content models is:

```
content model = model group, exceptions?
```

Model groups define the content of elements in terms of text and other elements.

Exceptions are optional; they modify the effect of the model group by allowing elements to be included in or excluded from the elements named in the element type and from any elements that occur within them in a document instance.

Model groups are expressions that are similar to the expressions of syntax productions. Each model group defines the content of elements named by the element type. The system of syntax productions for model groups is:

```
model group = GRPO, content token, ((AND | OR | SEQ), content
token)*, GRPC, (OPT | PLUS | REP)?
```



```
content token = "#PCDATA" | (GI, (OPT | PLUS | REP)? ) | model
group
```

The tokens OPT, PLUS, REP, AND, SEQ, OR, GRPO, and GRPC represent the operators for model groups. They are defined the same as the syntax production operators shown in Table 2.1: '?', '+', '*', '&', ',', '|', '(', and ')' respectively. In Example 2.2, which illustrates a model group, these default characters are used for the operators.

"#PCDATA" represents an occurrence of a character string (zero or more text characters) in a document instance and GI's, the names of elements, represent occurrences of the named elements. The element names and "#PCDATA" that occur in a model group, *m*, are the input symbols of the language *L(m)*. Each distinct occurrence of "#PCDATA" or an element name in a model group is a *primitive content token*. This term is used later in the SGML definition of ambiguous content models.

Example 2.2. Consider the model group

```
(header, (header, chapter)+)
```

This group specifies a header followed by one or more chapters, each preceded by a header. Header and chapter are GIs (element names), left and right parenthesis are GRPO and GRPC respectively, and '?', ',', and '+' are the default characters for the operator tokens, OPT, SEQ, and PLUS respectively. There are three primitive content tokens: one for each occurrence of an element name.

Although it is not shown in the syntax production for model group, the standard requires the following: "Only one kind of connector can occur in a single model group (but a model group nested within it could have a different connector)." The following syntax production for model group implements this syntactic requirement:

```
model group = GRPO, content token, ( (AND, content token)* | (OR,
content token)* | (SEQ, content token)* ) GRPC, (OPT | PLUS |
REP)?
```

Exceptions modify the effect of model groups to which they apply. The syntax productions for exceptions are:

```

exceptions = (exclusions, (inclusions)? ) | inclusions

inclusions = PLUS, GRPO, GI, GI*, GRPC

exclusions = MINUS, GRPO, GI, GI*, GRPC

```

Thus, inclusions and exclusions are parenthesized lists of one or more element names; inclusions are preceded by a PLUS and exclusions are preceded by a MINUS. Exceptions apply anywhere in the instance of the elements named in the element type, including within subelements of these elements. Exclusions override inclusions as illustrated in Example 2.3 below. Note that exceptions are optional within each element declaration.

Example 2.3. Consider the DTD

```

<!DOCTYPE TOP [
  <!ELEMENT TOP    (A, C?)  >
  <!ELEMENT  A     (B, C)  -(X) +(Y) >
  <!ELEMENT  B     (C, X?) >
  <!ELEMENT  C     (#PCDATA) +(X)
  <!ELEMENT  X     (#PCDATA) >
  <!ELEMENT  Y     (#PCDATA) >      ]>

```

In this DTD the included element, Y, may appear anywhere in an instance of an A, which includes anywhere in a B or a C in an A. The element X is excluded anywhere in an instance of A. Thus, even though it is an optional element in B, it cannot occur anywhere within a B. Exclusions have priority over inclusions. The exclusion of X overrides the inclusion of an X within a C, when the C is within an A. However, the standard is not clear about the scope of exceptions: "The exceptions apply anywhere in an instance of the element, including subelements..." The subelements of an element are defined to be only the elements contained immediately within the element, and not those at lower levels of nesting [13]. Thus, either the phrase "including subelements" is redundant or the phrase "anywhere in the instance of an element" is not precise. For example, it is not clear whether an X is allowed in a C within a B within an A. Exceptions are discussed in more detail in Chapter 7.

Declared content is the other alternative for defining the content of the elements named by the element type. The syntax production for declared content is:

```
declared content = "CDATA" | "RCDATA" | "EMPTY"
```

"EMPTY" means that the element has no content. "CDATA" and "RCDATA" are similar to "PCDATA" in model groups. However, the only markup recognized within "CDATA" or "RCDATA" content is markup that would end the element. It is assumed that a token recognizer can distinguish between markup and text. Thus, "CDATA", "RCDATA", and "PCDATA" each represent any string of zero or more text characters.

Omitted tag minimization allows for start and end tags of elements to be omitted in the document instance. The syntax productions for omitted tag minimization are:

```
omitted tag minimization = start-tag minimization, end-tag  
minimization
```

```
start-tag minimization = "O" | MINUS
```

```
end-tag minimization = "O" | MINUS
```

"O" means that the tag may be omitted from an occurrence of the element in the document instance, and MINUS ('-') means that the tag may not be omitted. All elements with declared content of "EMPTY" must not have an "O" specified for end tag omission.

Example 2.4. Omitted tag minimization.

a. Start tag omission. Consider the DTD

```
<!DOCTYPE TOP [
<!ELEMENT TOP - - (A) >
<!ELEMENT A O - (B) >
<!ELEMENT B - - (#PCDATA) ]>
```

and a document instance

```
<TOP>
<B>Text of a B element
</B>
</A>
</TOP>
```

The begin tag for A can be omitted because the begin tag minimization is set to "O".

b. End tag omission. Consider the DTD

```
<!DOCTYPE TOP [
<!ELEMENT TOP - - (A, B?) >
<!ELEMENT A - O (B?) >
<!ELEMENT B - O EMPTY > ]>
```

and a document instance

```
<TOP>
<A>
<B>
</TOP>
```

The end tag for A can be omitted because the end tag minimization for A is set to "O".

B is declared to have EMPTY content, and thus must have the end tag minimization set to "O"; a B element cannot have any content or an end tag (it consists only of a start tag).

There are additional restrictions placed on the omission of start and end tags. These are discussed in Section 2.2.3.

2.2.2. Ambiguous Content Models (Clause 11.2.4.3)

Ambiguous content models are defined (and prohibited) in Clause 11.2.4.3 of the standard as follows:

"A content model cannot be ambiguous; that is, an element or character string that occurs in the document instance must be able to satisfy only one primitive content token without looking ahead in the document instance. The priority rules stated earlier in 11.2.4 are not considered in determining whether a content model is ambiguous.

NOTE--For example, the content model in

```
<! element e ((a, b?), b)>
```

is ambiguous because after an 'a' element occurs, a 'b' element could satisfy either of the remaining tokens. The ambiguity can be avoided by using intermediate elements, as in:

```
<!element e (f, b) >
```

```
<!element f (a, b?) >
```

Here the token satisfied by 'b' is determined unambiguously by whether the 'f' element ends before the 'b' occurs..."

Example 2.5. An ambiguous content model. Consider a variation of Example 2.1 as follows. Let D be the DTD:

```
<! DOCTYPE book [
  <! ELEMENT book (header?, (header,chapter)+)>
  <! ELEMENT chapter (#PCDATA)>
  <! ELEMENT header (#PCDATA)> ]>
```

The book level header element is now optional (?). Consider the following document instance for D:

```
<book>
  <header>This is a header.
</header>
  <header>This is a header.
</header>
  <chapter>This is the text of a chapter
</chapter>
</book>
```

The model group for book is an ambiguous content model. There are two primitive content tokens for header. In this document instance, without looking ahead it cannot be determined whether the first header element is the optional book level header or the first required chapter level header.

Clause 11.2.4.3 shows by example that ambiguous model groups are ambiguous content models, but it does not clearly state whether or not exceptions and omitted tags affect ambiguous content models. The following examples show that they can.

Example 2.6. Exceptions and ambiguous content models. Consider the following element declaration.

```
<!ELEMENT A - - - (X | A | X) -(X) >
```

If exceptions are not considered, the content model "(X | A | X) -(X)" is ambiguous, because any occurrence of an X element within an A element could satisfy either primitive content token for X in the model group. However, if exceptions are considered, an X element can no longer occur in an A element in any valid document instance, and thus there can be no X in a document instance that satisfies either of the primitive content tokens for X in the model group. Therefore, when exceptions are considered, this content model is no longer ambiguous.

Example 2.7. Omitted tags and ambiguous content models. Consider the DTD

```
<!DOCTYPE TOP [
<!ELEMENT TOP - - - (A, B?) >
<!ELEMENT A - O (C, B?) >
<!ELEMENT B - - (#PCDATA) >
<!ELEMENT C - - (#PCDATA) > ]
```

and the document instance

```
<TOP>
<A>
<C>data characters</C>
<B>data characters</B>
</TOP>
```

When omitted tags are not considered, none of the individual model groups are ambiguous content models. However, because the primitive content token for B in the model group for A is optional (?), the B element that occurs in the example document instance can satisfy either primitive content token for B in the DTD. Clause 11.2.4.3 does not specify that the primitive content tokens must be in the same model group for the conditions of ambiguity to be met.

2.2.3. Ambiguity Caused by Omitted Tags (Clause 7.3.1)

Clause 7.3.1 of the standard places additional restrictions on tag omission as follows:

"A tag can be omitted only as provided in this sub-sub-clause, and only if the omission would not create an ambiguity..."

Clause 7.3.1.1 lists specific rules for start tag omission: "The start-tag can be omitted if the element is a contextually required element and any other elements that could occur are contextually optional elements, except if: a) the element type has a required attribute or declared content; or b) the content of the instance of the element is empty..."

Clause 7.3.1.2 lists specific requirements for end tag omission: "The end-tag can be omitted for an element that is followed either a) by the end of the SGML document entity or the SGML subdocument entity; b) by the end-tag of another open element; or c) by an element or SGML character that is not allowed in its content. NOTE -- An element that is not allowed because it is an exclusion has the same affect as one that is not allowed because no token appears for it in the model group."

Thus, even if tag omission is permitted by an "O" in the omitted tag minimization of the element declaration, a tag cannot be omitted unless it conforms to the restrictions of Clause 7.3.1. For example, in Example 2.4.a if the model group (A) for TOP was replaced by (A?), then the begin tag of A could not be omitted in the document instance because A would not be a required element; the '?' operator makes it optional.

The term *ambiguity* in Clause 7.3.1 is not defined in the standard, and thus it cannot be shown whether or not the rules in 7.3.1.1 and 7.3.1.2 are complete. It is not connected by the standard to ambiguous content models that are prohibited by Clause 11.2.4.3. If it were defined similarly, then Example 2.8. shows that the rules for start tag omission in Clause 7.3.1.1 are incomplete.

Example 2.8. Ambiguity caused by omitting start tags (adapted from [25]). Consider the DTD:

```

<!DOCTYPE A [
  <!ELEMENT A - - (B?, C) >
  <!ELEMENT B - - (#PCDATA) >
  <!ELEMENT C O - (D) >
  <!ELEMENT D O - (B) > ]

```

and the document instance

```

<A>
  <B>text of a B.
</B>
</D>
</C>
</A>

```

By the rules in Clause 7.3.1.1, the start tags for C and D can be omitted. However, when encountering the B element, without looking ahead in the document instance it cannot be determined which primitive content token for B in the DTD is satisfied.

Example 2.4.b illustrates end tag omission. By the rules in Clause 7.3.1.2, the document instance is allowed. Even by looking ahead to the end of the document instance, it cannot be determined which primitive content token for B in the DTD is satisfied. However, by using the rules of Clause 7.3.1.2 as disambiguating rules, SGML parsers can resolve this ambiguity while parsing the document instance. This is discussed in more detail in the following subsection.

2.2.4. Disambiguating Rules for SGML Parsers

The priority rules in Clause 11.2.4, which are referred to in Clause 11.2.4.3, state priorities for SGML document instance parsers as follows: "The elements and data characters of the content must conform to the content model by satisfying model group tokens and exceptions in the following order of priority: a) a repetition of the most recent satisfied token, if it has a REP or PLUS occurrence indicator; or b) some other token in a model group, possibly as modified by exclusion exceptions...; or c) a token in an inclusion exceptions group..."

The rules in Clause 7.3.1.2 for omitting end tags also imply priority rules for document instance parsers. For instance, the ambiguity described in the preceding section for Example 2.4.b could be resolved by the rules in 7.3.1.2 as follows: the B element in the document instance must be within the A element or the document instance would violate the rules in Clause 7.3.1.2 for omitting the end tag of A. However, resolution of ambiguity by this method would place the responsibility for correctness on data entry rather than on DTD design. The rules in 7.3.1.1 for omitting start tags do not resolve the ambiguity described in Example 2.8.

There are other clauses that could also imply disambiguating rules, including Clauses 7.4, 7.5, and 7.6. However, there is no precise statement or summary in the standard regarding disambiguating rules for SGML parsers.

2.3. Related work in Language Theory

Woods describes the use of recursive transition networks for the analysis of natural language sentences [26]. He defines a recursive transition network as a directed graph with labeled states (nodes) and arcs that is similar to a finite set of NFAs; one state is the start state and there is a nonempty set of final states. The arcs are labeled with either input symbols or with the names of states that are the start states of the individual NFAs. For each transition on a state name, the state at the end of the arc is placed on a pushdown list of states and control is transferred to the state named on the arc. The set of final states are the final states of the individual NFAs. When a final state is entered, control is transferred to the state on the top of the pushdown list and this state is popped from the pushdown list. Accepting configurations occur when the input is exhausted, the stack is empty, and the current state is a final state.

Woods observes that recursive transition networks describe the context free languages, and that they are inherently nondeterministic because the pushes and pops are e-moves; however, they may be optimized by removing all nondeterminism except for the e-moves

[26]. He also observes that many existing top down and bottom up parsing algorithms for context free grammars apply directly to recursive transition networks, and for those that do not, recursive transition networks have an analogous parsing algorithm. He argues that recursive transition networks are superior to pushdown automata and context free grammars in several respects; they allow for more efficient expression, more efficient parsing algorithms, and they are more easily extensible to context dependent models.

LaLonde defines an alternative to context free grammars called regular right part (RRP) grammars [15]. An RRP grammar consists of a set of terminal symbols, a set of nonterminal symbols, a goal (or start) symbol, and a finite set of productions. The left side of the production is a single nonterminal symbol, and the right side is a multiple entry NFA (it may have more than one start state). There may be more than one production with the same left hand side. A variant form of an RRP is that the right hand sides may be regular expressions. LaLonde states that the two forms are equivalent because regular expressions have equivalent NFAs. However, he also states that this does not necessarily imply that each form can be mechanically constructed from the other. A language is defined by an RRP as follows. A goal sentential form is any string that can be derived from the goal symbol (or another goal sentential form) by replacing a nonterminal symbol with an element of the language defined by its right hand side. The set of words defined by an RRP is the set of goal sentential forms that consist of only input symbols.

LaLonde gives an informal argument that RRP's are equivalent to the context free grammars [15]. He then defines a subset of RRP grammars, the RRP LR(k) grammars. He observes that it is equivalent to the definition of LR(k) grammars given in [3], and he presents a generalization of the standard LR(k) parser for context free grammars. He also argues that RRP's are more natural and easy to understand than context free grammars, and he demonstrates this by showing that context free grammars require recursive productions to define syntax that is not inherently recursive; the same syntax can be

described by RRP's without using recursion. Thus, RRP's can be reduced to contain only recursion that is inherent in the language defined.

Aho, Sethi, and Ullman describe a method for constructing predictive parsers by deriving state transition diagrams for the productions of a context free grammar [2]. They show how to combine diagrams by substituting the arcs for nonterminal symbols with their respective diagrams. The resulting system of state transition diagrams is similar to the recursive transition networks described by Woods [26].

2.4. Related Work on Ambiguity in SGML

Ambiguity is discussed in two places in the standard: Clause 11.2.4.3 defines and prohibits ambiguous content models and Clause 7.3.1 gives rules for tag omission that are designed to prevent ambiguity. These two clauses are not precise and they give no methods for detecting ambiguity. This leaves some open questions: 1. What kinds of ambiguity are defined in SGML? 2. Are the kinds of ambiguity discussed in the two clauses related? 3. Can these kinds of ambiguity be detected? In this section the related work is reviewed to determine if these questions have been answered. The related work for ambiguous content models (Clause 11.2.4.3) is reviewed first. All of the literature reviewed considers only model groups, which are the only required components of content models; the effects of exceptions and omitted tags on Clause 11.2.4.3 are not considered. With the exception of [16], the papers reviewed assume that these other components of content models do not affect Clause 11.2.4.3.

Warner and van Egmond present a method for detecting ambiguous content models [25]. They rewrite model groups as rules for an existing LL(1) parser generator and conclude that: "The difference between the LL(1) property and the unambiguity requirement for SGML is that there is one construct which is ambiguous for LL(1), but unambiguous for SGML." They observe that for the LL(1) property to hold, any nonterminal (primitive content token) that may be empty cannot have a '+' or '*'

occurrence indicator. Model groups containing this construct are not necessarily ambiguous in SGML. There is also another construct that is not ambiguous in SGML and is not LL(1). Any grammar that is left recursive cannot be LL(1) [2]. Thus, any production of the form $A \rightarrow A\alpha$ will not be LL(1), and consequently, element declarations of the form `<!ELEMENT a (a,...)>` cannot be LL(1). Although it is not a common construct, recursion is allowed and it is used in some DTDs [22]. Thus, there are two constructs for which LL(1) methods will detect ambiguity where SGML does not.

Klein shows that for a regular expression, E , the Glushkov automaton (NFA) of E , whose states correspond to the input symbols in E , can be constructed in time quadratic to the size of E [5] and that this improves on the cubic time demonstrated for alternative constructions in [2,4]. She restates the definition of unambiguous regular expressions from [4] as follows: "A regular expression E is unambiguous if, for each word w , there is at most one path through E that matches w ..." She then gives an algorithm for deciding unambiguity of a regular expression in time quadratic to the size of the expression. This method does not consider the model group operators, AND and OPT.

Klein shows a method for deciding if a content model is unambiguous in time linear to the size of the model group [6]. Her approach is based on marking expressions (model groups); each symbol (primitive content token) in an expression, E , is assigned a unique position (index) that distinguishes it from other symbols. She restates the definition of ambiguity in SGML in terms of marked expressions: an expression is unambiguous if and only if given a marking E' of E , for any two words uxv and uyw in $L(E')$ where x and y are marked symbols, $\text{symbol}(x)=\text{symbol}(y)$ implies $x=y$. She then gives a decision algorithm for this definition of unambiguity. For an expression E , with a marking E' and a marked symbol, x , she gives functions $\text{first}(E)$, $\text{last}(E)$, and $\text{follow}(E,x)$. These functions are derived from similar functions used to construct a deterministic finite automata (DFA) from a regular expression [2]; the DFA is not actually constructed. The expression E is unambiguous if and only if no two positions in E compete; two positions x and y compete

if and only if x and y are both in $\text{first}(E)$ or are both in $\text{follow}(E,z)$ for some position z in E . The algorithm does not consider #PCDATA tokens.

L. Price outlines a method for detecting ambiguous content models [18]. She shows NFAs for model groups in which the labels of the arcs are subscripted symbols representing primitive content tokens. She shows by example that a content model is ambiguous "if two or more arcs with the same label but with different subscripts emanate from the same node.", and she observes that "Detecting ambiguity in content models is similar to detecting nondeterminism in the FSAs that represent them." Note that this use of the term "nondeterminism in the FSAs" is equivalent to NFAs that are not deterministic [3]. She also shows a significant space saving representation for NFAs of "AND" groups.

Matzen, George, and Hedrick present a complete algorithm for detecting ambiguous model groups [16]. When the optional components of element declarations are not considered, the algorithm detects ambiguous content models as defined in Clause 11.2.4.3. The algorithm supports the methods outlined by L. Price in [18]. A parser is constructed for model groups; indexed NFAs are defined and translation actions are added to the parser to construct an indexed NFA recognizing $L(m)$ for a model group, m , where the arcs of the indexed NFA are indexed symbols denoting the primitive content tokens of m . After removing ϵ -transitions from the indexed NFA, the method for detecting ambiguity is the same as the method outlined in [18] as described above. This method of detecting ambiguity in expressions is different than the methods described in [4,5], where the states of the NFA contain the information necessary to detect ambiguity.

Clause 7.3.1 gives rules for tag omission that are designed to prevent ambiguities. However, the standard does not define the term *ambiguity* for this clause, and it does not state that it is related to ambiguous content models. Other work also does not relate these two types of ambiguity to each other [6,9,11,17,18,24,25]. In the four papers that show methods for detecting ambiguous content models [6,16,18,25], only one [25] shows

methods for detecting ambiguity prohibited by Clause 7.3.1. This paper and other work related to omitted tag minimization are discussed in the following paragraphs.

Because the term *ambiguity* is not defined for Clause 7.3.1, the specific rules in Clauses 7.3.1.1 and 7.3.1.2 for tag omission cannot be shown to be equivalent to the general rule prohibiting ambiguity in Clause 7.3.1. Warner and van Egmond give a restatement of the rules of Clause 7.3.1.1 for start tag omission, and they show by example that these are unnecessarily restrictive [25]. They also give an example (similar to Example 2.8) that shows for a reasonable interpretation of ambiguity, the rules for start tag omission in Clause 7.3.1.1 are not sufficient to prevent ambiguity. They do not formally define this kind of ambiguity.

Although the standard requires ambiguity caused by omitted tags to be detected, it does not require static detection while parsing the DTD. It also does not give any method for detection, either while parsing the DTD or while parsing a document instance. Warner and van Egmond implement static detection of ambiguities caused by start tag omission using a restatement of the rules of Clause 7.3.1.1 [25]. They state that "...end tag omission cannot be corrected during parsing of the DTD..."; ambiguities caused by omitted end tags are resolved while parsing the document instances using disambiguating rules derived from Clause 7.3.1.2 for end tag omission.

Because there is no definition in the standard for ambiguity caused by omitted tags, there are open questions in the SGML community regarding the implementation. The problems with omitted tags are first summarized, and then the literature is reviewed to illustrate how these problems are affecting the SGML community. 1. There is a significant burden on DTD designers to use omitted tag minimization in a manner that does not introduce ambiguity. 2. If ambiguity is not prevented in the DTD then the disambiguating rules for SGML parsers can introduce error. 3. This places the responsibility for preventing errors on the author or data entry personnel, who are least equipped to handle it. 4. There is not a consensus in the SGML community on the disambiguating rules for

document instance parsers; this compounds the problems described in 1-3 above. 5.

There have been no complete methods shown for detecting ambiguity caused by omitted tags.

Graf gives a number of examples of ambiguity introduced by omitted tags and illustrates how some SGML parsers may interpret (disambiguate) a document instance differently than others [11]. He also argues that SGML parsers can resolve ambiguities differently than the author intended and warns authors accordingly: "...you might find an important part of your data missing when you went to retrieve it. These are the unexpected results." He concludes the following: "The only real cure for this kind of ambiguity, is to eliminate or at least severely reduce the use of minimization."

McFadden and Wilmott [17] respond to Graf's article [11] and state that there are no ambiguities in his examples. They summarize their view as follows: "All conforming SGML parsers will perform the same way. SGML parsers are deterministic and never 'assume' anything. The examples may appear ambiguous to human readers who do not know the rules. SGML parsers are programmed to know the rules, and DTD designers are expected to know the rules." However, no references to the standard are cited for these rules. Although they observe that effective use of minimization in DTD design requires significant expertise, they conclude that there is no need to eliminate or restrict the use of the omitted tag minimization feature.

Heath and Welsh recommend that start and end tag minimization be removed from SGML [12] because "it places a high burden on both humans and programs that process SGML documents." Davis describes difficulties with the omittag feature, and he observes that parsers using the disambiguating rules of the standard can produce results that are different than the author intended (no references are cited for the disambiguating rules) [9]. He illustrates the current status of the problem when he begins this recent article as follows: "Probably the most misunderstood concept in SGML today is the use of the two characters, the hyphen and the 'o'...that indicate 'omitted tag minimization'..."

Waldt shows examples that illustrate how exceptions can affect ambiguity in SGML [24]. In the literature there are no methods shown for detecting the effect of exceptions on ambiguity. The difficulty of this problem is illustrated by the way that exceptions are implemented in document instance parsers. L. Price outlines a document instance parser based on stacks of NFAs [18]. Exceptions are not implemented in the static model of the NFAs; they are implemented dynamically while parsing the instance. Warner and van Egmond implement a document instance parser that is constructed by an LL(1) parser generator [25]. Exceptions are not implemented as part of the LL(1) grammar; they are implemented dynamically using a stack of currently applicable exceptions.

The literature review in this chapter shows that there are open problems related to detecting ambiguity in SGML: 1. Although ambiguous model groups can be detected, there are still open questions regarding the scope and interpretation of Clause 11.2.4.3. 2. There is no precise definition for ambiguity caused by omitted tags (Clause 7.3.1), and no complete methods have been shown for detecting this kind of ambiguity. 3. No method has been shown for determining how exceptions affect ambiguity in SGML.

In the next chapter, a formal model is shown for recognizing DTDs and document instances, and in Chapter 4 a parser is constructed for DTDs and model groups. In Chapter 5 a definition is given for ambiguous model groups, and the parser is used to show a generalized algorithm for detecting them. In Chapter 6, ambiguity caused by omitted tags is defined, and an algorithm is given for detecting this kind of ambiguity when exceptions are not considered. In Chapter 7 a language model is shown for document instances when exceptions are considered, and examples are shown to illustrate that the algorithm in Chapter 6 for detecting ambiguity caused by omitted tags still applies.

3. A Language Model for SGML DTDs

Input to an SGML parser is a DTD and one or more document instances. The parser must parse the DTD and then construct a parser for document instances. The system of syntax productions shown in Chapter 2.2.1 defines the syntax of DTDs, and DTDs define the high level syntax of document instances. In this chapter a model is developed that can be used to construct a recognizer for DTDs and recognizers for the document instances defined by DTDs.

In the first section, systems of regular expressions and a corresponding class of recognizers, systems of finite automata, are defined. In the second section, an algorithm is presented for constructing a system of finite automata S from a system of regular expressions R , such that $L(S)=L(R)$. Useless NFAs, inaccessible NFAs, and useless states in systems of finite automata are also defined, and algorithms are also given for removing these from a system of finite automata..

3.1 Preliminary Definitions

Definition 3.1. Systems of regular expressions. A system of regular expressions, R , is defined to be a 4-tuple, $R=(\Sigma, N, N_0, P)$, where:

1. Σ is a set of terminal symbols.
2. N is a set of nonterminal symbols.
3. N_0 is a distinguished symbol in N , the start symbol.
4. P is a set of productions, $A_i \rightarrow \alpha_i$, $i=1, \dots, n$, $n \geq 1$. There is exactly one production for each nonterminal in N ; A_i , $i=1, \dots, n$, are the elements of N and each α_i is a regular expression over $(\Sigma \cup N)$. $N_0=A_i$ for some $i=1, \dots, n$.

A system of regular expressions defines a language over Σ in a recursive manner using strings called regular expression forms that are regular expressions over $(\Sigma \cup N)$. Regular expression forms are defined using a terminology similar to that used for sentential forms in context free grammars in [3] as follows:

1. N_0 is a regular expression form of R .
2. If $\gamma A \beta$ is a regular expression form of R , and A is in N where $A \rightarrow \alpha$ is in P , then $\gamma \alpha \beta$ is a regular expression form of R .

A word of R is any string x such that x is in $(L(\alpha) \cap \Sigma^*)$ for some regular expression form α of R . The language generated by R is the set of words of R and is denoted as $L(R)$.

A relation, \Rightarrow (directly derives), is defined on the regular expression forms of R as follows: if $\gamma A \beta$ is a regular expression form of R and A is in N where $A \rightarrow \alpha$ is in P , then $\gamma A \beta \Rightarrow \gamma \alpha \beta$. \Rightarrow^+ denotes the transitive closure of R and \Rightarrow^* denotes the reflexive-transitive closure of R . If $\alpha_0 \Rightarrow \alpha_1 \dots \Rightarrow \alpha_n$, $n \geq 1$, this is denoted by $\alpha_0 \Rightarrow^n \alpha_n$.

Example 3.1. A system of regular expressions. Let R be the system of regular expressions defined as follows:

1. $\Sigma = \{x, y\}$
2. $N = \{A, B\}$
3. $N_0 = A$
4. The productions in P are:

$$A \rightarrow ((x \mid B)^* B)$$

$$B \rightarrow (y \mid A)$$

Some derivations for regular expression forms of R are

- a. $A \Rightarrow ((x \mid B)^* B) \Rightarrow ((x \mid (y \mid A))^* B) \Rightarrow ((x \mid (y \mid A))^* (y \mid A))$
- b. $A \Rightarrow ((x \mid B)^* B) \Rightarrow ((x \mid B)^* (y \mid A))$

Let A be α_0 . Notice that there are no words defined for R by α_1 and α_2 in example a; for each α_i there are strings x , such that x is in $L(\alpha_i)$. However, there are no x 's such that x is in Σ^* until α_3 . In example b, using a different derivation sequence, this condition is satisfied by α_2 . For example, the strings $x^i y$, for any $i \geq 0$ are in $(L(\alpha_2) \cap \Sigma^*)$.

Definition 3.2. Systems of finite automata. Given the definition in Chapter 2 of a nondeterministic NFA as a 5-tuple, $M = (Q, \Sigma, \delta, q_0, F)$, then a system of finite automata S is defined to be a 7-tuple, $S = (M, N, \Sigma, Q, \Gamma, \Delta, M_0)$ where:

1. M is a finite set of NFAs, $M_i, i=1\dots n$. The components of each M_i are denoted as $Q_i, \Sigma_i, \delta_i, q_{0i}$, and F_i .
2. N is a set of unique symbols (names) for $M_i, i=1,\dots,n$, such that $(N \cap \Sigma)=\emptyset$. The symbol for M_i is denoted as N_i .
3. Σ is the input alphabet for S . Each Σ_i is a subset of $(\Sigma \cup N)$.
4. Q is the set of states of S ; (M_i, q) , for all M_i and for each q in $M_i, i=1,\dots,n$.
5. Γ is a finite set of unique symbols representing the states in Q . The elements of Γ are the alphabet of the pushdown list.
6. Δ is a mapping from $(Q \times (\Sigma \cup e) \times (\Gamma \cup e))$ to $(Q \times (\Gamma \cup e))$, and is defined by $\delta_i, i=1\dots n$ as follows: For each δ_i , and for all $\delta_i(q,b) \rightarrow p$ and α in Γ^* :
 - A. if b is in $(\Sigma \cup e)$, then $\Delta((M_i, q), b, e) \rightarrow ((M_i, p), e)$.
 - B. if $b=N_j$ for some $j=1\dots n$, then
 1. $\Delta((M_i, q), e, e) \rightarrow ((M_j, q_{0j}), (M_i, p))$, and
 2. for each final state f in F_j , $\Delta((M_j, f), e, (M_i, p)) \rightarrow ((M_i, p), e)$
7. M_0 is the top level (or start) automaton, where M_0 is some $M_i, i=1\dots n$: $q_0=q_{0i}$ is the start state of S and $F_0=F_i$ is the set of the final states of S .

A configuration of S is a triple $((M_i, q), w, \alpha)$, where (M_i, q) represents the current state of the finite control, $(M_i, q)\alpha$ in Γ^+ represents the open NFAs and their respective current states, and w in Σ^* represents the remaining input. An initial configuration of S is $C_0=((M_0, q_0), w, e)$, for some w in Σ^* , and the final (accepting) configurations are $C_f=((M_0, f), e, e)$ for all f in F_0 . An input string w in Σ^* is accepted by S if $C_0 \vdash^* C_f$, for some f . The language recognized by S is denoted by $L(S)$. Moves are defined by Δ as described above. For each δ_i , and for all $\delta_i(q,b) \rightarrow p, \alpha$ in Γ^* :

- A. if b is in $(\Sigma \cup e)$, then $((M_i, q), b, \alpha) \vdash ((M_i, p), e, \alpha)$.
- B. if $b=N_j$ for some $j=1\dots n$, then
 1. $((M_i, q), e, \alpha) \vdash ((M_j, q_{0j}), e, (M_i, p)\alpha)$, and
 2. for each final state f in F_j , $((M_j, f), e, (M_i, p)\alpha) \vdash ((M_i, p), e, \alpha)$.

The moves in A are local moves and the moves in B are pushdown moves. All pushdown moves are e-moves. The moves in B.1 push (M_i, p) , the current NFA and next state, onto the pushdown list and move to the start state of the NFA for N_j . From the final states of N_j , the moves in B.2 pop (M_i, p) from the pushdown list and it becomes the current state. Because Δ is completely defined by $\delta_i, i=1, \dots, n, n \geq 1$, and Q is completely defined by $Q_i, i=1, \dots, n$, a system of finite automata can be represented in graph form that consists of a graph representation of each M_i in M . This is illustrated in Example 3.2.

Example 3.2. A system of finite automata. Let S be the system of finite automata:

1. $Q = \{(A,0), (A,1), (A,2), (A,3), (B,0), (B,1), (B,2)\}$
2. $\Sigma = \{x, y\}$
3. $N = \{A, B\}$
4. Γ contains a unique symbol for each state in Q .
5. $M_0 = A, q_0 = (A,0)$, and $F_0 = \{(A,3)\}$.
6. The NFAs of M are shown in Figures 3.1-3.2.

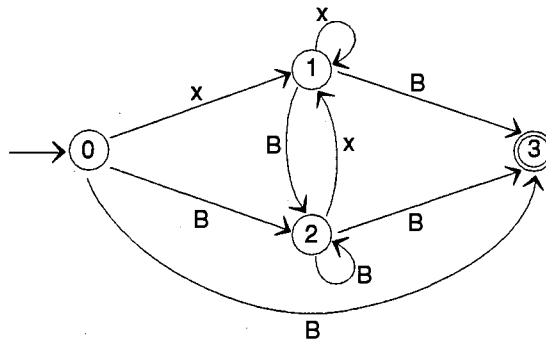


Figure 3.1. The NFA for A in Example 3.2.

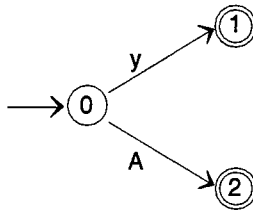


Figure 3.2. The NFA for B in Example 3.2.

7. Δ is completely defined by δ of A, δ of B, and Q. However, to illustrate this, the individual mappings of Δ are shown below. In future examples, systems of finite automata are represented in graph form as shown above, and the individual mappings of Δ are not explicitly shown.

a. local moves in A and B

$((A,0), x, e) \rightarrow ((A,1), e)$	local move on x
$((A,1), x, e) \rightarrow ((A,1), e)$	local move on x
$((A,2), x, e) \rightarrow ((A,1), e)$	local move on x
$((B,0), y, e) \rightarrow ((B,1), e)$	local move on y

b. Pushdown moves (pushes)

$((A,0), e, e) \rightarrow ((B,0), (A,2))$	push (A,2)
$((A,0), e, e) \rightarrow ((B,0), (A,3))$	push (A,3)
$((A,1), e, e) \rightarrow ((B,0), (A,2))$	push (A,2)
$((A,1), e, e) \rightarrow ((B,0), (A,3))$	push (A,3)
$((A,2), e, e) \rightarrow ((B,0), (A,2))$	push (A,2)
$((A,2), e, e) \rightarrow ((B,0), (A,3))$	push (A,3)
$((B,0), e, e) \rightarrow ((A,0), (B,2))$	push (B,2)

c. pushdown moves (pops)

$((A,3), e, (B,2)) \rightarrow ((B,2), e)$	pop (B,2)
$((B,2), e, (A,2)) \rightarrow ((A,2), e)$	pop (A,2)
$((B,2), e, (A,3)) \rightarrow ((A,3), e)$	pop (A,3)
$((B,1), e, (A,2)) \rightarrow ((A,2), e)$	pop (A,2)
$((B,1), e, (A,3)) \rightarrow ((A,3), e)$	pop (A,3)

3.2 Algorithms

ALGORITHM 3.1. Construct a system of finite automata, S, from a system of regular expressions, R, such that $L(S)=L(R)$.

Input: A system of regular expressions, R .

Output: A system of finite automata, S , such that $L(S)=L(R)$.

Method:

Step 1: Let Σ of S be Σ of R .

Step 2: Let N of S be N of R .

Step 3: For each production $A \rightarrow \alpha$ in R , construct an NFA for α using Thompson's construction [2]. Remove the ϵ -transitions from the NFA using Algorithm 5.2.a (in Chapter 5). Add the NFA to M of S , and let A added to N of S in Step 2 be the name of the NFA. Thus, M of S consists of a finite set of NFAs, one constructed from the right side of each production in R .

Step 4: Let the name of the top level NFA M_0 of M be N_0 , the start symbol of R . For each M_i in M of S , and for each q in Q_i of M_i , let (M_i, q) be in Q . For each state (M_i, q) in Q let (M_i, q) be a symbol in Γ . Δ of S is completely defined by δ_i , for all M_i in M of S .

PROOF. Proof is given by Theorem 3.1, after supporting definitions, and lemmas.

Example 3.3. A system of automata constructed from a system of regular expressions.

Let R be the system of regular expressions from Example 3.1:

1. $\Sigma = \{a, b\}$
2. $N = \{A, B\}$
3. $N_0 = A$
4. The productions in P are:

$$A \rightarrow ((x \mid B)^* B)$$

$$B \rightarrow (y \mid A)$$

The system of automata constructed from R by Algorithm 3.1 is $S=(M,N,Q,\Sigma,\Gamma,\Delta,M_0)$

where:

1. $N = \{A, B\}$
2. $\Sigma = \{x, y\}$

3. $Q = \{(A,0), (A,1), (A,2), (A,3), (A,4), (A,5), (A,6), (A,7), (A,8), (B,0), (B,1), (B,2), (B,3), (B,4), (B,5)\}$
4. $\Gamma = Q$.
5. $M_0 = A$
6. Δ is completely defined by the automata of M below.
7. The NFAs of M (before removing e-transitions) are shown in Figures 3.3-3.4.

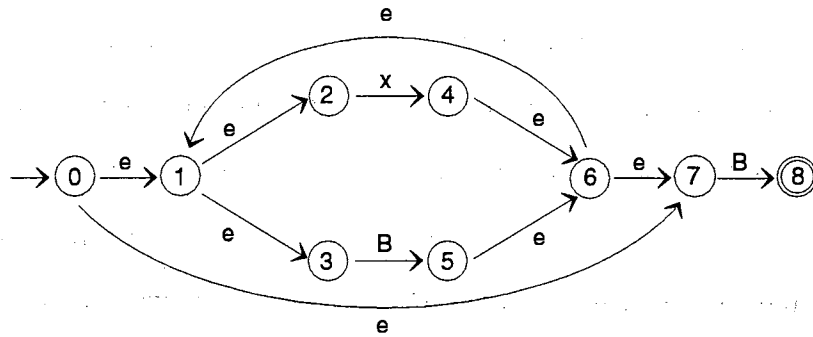


Figure 3.3. The NFA for A in Example 3.3

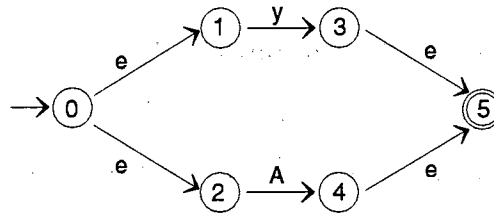


Figure 3.4. The NFA for B in Example 3.3.

Definition 3.3. Expansions of systems of automata. Consider any system of regular expressions R and the system of automata S constructed from R by Algorithm 3.1. Let $A \rightarrow \alpha_A$ be in P of R , where A is the start symbol of R . Then A is the first regular expression form of any derivation sequence of R , and α_A is the second regular expression form in any derivation sequence of R . In S , the NFA for A is constructed from the regular expression form, α_A . Construct a system of automata S' from R as follows:

1. Let B be any nonterminal symbol in N , where $B \rightarrow \alpha_B$ is in R . In the regular expression form α_A , replace any one occurrence of B with α_B . This is the first optional regular expression form in any derivation sequence of R . Denote this regular expression form as A_1 . The derivation for A_1 is $A \Rightarrow \alpha_A \Rightarrow A_1$.
2. Because A_1 is a regular expression form, it is a regular expression over $(\Sigma \cup N)$. Construct an NFA by applying Thompson's construction to A_1 . Let this be the NFA for A in S' .
3. Let the rest of S' be exactly the same as S .

The resulting system of automata, S' , is called a first order expansion of S , and is denoted as S^1 . Consider any occurrence of a nonterminal C in A_1 , such that $C \rightarrow \alpha_C$. C may be replaced by α_C to derive A_2 . Thus $A \Rightarrow \alpha_A \Rightarrow A_1 \Rightarrow A_2$. Step 2 is applied to A_2 to derive the NFA for A in the new S' . This S' is a second order expansion of S , denoted as S^2 . Expansions may be continued in this way, for any S^n , $n \geq 1$, as long as A_{n-1} contains some nonterminal symbol to replace. α_A is called the zeroth order expansion of S and is denoted by S^0 . Thus, for every regular expression form except for the start symbol of R , there is a corresponding expansion of S , S^i , $i \geq 0$.

Example 3.4. A first order expansion of S . Let R be the system of regular expressions from Example 3.2:

$$A \rightarrow ((x|B)^* B)$$

$$B \rightarrow (y|A)$$

The NFA for A in S is shown in Figure 3.3. Replace the second occurrence of B in $\alpha_A = ((x|B)^* B)$ with $\alpha_B = (y|A)$. This gives the regular expression form $A_1 = ((x|B)^* (y|A))$. The NFA constructed for A_1 becomes the NFA of A in S^1 as shown in Figure 3.5. The states labeled B/0-B/5 are the states that are added to the NFA of A in S^1 by replacing the component NFA for B with the component NFA for α_B . The NFA for B is shown in Figure 3.6.

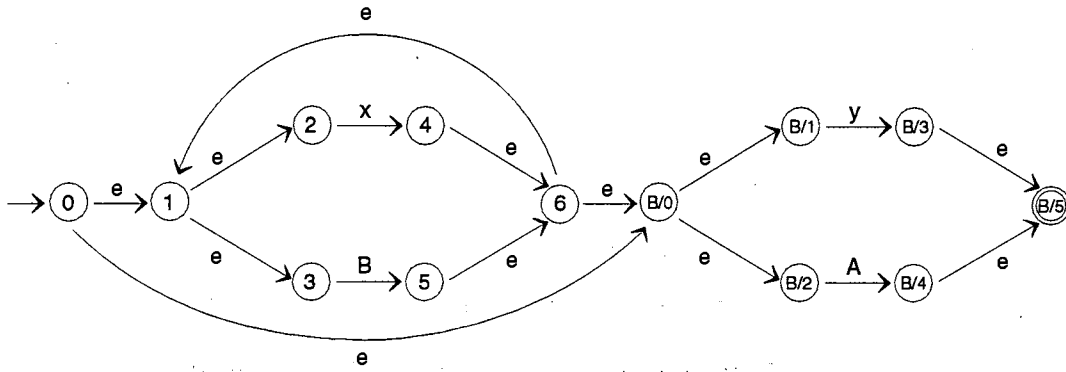


Figure 3.5. The NFA for A_1 in Example 3.4.

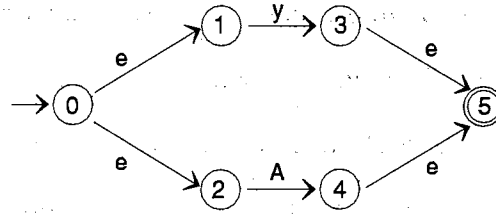


Figure 3.6. The NFA for B in Example 3.4.

LEMMA 3.1. *Let s be any regular expression, and let s' be a regular expression derived from s by replacing any one occurrence of a symbol B in s by a regular expression r . Let $M(s)$ and $M(s')$ be the NFAs constructed for s and s' by Thompson's construction [2]. Then, $M(s')$ will be exactly the same as $M(s)$ except that the component NFA constructed for B in $M(s)$ will be replaced in $M(s')$ by the component NFA constructed for r .*

PROOF. The first step in Thompson's construction is to construct a parse tree for the regular expression; each subtree represents a subexpression. Then the parse tree is traversed and NFAs are constructed inductively for each subexpression from the component NFAs for its respective subexpressions.

1. The subtree for B (a leaf node) in the parse tree of s will be replaced by the subtree for r in the parse tree of s' .

2. The component NFA constructed for each subexpression has the following properties [2]:
 - a. There is exactly one start state, q , and one final state, p , distinct from q .
 - b. There are no arcs entering q and no arcs leaving p .
3. When the inductive rules are applied to a component NFA they are applied independently of the value of the regular expression from which the component is constructed; there are no arcs constructed that enter any state of the component NFA other than the start state q , and there are no arcs constructed that leave any state of the component NFA other than the final state p .

Let q and q' be the start states of $M(B)$ and $M(r)$ respectively and p and p' be the final states of $M(B)$ and $M(r)$ respectively. Then $M(s')$ is constructed exactly the same as $M(s)$ except that q' replaces q and p' replaces p . For all states t and u in Q of $M(s)$ except for q and p , and any b in $(\Sigma \cup e)$, if $\delta(t,b) \rightarrow q$ in $M(s)$, then $\delta(t,b) \rightarrow q'$ in $M(s')$, and if $\delta(p,b) \rightarrow u$ in $M(s)$, then $\delta(p',b) \rightarrow u$ in $M(s')$. For the special case of B^* in $M(s)$, where $\delta(p,e) \rightarrow q$, then $\delta(p',e) \rightarrow q'$ in $M(s')$. Thus for all y in $L(r)$, all moves $(t,bB) \vdash (q,B) \vdash (p,e)$ in $M(s)$ are removed and replaced by $(t,by) \vdash (q',y) \vdash^* (p',e)$ in $M(s')$, and all moves $(q,Bb) \vdash (p,b) \vdash (u,e)$ in $M(s)$ are removed and replaced by $(q',yb) \vdash^* (p',b) \vdash (u,e)$ in $M(s')$. \square

In the lemmas and theorems that follow, this result will be applied to expansions of systems of automata. Figures 3.3 and 3.5 illustrate this application of Lemma 3.1. Figure 3.3 in Example 3.3 shows the NFA constructed for the right side of the production for A , $\alpha_A = ((x|B)^* B)$. Figure 3.5 in Example 3.4 shows the NFA constructed for $A_1 = ((x|B)^* (y|A))$, in which $\alpha_B = (y|A)$ (the right side of the production for B) has replaced the second occurrence of B in α_A . The NFA for A in Figure 3.3, $M(\alpha_A)$, is exactly the same as the NFA for A_1 in Example 3.5, $M(A_1)$, except that the component NFA constructed for B in $M(\alpha_A)$ is replaced in $M(A_1)$ by the component NFA constructed for α_B .

In Figure 3.5 all states in $M(A_1)$ that are from $M(\alpha_B)$ are shown as B/t , where t was the name of the state in α_B . Thus, these states in S^1 are denoted by $(A, B/t)$. This notation, will be used in Lemma 3.2.

Lemma 3.1 shows that the transitions into state (A,7) in Figure 3.3 are exactly the same as the transitions into (A,B/0) in Figure 3.5. It also shows that the transitions leaving state (A,8) in Figure 3.3 are exactly the same as the transitions leaving state (A,B/5) in Figure 3.5. This result from Lemma 3.1 will be used in Lemma 3.2.

LEMMA 3.2. *If S^n is an n^{th} order expansion of S , for some $n \geq 1$, then $L(S^n) = L(S)$.*

PROOF. The proof is by induction.

Basis: First, the proof is shown for $n=1$. Let R be any system of regular expressions and let S be the system of automata constructed from R by Algorithm 3.1. Let $A \rightarrow \alpha_A$ and $B \rightarrow \alpha_B$ be in P of R and let A be the start symbol of R . Let A_1 be the regular expression form derived from α_A by replacing some occurrence of B in α_A with α_B . Let S^1 be the first order expansion of S defined by A_1 .

Case A: Consider any word w in $L(S)$. This implies S makes some sequence of moves $((A, q_0), w, e) \vdash^* ((A, f), e, e)$ for some f in F of $M(\alpha_A)$. Suppose this sequence of moves does not make a transition on the occurrence of B that is in replaced in S^1 . Then by Lemma 3.1, S^1 has exactly the same sequence of moves as S , and thus w is in $L(S^1)$.

Suppose the sequence of moves of S recognizing w makes some transition on the occurrence of B that is replaced in S^1 . Let $w = xyz$ for some x , y , and z in Σ^* . Then by Definition 3.2, for some p and q in Q of $M(\alpha_A)$, f in F of $M(\alpha_A)$ and f'' in F of $M(\alpha_B)$, the moves of S are of the form: $((A, q_0), w, e) \vdash^* ((A, q), yz, e) \vdash ((B, q_0), yz, (A, p)) \vdash^* ((B, f''), z, (A, p)) \vdash ((A, p), z, e) \vdash^* ((A, f), e, e)$.

Let $y = y_1, \dots, y_n$, $n \geq 1$, where each y_i may be e . The subsequence of moves of S recognizing y in w can be shown in expanded form as: $\dots((A, q), yz, e) \vdash ((B, q_0), yz, (A, p)) \vdash ((B, q_1), y_2, \dots, y_n z, (A, p)) \dots \vdash ((B, q_n), y_n z, (A, p)) \vdash ((B, f''), z, (A, p)) \vdash ((A, p), z, e) \dots$

Then by Definition 3.2 and Lemma 3.1, S^1 has a corresponding subsequence of moves on y : $\dots((A/B, q_0), yz, e) \vdash ((A/B, q_1), y_2, \dots, y_n z, e) \dots \vdash ((A/B, q_n), y_n z, e) \vdash ((A/B, f''), z, e) \dots$, for q_i , $i=1, \dots, n$ in Q of $M(\alpha_B)$ and for q_0 the start state and f'' in F of $M(\alpha_B)$. Also by Lemma 3.1, the transitions into q in $M(\alpha_A)$ are exactly the same as the transitions into

B/q_0 in $M(A_1)$, and the transitions from p in $M(\alpha_A)$ are exactly the same as the transitions from B/f'' in $M(A_1)$. Thus, S^1 must make the sequence of moves: $((A, q_0), xyz, e) \vdash^* ((A, B/q_0), yz, e) \vdash^* ((A, B/f''), z, e) \vdash^* ((A, f), e, e)$.

Thus if w is in $L(S)$, w is in $L(S^1)$, and therefore $L(S) \subseteq L(S^1)$.

Case B: A similar proof holds for $L(S^1) \subseteq L(S)$. Consider any word $w=xyz$ in $L(S^1)$. This implies S^1 makes some sequence of moves $((A, q_0), w, e) \vdash^* ((A, f), e, e)$ for some f in F of $M(A_1)$. Suppose this sequence of moves does not make a transition on the occurrence of B that replaced in S^1 . Then by Lemma 3.1, S^1 has exactly the same sequence of moves as S , and thus w is in $L(S)$.

Suppose the sequence of moves of S recognizing w makes some transition on the occurrence of B that is replaced in S^1 . Then by Definition 3.2, if $w=xyz$ for some x , y , and z in Σ^* the moves of S^1 are of the form: $((A, q_0), xyz, e) \vdash^* ((A, B/q_0), yz, e) \vdash^* ((A, B/f''), z, e) \vdash^* ((A, f), e, e)$. Let $y=y_1 \dots y_n$, $n \geq 1$, where any y_i may be e . The subsequence of moves of S^1 recognizing y in w can be shown in expanded form as:
 $\dots((A, B/q_0), yz, e) \vdash ((A, B/q_1), y_2, \dots y_n z, e) \dots \vdash ((A, B/q_n), y_n z, e) \vdash ((A, B/f''), z, e) \dots$

Then by Definition 3.2 and Lemma 3.1, S has a corresponding subsequence of moves on y : $\dots((A, q), yz, e) \vdash ((B, q_0), yz, (A, p)) \vdash ((B, q_1), y_2, \dots y_n z, (A, p)) \dots \vdash ((B, q_n), y_n z, (A, p)) \vdash ((B, f''), z, (A, p)) \vdash ((A, p), z, e) \dots$ Also by Lemma 3.1, the transitions into q in $M(\alpha_A)$ are exactly the same as the transitions into B/q_0 in $M(A_1)$, and the transitions from p in $M(\alpha_A)$ are exactly the same as the transitions from B/f'' in $M(A_1)$. Therefore, S must make the sequence of moves $((A, q_0), xyz, e) \vdash^* ((A, q), yz, e) \vdash ((B, q_0), yz, (A, p)) \vdash^* ((B, f''), z, (A, p)) \vdash ((A, p), z, e) \vdash^* ((A, f), e, e)$.

Thus, if w is in $L(S^1)$, w is in $L(S)$, and this implies $L(S^1) \subseteq L(S)$.

Therefore, by A and B, $L(S)=L(S^1)$.

Inductive step: Suppose $L(S)=L(S^n)$, for any $n \geq 1$. Then by reapplying the proof used for $n=1$, the same results show that $L(S^n)=L(S^{n+1})$.

Therefore, by the basis and inductive steps, $L(S)=L(S^n)$ for $n \geq 1$.

THEOREM 3.1. *For every system of regular expressions, R , there is a system of finite automata, S , such that $L(S)=L(R)$.*

PROOF. Consider any R . Construct a system of automata, S , from R using Algorithm 3.1. The following proof shows that $L(S)=L(R)$.

Case A: Prove $L(R)$ is a subset of $L(S)$. Suppose w is in $L(R)$. By Definition 3.1 if w is in $L(R)$, then there is a regular expression form α of R , such that w is in $L(\alpha)$ and w is in Σ^* . Because α is a regular expression form of R , there is some α_i , $i=1, \dots, n$, $n \geq 1$, such that $N_0 \Rightarrow \alpha_1, \dots \Rightarrow \alpha_{n-1} \Rightarrow \alpha$, and one nonterminal in each α_{i-1} is replaced in α_i . Then by Definition 3.3, there is some expansion S^n defined by R and α . The NFA for M_0 in S^n (for N_0 , the start symbol of R) is constructed by applying Thompson's method to α . Thus, $M_0=M(\alpha)$ recognizes $L(\alpha)$, and since w is in $L(\alpha)$ and w is in Σ^* , $M(\alpha)$ must make some sequence of moves $(q_0, w) \vdash^* (f, e)$ for some f in M_0 , and therefore by Definition 3.2, S^n must make a sequence of moves $((M_0, q_0), w, e) \vdash^* ((M_0, f), e, e)$. This implies w is in $L(S^n)$. Then, by Lemma 3.2, $L(S^n)=L(S)$. Thus, w is in $L(S)$ and therefore, $L(R)$ is a subset of $L(S)$.

Case B: Prove $L(S)$ is a subset of $L(R)$. To prove this it must be shown that if w is in $L(S)$, then w is in $L(R)$. The method used is to show that there is an expansion of S , S^n for some $n \geq 0$, such that the top NFA, M_0 of S^n recognizes w . For all S^n , M_0 is an NFA constructed from some regular expression form α of R such that $L(M_0)=L(\alpha)$. Then if w is in $L(M_0)$ w is in $L(\alpha)$, and thus w is in $L(R)$.

Let A denote the start symbol of R . Thus A is also the name of M_0 of S and M_0 of any S^n . Suppose w is in $L(S)$. This implies there is some sequence of moves of S , $((A, q_0), w, e) \vdash^* ((A, f), e, e)$ for some final state f of A . If there are no pushdown moves in this sequence, then S^0 satisfies the requirements that w is recognized by the top NFA of some S^n ; in particular, S^0 . That is, if $A \rightarrow \alpha$ is in P of R , w is in $L(\alpha)$, and thus w is in $L(R)$.

Consider any sequence of moves of S recognizing w in which there are pushdown moves. This implies that for some y and z in Σ^* there some $j \geq 1$ subsequences of moves of

S on w of the form: $\dots((A,q),yz,e) \vdash ((B,q_0),yz,(A,p)) \vdash^* ((B,f),z,(A,p)) \vdash ((A,p),z,e)\dots$

where the sequence corresponds to some arc labeled B in A. For each configuration in the sequence of moves recognizing w, the pushdown list has a finite number of symbols k, $k \geq 0$.

Begin with the regular expression form α_A , where $A \rightarrow \alpha_A$ is in R. Let α_0 denote α_A . Repeatedly construct regular expression forms α_i , $i=i+j$ by replacing all nonterminal symbols in the regular expression form α_{i-j} . The derivations are of the form $\alpha_{i-j} \Rightarrow \alpha_{i-j+1} \dots \Rightarrow \alpha_i$ where all j nonterminals in α_{i-j} have been replaced in α_i . Then there are expansion of S, S^{i-j} , S^{i-j+1}, \dots, S^i , where S^i is constructed from α_i . Thus for each α_i , there is an expansion of S, S^i .

By Lemma 3.1, in each S^i constructed, for every sequence of moves in S^{i-j} there is a corresponding sequence of moves in S^i , except that the top level pushdown moves in S^{i-j} are removed in S^i . Thus for each S^i any sequence of moves recognizing w has the following property: for each configuration in the sequence, if $k \geq 0$, then in the corresponding configuration of S^i , the number of symbols on the stack is $k-1$.

Thus, by repeatedly deriving α_i and constructing S^i , it must eventually true for some S^i that $k=0$; every configuration in the sequence of moves recognizing w has an empty stack. Then, because there are no pushdown moves in the sequence, and it must be true that the top NFA, A, in S^i recognizes w. Since A is M_0 of S^i , by the construction of S^i , the NFA for A in S^i recognizes $L(\alpha_i)$. Thus, w is in α_i , and by Definition 3.2, w is in $L(R)$.

Therefore, by A and B, $L(S)=L(R)$

Definition 3.4. Useless NFAs in a system of finite automata. In a system of finite automata S, an NFA M_j in M is useless iff there is no x in Σ^* , f in F_j , and β in Γ^* such that $((M_j, q_0, x, \beta) \vdash^* ((M_j, f), e, \beta))$. An NFA is useful iff it is not useless.

Definition 3.5. Inaccessible NFAs in a system of finite automata. Let S be a system of finite automata. An NFA M_j is inaccessible iff there is no x in Σ^* and β in Γ^* such that $((M_0, q_0), x, e) \vdash^* ((M_j, q_0), e, \beta)$. An NFA is accessible iff it is not inaccessible.

Definition 3.6. Useless states in a system of finite automata. Let S be a system of finite automata. A state (M_i, p) in S is useless iff there is no x and y in Σ^* and β in Γ^* such that $((M_0, q_0), xy, e) \vdash^* ((M_i, p), y, \beta) \vdash^* ((M_0, f), e, e)$ for some f in F_0 .

ALGORITHM 3.2. Derive the set of useful NFAs in a system of finite automata.

Input: A system of finite automata S .

Output: The set U , the names of the useful NFAs in S .

Method:

Step 1: Let $U_0 = \emptyset$ and $n=1$.

Step 2: Let $U_n = (U_{n-1} \cup U')$ where U' is derived as follows: for each N_j in $(N - U_{n-1})$ perform steps A-D:

A. Let $A_0 = \{q_0\}$ where q_0 is the start state of M_j and let $m=1$.

B. Let $A_m = (A_{m-1} \cup A')$ where A' is derived as follows: for all q in A_{m-1} , p in Q_n , and b in $(\Sigma \cup U_{n-1})$, such that for δ of M_j , $\delta(q, b) \rightarrow p$, add p to A' .

C. If $A_m \neq A_{m-1}$ then let $m=m+1$ and repeat Step B. Else let $A = A_m$.

D. A is the set of states in M_j that can be reached on either an input symbol in Σ or on a nonterminal in U_{n-1} . If there is any state f in A such that f is in F_j , then add N_j to U' .

Step 3: If $U_n \neq U_{n-1}$, let $n=n+1$ and repeat Step 2. Else let $U = U_n$, output U , and halt.

PROOF. N_j is in U iff M_j is useful.

Case 1: Suppose N_j is in U . The following proof by induction shows that if N_j is in U_n , for some $n \geq 1$, then M_j is useful.

Basis. Consider U_0 . This holds vacuously because $U_0 = \emptyset$.

Inductive step. Suppose for all N_j in U_{n-1} , M_j is useful. Consider all N_k in U_n . By Step 2 of Algorithm 3.2, for M_k , either q_0 of M_k is in F_k or there is some string $x = x_1 \dots x_m$, $m \geq 1$, in $(\Sigma \cup U_{n-1})$ such that $\delta(q_0, x_1) \rightarrow q_1 \dots \delta(q_{m-1}, x_m) \rightarrow q_m$, such that q_m is in F_j , and for each x_i , $i=1, \dots, m$, x_i is either in Σ or in U_{n-1} .

If x_i is in Σ , then by Definition 3.2, $((M_k, q_{i-1}), x_i, \beta) \vdash ((M_k, q_i), e, \beta)$. If x_i is some N_j in N , then by Definition 3.2, $((M_k, q_{i-1}), e, \beta) \vdash ((M_j, q_0), e, (M_k, q_i)\beta)$ and for all f in F_j ,

$((M_j, f), e, (M_k, q_i)\beta) \vdash ((M_k, q_i), e, \beta)$. By the inductive assumption for N_j in U_{n-1} , N_j is useful and thus for some y_i in Σ^* and f in F_j , $((M_j, q_0, y_i), (M_k, q_i)\beta) \vdash^* ((M_j, f), e, (M_k, q_i)\beta)$.

Thus there is some $v = v_1, \dots, v_m$, such that each v_i , $i=1, \dots, m$ is either u_i in Σ or is some string y_i in Σ^* and $((M_k, q_{i-1}), v_i, \beta) \vdash ((M_k, q_i), e, \beta)$. Therefore, $((M_k, q_0), v, \beta) \vdash^+ ((M_k, q_m), e, \beta)$ where q_m is in F_j .

If the result holds for any U_{n-1} , it holds for U_n , and therefore if N_j is in some U_n , for some $n \geq 1$, then M_j is useful.

Case 2: If M_j is useful, then N_j is in U . The following proof shows that if N_j is not in U , then N_j is useless. Then because $\wedge Q$ implies $\wedge P$ if and only if P implies Q , the result holds.

By the last iteration of Step 2 of Algorithm 3.2, there is no N_j in $N-U$, such that there is any string y , such that y is in $L(M_j)$ and y is in $(\Sigma \cup U)^*$. Thus any accepting sequence of moves of M_j must be of the form $(q_0, x) \vdash (q_1, x_2, \dots, x_m) \vdash (q_{i-1}, x_m) \vdash (q_m, e)$ where q_m is in F_j , for some $x = x_1, \dots, x_m$, $m \geq 1$, and such that for some i , $i=1, \dots, m$, $x_i = N_k$ for some N_k in $(N-U)$. Thus $(q_{i-1}, x_i) \vdash (q_i, x_{i+1})$ for $i=1, \dots, m$, and some x_i is N_k in $(N-U)$.

Then, for each N_j in $N-U$, it must be true for N_k that there is a transition in M_k on some N_g in $N-U$, $\delta(q, N_g) \rightarrow p$, q and p in Q_k . Then for some y, z in Σ^* if $((M_j, q_{i-1}), y, \beta) \vdash ((M_k, q_0), y, (M_j, q_i)\beta) \vdash^+ ((M_k, q), e, (M_j, q_i)\beta) \vdash ((M_g, q_0), e, (M_k, p)(M_j, q_i)\beta)$, then the moves $((M_k, q_0), y, (M_j, q_i)\beta) \vdash^+ ((M_k, q), e, (M_j, q_i)\beta)$ contain no move of the form $((M_k, f), u, (M_j, q_i)\beta) \vdash ((M_j, q_i), u, \beta)$ for any u in Σ^* and any (M_k, f) in Q such that f is in F_k .

Thus any sequence of moves beginning with $((M_j, q_0), y, \beta)$ for any y in Σ , β in Γ^* , such that N_j is in $N-U$, can never terminate. There will always be a symbol (M_k, q_0) for N_k in $N-U$ placed on the stack before the previous stack top, (M_j, q_i) , can be popped.

Thus, if N_j is in $N-U$ ($\wedge Q$), then M_j is useless ($\wedge P$). Then P implies Q , and thus if M_j is useful then N_j is in U .

Therefore by 1 and 2, N_j is in U iff M_j is useful.

ALGORITHM 3.3. Remove useless NFAs from a system of finite automata.

Input: A system of finite automata, S .

Output: A system of finite automata S' such that S' has no useless NFAs and $L(S')=L(S)$. The output is "undefined" if M_0 is useless.

Method:

Step 1: Remove the ϵ -transitions from all NFAs in M of S using Algorithm 5.2.a.

Step 2. Use Algorithm 3.2 to derive the set U , the names of the useful NFAs of S . If N_0 , the name of M_0 of S is not in U then halt, and output "undefined." Note that if N_0 is not in U , then $L(S)=\emptyset$.

Step 3: For each N_j in $N-U$, remove M_j from M , remove N_j from N , remove all states (M_j, q) for all q from Q , and remove the corresponding symbol for (M_j, q) from Γ .

Step 4: For each N_k in U and for each q and p in Q_k , for all N_j in $N-U$ such that $\delta(q, N_j) \rightarrow p$, remove $\delta(q, N_j) \rightarrow p$ from δ of M_k . Remove all inaccessible states from M_k using Algorithm 5.2.b and use Algorithm 5.2.c to remove all useless states from M_k . For all states q removed from Q_k , remove (M_k, q) from Q and remove the symbol for (M_k, q) from Γ .

Step 5: Let $M'=M$, $N'=N$, $\Sigma'=\Sigma$, $Q'=Q$, $\Gamma'=\Gamma$, $M'_0=M_0$. Δ' is completely defined by all δ_i of M_i , for all M_i in M' . Let $S'=\{M', N', \Sigma', Q', \Gamma', M'_0, \Delta'\}$

PROOF. No states or moves are added to S' so if x is in $L(S')$ then x is $L(S)$. Every M_j removed by Step 3 is useless, and thus there is no xyz , x , y , and z in Σ^* such that $((M_0, q_0), xyz, e) \vdash^* ((M_j, q_0), yz, \beta) \vdash^* ((M_j, f''), z, \beta) \vdash ((M_0, f), e, e)$ for some f'' in F_j and f in F_0 .

For every N_k in U , for each $\delta(q, N_j) \rightarrow p$ in M_k removed by Step 4, because N_j is useless, for any x , y , and z in Σ^* and β in Γ^* there is no sequence of moves $((M_k, q), y, \beta) \vdash ((M_j, q_0), y, (M_k, p)\beta) \vdash^* ((M_j, f''), e, (M_k, p)\beta) \vdash ((M_k, p), e, \beta)$. Thus, there is no sequence of moves $((M_0, q_0), xyz, e) \vdash^* ((M_k, q), yz, \beta) \vdash^* ((M_k, p), z, \beta) \vdash ((M_0, f), e, e)$ for f in F_0 .

Therefore, if x is in $L(S)$ then x is in $L(S')$, and therefore, $L(S')=L(S)$.

ALGORITHM 3.4. Derive the set of accessible NFAs in a system of finite automata.

Input: A system of finite automata S with useless NFAs removed by Algorithm 3.3, such that M_0 is in M of S .

Output: The set A , the names of accessible NFAs in S .

Method:

Step 1: Let $A_0 = \{N_0\}$ and $n=1$.

Step 2: Let $A_n = A_{n-1} \cup A'$ where A' is constructed as follows. For each N_j in A_{n-1} , for δ of M_j and for all q and p in Q_j , if $\delta(q, N_k) \rightarrow p$, then add N_k to A' .

Step 3: If $A_n \neq A_{n-1}$ then let $n=n+1$ and repeat Step 2. Else let $A = A_n$.

PROOF. N_j is in A iff M_j is accessible.

Case 1: Suppose N_j is in A . The following proof by induction shows that if N_j is in A_n , for some $n \geq 0$, then M_j is accessible.

Basis. Consider A_0 . Let $m=0$. If N_j is in A_0 , then $M_j = M_0$ and the result holds.

Inductive step. Suppose for all N_j in A_{n-1} that M_j is accessible. The following proof shows for all N_k in A_n that M_k is accessible. Consider N_j in A_{n-1} . By Definition 3.5, there is some x in Σ^* and β in Γ^* such that $((M_0, q_0), x, e) \vdash^* ((M_j, q_0), e, \beta)$.

Consider any N_k in A_n . Then by Algorithm 3.4, there is some M_j in A_{n-1} such that for δ of M_j and some q and p in Q_j , $\delta(q, N_k) \rightarrow p$. Then because all inaccessible states in M_j have been removed by Step 3 of Algorithm 3.2, there is some $u = u_1, \dots, u_m$, $m \geq 1$, such that $u_m = N_k$ and $\delta(q_0, u_1) \rightarrow q_1, \dots, \delta(q_{m-2}, u_{m-1}) \rightarrow q_{m-1}$, $\delta(q_{m-1}, u_m) \rightarrow q_m$. Then for $i=1, \dots, m$, for each $\delta(q_{i-1}, u_i) \rightarrow q_i$, q_i is either in Σ or in A_n . Choose u so that u_m is the first occurrence of some N_k in u . Then by adding N_k to A_{n-1} and choosing another $u = u_1, \dots, u_{m'}, m' > m$, the following proof holds for any subsequent occurrences of N_k .

By Definition 3.2 if u_i is in Σ , then $((M_j, q_{i-1}), u_i, \beta) \vdash ((M_j, q_i), e, \beta)$, and if u_i is in A_{n-1} , $((M_j, q_{i-1}), e, \beta) \vdash ((M_k, q_0), e, (M_j, q_i)\beta)$ and for all f in F_k , $((M_k, f), e, (M_j, q_i)\beta) \vdash ((M_j, q_i), e, \beta)$. Then because all NFAs in S are useful, by Definition 3.4 there is some y_i in

Σ^* such that $((M_k, q_0), y_i, (M_j, q_i)\beta) \vdash ((M_k, f), e, (M_j, q_i)\beta)$ for f in F_k . Thus, for u_i in Σ $((M_j, q_{i-1}), u_i, \beta) \vdash ((M_j, q_i), e, \beta)$, and for u_i in A_{n-1} $((M_j, q_{i-1}), y_i, \beta) \vdash ((M_j, q_i), e, \beta)$.

Let $v = v_1, \dots, v_m$, where $v_i, i=1, \dots, m$ are in Σ^* , and v is constructed as follows: if u_i is in Σ then $v_i = u_i$ and if u_i is in A_{n-1} , then $v_i = y_i$. Thus, $((M_j, q_0), v, \beta) \vdash^* ((M_j, q_m), v_m, \beta)$. Then by Definition 3.2 because $u_m = N_k$, $((M_j, q_m), v_m, \beta) \vdash ((M_k, q_0), v_m, (M_j, q_m)\beta)$.

Thus, for xv_1, \dots, v_{m-1} , $((M_0, q_0), xv_1, \dots, v_{m-1}, e) \vdash^* ((M_j, q_0), v_1, \dots, v_{m-1}, \beta) \vdash ((M_j, q_m), e, \beta) \vdash ((M_k, q_0), e, (M_j, q_m)\beta)$, and if N_j is in A_n for $n \geq 0$, then M_j is accessible.

Case 2: Suppose M_j is accessible. The following proof by contradiction shows that if N_j is not in A , then M_j is inaccessible. Then because P implies Q iff $\neg Q$ implies $\neg P$, if M_j is accessible, then N_j is in A .

Suppose N_j is not in A and M_j is accessible. By Definition 3.5, M_j is accessible iff there is some x in Σ^* and β in Γ^* such that $((M_0, q_0), x, e) \vdash^* ((M_j, q_0), e, \beta)$. Then, by Definition 3.2 there is some sequence of NFAs in M , M_0, \dots, M_j , such that: for M_0 and some q_0 and p_0 in Q_0 , $\delta(q_0, N_1) \rightarrow p_0$, for M_1 and some q_1 and p_1 in Q_1 , $\delta(q_1, N_2) \rightarrow p_1, \dots$ for M_{j-1} and some q_{j-1} and p_{j-1} in Q_{j-1} , $\delta(q_{j-1}, N_j) \rightarrow p_{j-1}$. Note that q_0 does not necessarily denote the start state of M_0 . After the final iteration of Step 2 of Algorithm 3.4, N_1, \dots, N_j must be in A . This contradicts the assumption that N_j is not in A , and thus if N_j is not in A , then M_j is inaccessible. Then because P implies Q iff $\neg Q$ implies $\neg P$, if M_j is accessible, then N_j is in A .

Therefore by 1 and 2, N_j is in A iff M_j is accessible.

ALGORITHM 3.5. Remove inaccessible NFAs from a system of finite automata.

Input: A system of finite automata S .

Output: A system of finite automata S' with no inaccessible NFAs such that $L(S') = L(S)$.

Method:

Step 1: Use Algorithm 3.4 to derive the set A , the names of all accessible NFAs in S .

Step 2: For each N_j in $N - A$, remove N_j from N , remove M_j from M , remove all states (M_j, p) for all p from Q of S , and remove the symbol for (M_j, p) from Γ .

Step 3: For each N_k in A , for M_k and for all q and p in Q_k such that $\delta(q, N_j) \rightarrow p$, for N_j in $N-A$, remove $(q, N_j) \rightarrow p$ from δ of M_k . Use Algorithm 4.2.b to remove all inaccessible states from M_k [3] and use Algorithm 4.2.c to remove all useless states from M_k . For all states q removed from Q_k , remove (M_k, q) from Q and remove the symbol for (M_k, q) from Γ .

Step 4: Let $M'=M$, $N'=N$, $\Sigma'=\Sigma$, $Q'=Q$, $\Gamma'=\Gamma$, $M'_0=M_0$. Δ' is completely defined by all δ_i of M_i , for all M_i in M' . Let $S'=\{M', N', \Sigma', Q', \Gamma', M'_0, \Delta'\}$.

PROOF.

Case 1: No states or moves are added to S' so if w is in $L(S')$ then w is $L(S)$.

Case 2: Every M_j removed by Step 2 is inaccessible, and thus there is no xy in $L(S)$ such that $((M_0, q_0), xy, e) \vdash^* ((M_j, q_0), y, \beta) \vdash^* ((M_0, f), e,)$ for f in F_0 . For every N_k in A , for each $\delta(q, N_j) \rightarrow p$ removed by Step 3, because M_j is inaccessible, it is for any x, y and z in Σ^* , β in Γ^* , and (M_k, q) and (M_k, p) in Q , there is no sequence of moves $((M_0, q_0), xyz, e) \vdash^* ((M_k, q_0), yz, \beta) \vdash^* ((M_k, q), y, \beta) \vdash ((M_j, q_0), y, (M_k, p)\beta) \vdash^* ((M_0, f), e,)$ for f in F_0 . Thus, if w is in $L(S)$ then w is in $L(S')$.

Therefore, by 1 and 2, $L(S')=L(S)$.

ALGORITHM 3.6. Remove useless states from a system of automata.

Input: A system of finite automata S .

Output: A system of finite automata S'' , with no useless states such that $L(S'')=L(S)$

Method:

Step 1: Remove useless NFAs from S using Algorithm 3.3 to derive S' .

Step 2: Remove inaccessible NFAs from S' using Algorithm 3.5 to derive S'' .

PROOF. After Step 2, there are no useless or inaccessible NFAs in S'' and for all NFAs M_j in M'' , by Step 3 of Algorithm 3.5, there are no useless or inaccessible states in M_j , and by Step 1 of Algorithm 3.3 there are no e -transitions in M_j . Consider any state (M_j, q) in S'' .

Part 1: Because all NFAs in S'' are accessible, then for some x in Σ^* and β in Γ^* ,
 $((M_0, q_0), x, e) \vdash^* ((M_i, q_0), e, \beta)$.

Part 2: Because all inaccessible states have been removed from M_i , either $q_0 = q$ or there is some $u = u_1, \dots, u_m$, $m \geq 1$, such that u_i , $i = 1, \dots, m$ is in $(\Sigma \cup N)$ and for δ of M_i , $\delta(q_0, u_1) \rightarrow q_1, \delta(q_1, u_2) \rightarrow q_2, \dots, \delta(q_{m-1}, u_m) \rightarrow q$. If u_i is in Σ then $((M_i, q_{i-1}), u_i, \beta) \vdash ((M_i, q), e, \beta)$, and if u_i is some N_k in N , then by Definition 3.2, for each i , $((M_i, q_{i-1}), e, \beta) \vdash ((M_k, q_0), e, (M_i, q_i)\beta)$ and for all f in F_k $((M_k, f), e, (M_i, q_i)\beta) \vdash ((M_i, q_i), e, \beta)$. Because there are no useless NFAs in S'' , for each $u_i = N_k$ in N , there is some y_i in Σ^* and β' in Γ^* such that $((M_k, q_0), y_i, (M_i, q_i)\beta) \vdash^* ((M_k, f), e, (M_i, q_i)\beta)$. Thus $((M_i, q_{i-1}), y_i, \beta) \vdash^+ ((M_i, q), e, \beta)$. Let $v = v_1, \dots, v_m$ such that if u_i is in Σ , $v_i = u_i$ and if u_i is N_k in N , then $v_i = y_i$. Then,
 $((M_i, q_0), v, \beta) \vdash^* ((M_i, q), e, \beta)$.

Part 3: The proof in 2 can be applied to show that there is some v' in $(\Sigma \cup N)$ such that $((M_i, q), v', \beta) \vdash^* ((M_i, f'), e, \beta)$ for some f' in F_i .

Part 4: From 1, 2, and 3, there is a string xvv' in Σ^* such that $((M_0, q_0), xvv', e) \vdash^* ((M_i, q_0), vv', \beta) \vdash^* ((M_i, q), v, \beta) \vdash ((M_i, f), e, \beta)$ for f' in F_i . It only remains to be shown that there is a string z such that $((M_i, f'), z, \beta) \vdash^* ((M_0, f), e, e)$ for f in F_0 .

Case a: If $\beta = e$, then $M_i = M_0$ and by the proof in 3, there is some $z = v$ in Σ^* such that $((M_0, q), z, e) \vdash^* ((M_0, f), e, e)$ for f in F_0 .

Case b: Consider $\beta = \beta_1, \dots, \beta_n$, $n \geq 1$, such that β_1 is the top of the stack; β_k , $k = 1, \dots, n$ are symbols denoting states (M_k, r_k) , $k = 1, \dots, n$. Let f_k , $k = 1, \dots, n$ be a final state in F_k . By Definition 3.2, $((M_i, f'), z, \beta) \vdash ((M_1, r_1), z, \beta_2, \dots, \beta_n)$ and by the proof in 3, for some $z = z_1, \dots, z_n$ in Σ^* $((M_1, r_1), z_1, \dots, z_n, \beta_2, \dots, \beta_n) \vdash^* ((M_1, f_1), z_2, \dots, z_n, \beta_2, \dots, \beta_n) \vdash ((M_2, r_2), z_2, \dots, z_n, \beta_3, \dots, \beta_n) \dots ((M_{n-1}, r_{n-1}), z_{n-1}, z_n, \beta_n) \vdash^* ((M_{n-1}, f_{n-1}), z_n, \beta_n) \vdash ((M_n, f_n), z_n, e)$.

The bottom symbol on the stack must always be some state in M_0 , and thus (M_n, f_n) must be (M_0, f_n) for some f_n in F_0 . The proof in 3 shows that for some z_n $((M_0, f_n), z_n, e)$

$\vdash^* ((M_0, f), e, e)$ for some f in F_0 . Note that z_n may be e and f_n may be f . Then it must be true that for some z in Σ^* that $((M_i, q), z, \beta) \vdash^+ ((M_0, f), e, e)$ for some f in F_0 .

Therefore by Parts 1-4, there is some string $xvv'z$ such that $((M_0, q_0), xvv'z, e) \vdash^* ((M_i, q), v'z, \beta) \vdash^* ((M_0, f), e, e)$ for f in F_0 , and thus each (M_i, q) in Q'' of S'' is not a useless state. The proof that $L(S) = L(S') = L(S'')$ is already given in the proofs of Algorithms 3.3 and 3.5. \square

In this chapter, systems of regular expressions and a corresponding class of recognizers, systems of finite automata, are defined, and an algorithm is given for constructing a system of finite automata S from a system of regular expressions R , such that $L(S) = L(R)$.

Useless NFAs, inaccessible NFAs, and useless states in systems of finite automata are also defined, and algorithms are given for removing these from a system of finite automata. In Chapter 4, this language model (systems of finite automata from systems of regular expressions) is used to construct a parser for DTDs. The parser is used in Chapters 5 and 6 in algorithms to detect ambiguity in model groups and DTDs. Chapter 6 shows that the high level syntax of document instances can also be represented by systems of regular expressions, and thus systems of finite automata can be constructed that recognize document instances.

4. A Parser for DTDs

In this chapter a parser is constructed for the high level syntax of DTDs from the syntax productions shown in Chapter 2.2.1. First an equivalent regular expression is derived for the right hand side of each syntax production; the result is a system of regular expressions defining the syntax of DTDs. Using Algorithm 3.1, a system of finite automata recognizing DTDs is constructed from this system of regular expressions. Then a parser is constructed for DTDs from this system of automata. There are two purposes for constructing the parser: 1. It demonstrates that parsers for SGML syntax productions can be constructed using systems of finite automata derived from the syntax productions. 2. The parser is used in Chapters 5 and 6 in the algorithms for detecting ambiguity in model groups and ambiguity caused by omitted tags.

4.1 A System of Regular Expressions for DTDs

The expressions of syntax productions have equivalent regular expressions [13]. Thus, a system of regular expressions, R , that defines the syntax of DTDs can be derived from the syntax productions in Chapter 2.2.1 by the following steps:

1. In the syntax production for "element declaration", the syntactic variable "content model" in the expression for "element declaration" is replaced by its right hand side (the expression for "content model"). This is similar to removing single productions in context free grammars [3] and does not affect the language defined (DTDs).
2. For each syntax production of the form, $A = \text{expr1}$, convert expr1 to an equivalent regular expression, expr2 . Replace all occurrences of $B?$, where B is any subexpression of expr1 , with the equivalent regular expression $(B|e)$ in expr2 , and replace all occurrences of B^+ with BB^* [13].
3. Add the syntactic variable A to N of R and add $A \rightarrow \text{expr2}$ to P of R .
4. All tokens other than syntactic variables and operators in expr1 represent terminal symbols. Add each of these tokens to Σ of R .

5. Designate the syntactic variable, "DTD", as the start symbol of R.

By applying steps 1-4 to the system of syntax productions for DTDs given in Chapter 2.2.1, the following system of regular expressions for DTDs is derived.

```

DTD = "DOCTYPE", GI, element declaration, element declaration*
element declaration = MDO, "ELEMENT", element type, (omitted tag
    minimization | e), (declared content | (model group, (exceptions
    | e))), MDC
element type = GI | (name group)
omitted tag minimization = start tag minimization, end tag
    minimization
start tag minimization = "O" | MINUS
end tag minimization = "O" | MINUS
exceptions = (exclusions, (inclusions|e)) | inclusions
exclusions = MINUS, GRPO, GI, GI*, GRPC
inclusions = PLUS, GRPO, GI, GI*, GRPC
declared content = "CDATA" | "RCDATA" | "EMPTY"
model group = GRPO, content token, ( (AND, content token)* | (OR,
    content token)* | (SEQ, content token)* ) GRPC, ((OPT | PLUS |
    REP) | e)
content token = "#PCDATA" | (GI, ((OPT | PLUS | REP) | e)) | model
    group

```

4.2 A System of Finite Automata for DTDs

A system of finite automata recognizing DTDs can be constructed from the above system of regular expressions by applying Algorithm 3.1. The resulting system of finite automata is shown below in graph form. The only NFAs shown are for DTD, element declaration, model group, and content model; these are used in Chapters 5 and 6 in algorithms for detecting ambiguity. Because of the hierarchical nature of systems of finite

automata, the parser for model groups can be used independently of the parser for DTDs by considering only Figures 4.3-4.4 and Tables 4.3-4.4.

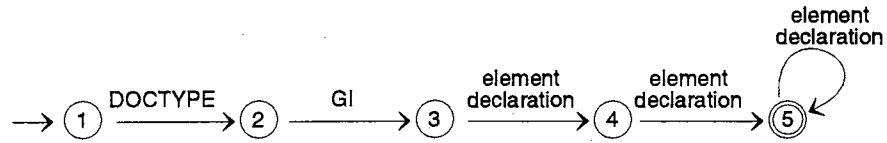


Figure 4.1. The NFA for DTD.

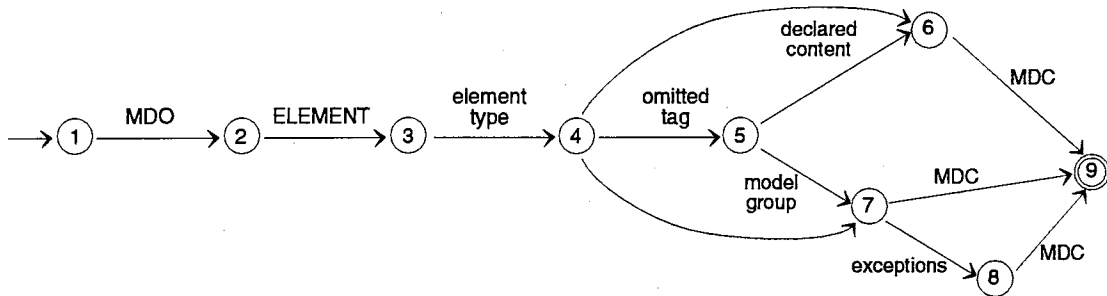


Figure 4.2. The NFA for element declaration.

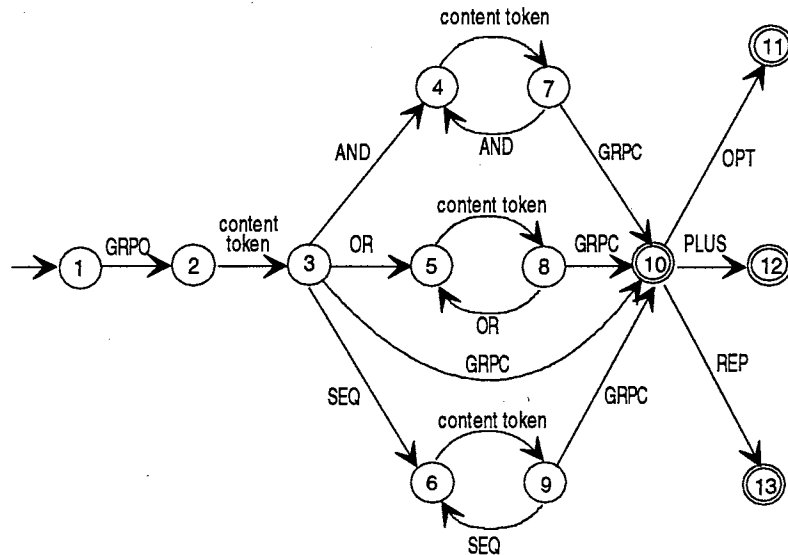


Figure 4.3. The NFA for model group.

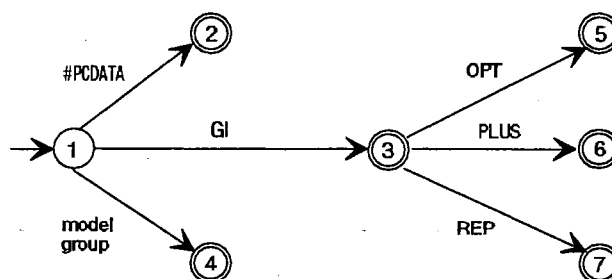


Figure 4.4. The NFA for content token.

4.3 A Parser for DTDs

A parser for DTDs can be implemented as a set of functions; each function implements the NFA for a nonterminal symbol in the system of finite automata shown above. For each transition $\delta(q, by) \rightarrow p$ in the NFA for a function, moves are made as follows: 1. if b is a terminal symbol then a local move is made. 2. If b is a nonterminal, then the current state of the NFA is set to p and a call is made to the function for the NFA named by b . For the final states of each NFA either a local move is made, the function returns to its calling function, or for the top level NFA, the parser may halt on a final configuration. Because all pushdown list moves in a system of finite automata are e -moves, for many configurations there may be more than one next move. Thus, systems of automata are inherently nondeterministic [3]. The parser shown in Tables 4.1-4.4 is a deterministic implementation of the above system of automata. This implementation requires a particular property: for each configuration there is at most one possible sequence of moves on the next input symbol. The parser uses a one symbol lookahead; the function `next_symbol()` returns the next input symbol without advancing the input pointer. Only accepting moves are shown; for any state in any NFA, if no move is defined for the current input symbol the parser halts and does not accept the input. Tables 4.1-4.4 show the parser for DTDs; only the functions for DTD, element declaration, content model, and model group are shown. Tables 4.3-4.4 can also be considered independently as a parser for model groups.

Table 4.1: DTD: parsing actions.

states	actions
1,2	local move
3/4	current state = 4 / 5 call element_declaration()
5	if next_symbol() = end of input if pushdown list is empty HALT: accept else current state = 5 call element_declaration()

NOTE 1: next_symbol() returns the next input symbol without advancing the input pointer.
NOTE 2: calling element_declaration() pushes (current_state,DTD) onto the pushdown list.

Table 4.2. Element_declaration: parsing actions.

states	Actions
1,2,6,8	make a local move
3	current_state = 4 call element_type()
4	if next_symbol() = ("O" or MINUS) current_state = 5 call omitted_tag() else if next_symbol() = ("CDATA" or "RCDATA" or "EMPTY") current state = 6 call declared_content() else if next_symbol() = GRPO current_state = 7 call model_group()
5	if next_symbol() = ("CDATA" or "RCDATA" or "EMPTY") current state = 6 call declared_content() else if next_state = GRPO current_state = 7 call model_group()
7	if next_symbol = (MINUS or PLUS) current_state = 8 call exceptions() else make a local move

Table 4.3. Model_group: parsing actions.

States	Actions
1,3,7,8,9	make a local move only
2,4,5,6	current_state = 3,7,8,9 respectively call content_token()
10	if next_symbol()=(OPT or PLUS or REP) make a local move. else if next_symbol() = end of input if the pushdown list is empty HALT and accept else return()
11,12,13	return()

NOTE 1: The Halt and accept action for state 10 is used only if the parser for model_group() is to be implemented independently from the DTD parser.

NOTE 2: Calling content_token() pushes model_group,current_state) onto the pushdown list. Return() pops the pushdown list.

Table 4.4. Content_token: parsing actions.

States	Actions
1	if next_symbol() = ("#PCDATA" or GI) make a local move else current_state = 4; call model_group()
3	if next_symbol()=(OPT or PLUS or REP) make a local move else return()
2,4,5,6,7	return()

NOTE 1: Calling model_group() pushes (content token, current_state) onto the pushdown list. Return() pops the pushdown list.

In this chapter, a system of finite automata is constructed from the system of syntax productions defining DTDs, and a parser for DTDs is constructed from this system of

finite automata. The component of the parser for DTDs that parses model groups can be implemented independently of the parser for DTDs. In the next chapter, the parser for model groups is used in algorithms to detect ambiguity in model groups. In Chapter 6, the parser for DTDs is used in algorithms to detect ambiguity caused by omitted tags.

5. Ambiguous Model Groups

In this chapter a method is shown for detecting model groups that are ambiguous content models under Clause 11.2.4.3. of the standard. The first section gives preliminary definitions for ambiguity in model groups. The second section contains algorithms: Algorithm 5.1 shows the construction of an indexed model group from a model group. Algorithm 5.2 describes the construction of an NFA from an indexed model group and shows methods for optimizing the number of states during the construction of the NFA and in the final NFA. Algorithm 5.3 uses the NFA constructed in Algorithm 5.2 to show a method for detecting model groups that are ambiguous without lookahead. The method of construction of the NFA in Algorithm 5.2 is generalized; a specific method is shown, but any construction may be used as long as the resulting NFA has certain properties required by the algorithm. This generalizes Algorithm 5.3 for detecting model groups that are ambiguous without lookahead, and thus is an improvement over existing methods [16].

5.1 Preliminary Definitions

Definition 5.1. Indexed model group [16]. An indexed model group is a model group in which each occurrence of a GI or #PCDATA token is assigned a unique index that distinguishes it from all other occurrences of the same symbol. Let $I=\{1,...,n\}$, for some $n \geq 1$ be the index set. Index the tokens from 1,...,n in order of occurrence from left to right. If m is a model group, the indexed model group derived from m is denoted by m' .

Σ denotes the input alphabet of m , the set of all GIs (element names) in m and #PCDATA if it occurs in m . Then m' may be considered as an expression (model group) over Σ' , where Σ' denotes the set of indexed elements of Σ that occur in m' [6]. Let $L(m)$ denote the language over Σ defined by m and let $L(m')$ denote the language over Σ' defined by m' .

Definition 5.3 clarifies the scope of Clause 11.2.4.3 by clearly restricting it to model groups; this is consistent with the assumptions in the literature [5,18,25]. The following paragraph is an informal restatement of Definition 5.3 that conforms closely to the style of Clause 11.2.4.3 and is defined completely in SGML terms.

A model group is ambiguous if and only if there is any instance of an element defined by the model group that contains an element or character string that satisfies more than one occurrence of a GI or #PCDATA token in the model group without looking ahead in the instance of the element. NOTE: The only instances of elements considered are those defined by the model group; the affects of exceptions, omitted tag minimization, and other features of DTDs are not considered.

Example 5.2. Ambiguous model group (without lookahead). Let $m=(X,B?,(A|B),C)$. Then $m'=(X_1,B_1?,(A_1|B_2),C_1)$.

The string XBAC is in $L(m)$ and the strings $X_1B_1A_1C_1$ and $X_1B_2C_1$ are both in $L(m')$. The B in XBAC satisfies B_1 in m' without lookahead because $X_1B_1A_1C_1$ is in $L(m')$ and $\text{symbol}(X_1B_1)=XB$. B also satisfies B_2 without lookahead because $X_1B_2C_1$ is in $L(m')$, and $\text{symbol}(X_1B_2)=XB$. Thus, m is ambiguous without lookahead.

THEOREM 5.1. *If a model group is not ambiguous without lookahead, then it is not ambiguous.*

PROOF. The following proof shows that if m is ambiguous, then m is ambiguous without lookahead. Then since P implies Q iff $\neg Q$ implies $\neg P$, the result holds. Suppose m is ambiguous. Then there is some u in $L(m)$ and v' and z' in $L(m')$ such that $\text{symbol}(v')=\text{symbol}(z')=u$. Because $\text{symbol}(v')=\text{symbol}(z')=u$, then $u=u_1,...,u_n$, $v'=v'_1,...,v'_n$, and $z'=z'_1,...,z'_n$, for $n \geq 0$. Then $v' \neq z'$ implies that $n \geq 1$, and that there must be some first occurrence of $k=1,...,n$ such that $v'_k \neq z'_k$.

Let $x'=v'_1,...,v'_{k-1}=z'_1,...,z'_{k-1}$. Because $\text{symbol}(v')=\text{symbol}(z')$, then $\text{symbol}(v'_k)=\text{symbol}(z'_k)$. Let $v'_k=a_i$ and $z'_k=a_j$. Let y' denote $v'_{k+1},...,v'_n$, and let w' denote $z'_{k+1},...,z'_n$.

Then $xay=u$ is in $L(m)$ and $v'=x'a_jy'$ and $z'=x'a_jw'$ are in $L(m')$ such that $\text{symbol}(x'a_j)=xa$, $\text{symbol}(x'a_j)=xa$, and $a_i \neq a_j$. Then, a satisfies a_i in m' without lookahead and a satisfies a_j in m' without lookahead. Thus if m is ambiguous it is ambiguous without lookahead and therefore, if m is not ambiguous without lookahead it is not ambiguous.

5.2 Algorithms

ALGORITHM 5.1. Construct an indexed model group m' from a model group m .

Input: A model group m .

Output: An indexed model group m' .

Method: Add translating actions to the parser for model groups that is shown in Figures 4.3-4.4 and Tables 4.3-4.4. These actions are shown in Tables 5.1 and 5.2.

Table 5.1. Model_group: indexing a model group.

States	Actions
1	expr = null make a local move
2/4/5/6	expr = expr,symbol current_state = 3 / 7 / 8 / 9 respectively s = content_token()
3,7,8,9	expr = expr,s make a local move
10	expr = expr,symbol if next_symbol() = (OPT or PLUS or REP) make a local move else if the pushdown list is empty HALT: output expr else return(expr)
11,12,13	expr = expr,symbol return(expr)

NOTE 1: Expr and s are strings and symbol is the current input symbol. All are local variables.

NOTE 2: The ',' operator denotes string concatenation

NOTE 3: The indexing is performed by content_token() and is shown in Table 5.2.

Table 5.2. Content_token: indexing a model group.

States	Actions
1	<pre> if next_symbol() = ("#PCDATA" or GI) make a local move else current_state = 4 s = model_group() </pre>
2	<pre> pos = lookup(symbol, names) if pos = null pos = insert(symbol, names) index[pos] = 1 else increment index[pos] expr = #PCDATA_iindex[pos] return(expr) </pre>
3	<pre> pos = lookup(symbol, names) if pos = null pos = insert(symbol, names) index[pos] = 1 else increment index[pos] expr = GI_iindex[pos] if next_symbol = (OPT or PLUS or REP) make a local move else return(expr) </pre>
4	<pre> return(s) </pre>
5,6,7	<pre> expr = expr,symbol return(expr) </pre>
<p>NOTE 1: Expr and s are strings, pos is a counter, symbol is the current input symbol, names[] is a list of symbols (GI's and "#PCDATA"), and index[pos] is the index of the symbol in names[pos]. Names[] and index[] are initialized to null at program start. Expr, s, pos, and symbol are local variables. The indexed symbols are returned as GI_i and #PCDATA_i for some i ≥ 1.</p> <p>NOTE 2: The ',' operator denotes string concatenation.</p> <p>NOTE 3: Lookup(symbol,names) returns the position of symbol in names[], and insert(symbol, names) inserts symbol into names[] and returns the position.</p>	

Example 5.3. An indexed model group. Consider the model group $m = (B?, (A \& B)^*)$. The indexed model group for m is $m' = (B_1?, (A_1 \& B_2)^*)$.

ALGORITHM 5.2: Construct an NFA from an indexed model group.

Input: An indexed model group m' .

Output: An NFA, M' , such that $L(M') = L(m')$.

Method: Construct any NFA M' with the following four properties:

1. $L(M')=L(m')$.
2. M' has no ϵ -transitions.
3. M' has no inaccessible states.
4. M' has no useless states.

One method for constructing M' is shown in Steps 1-3 below. In Step 1 an equivalent regular expression r' is derived from m' , such that $L(r')=L(m')$. In Step 2, an NFA M' , is constructed from r' such that $L(M')=L(r')=L(m')$. In Step 3, subalgorithms are given for removing the ϵ -transitions, inaccessible states, and useless states from M' .

Model groups containing '&' and '+' can result in a large number of states in M' . In Step 2, constructions are described for reducing the number of states during the construction of M' , and in Step 3 a method is described for optimizing the number of states in M' .

Step 1: Construct any regular expression r' , such that $L(r')=L(m')$. One method is to add the actions shown in steps a-c below to the parser for model groups.

- a. Modify the parser for model groups from Chapter 4 (Figures 4.3 and 4.4 and Tables 4.3 and 4.4) to recognize model groups containing symbols that are indexed GI and #PCDATA tokens. Recognition of these indexed tokens can be handled by the token recognizer for the parser. Thus, these modifications can be illustrated by replacing the arcs labeled with GI and PCDATA in Figure 4.4 with arcs labeled as GI_i and $PCDATA_i$, where $i \geq 1$ represents the index of the tokens.
- b. Annex H of the standard states that each model group has an equivalent regular expression, and it gives equivalent regular expressions for subexpressions of m containing '?' and '&'. The model group operators ',', '|', and '*' are equivalent to the regular expression operators concatenation, '|', and '*' respectively. Add

translating actions to the parser to replace each subexpression of m' with an equivalent regular expression as follows:

- OPT. For $A?$, '?' Annex H gives an equivalent regular expression as $A? = (A|e)$.
- AND. For $(A_1 \& A_2 \dots \& A_n)$, '&' is defined in the standard to mean "All must occur in any order". Annex H of the standard states the following: "AND groups reduce to an OR group of SEQ group permutations; for example: $(a \& b)$ is equivalent to the regular expression (or SGML model group:) $((a, b) | (b, a))$." Thus $(A_1 \& \dots \& A_n)$ is an or group, $(P_1 | P_2 \dots | P_n!)$, where each $P_i, i=1, \dots, n!$ is a sequence group permutation of $\{A_1, A_2, \dots, A_n\}$. To construct this group use any algorithm for generating the permutations of n items. One such algorithm is given in [14].
- SEQ. Annex H shows that the SEQ operator ',' is equivalent to regular expression concatenation. Thus $(A_1, A_2, A_3, \dots, A_n) = (\dots((A_1 A_2) A_3) \dots A_n)$ [3].
- OR. Annex H shows that the OR operator '|' is equivalent to the regular expression operator '|'. Thus $(A_1 | A_2 | A_3 \dots | A_n) = (\dots((A_1 | A_2) | A_3) \dots | A_n)$ [3].
- PLUS. The regular expression for A^+ is derived directly from the definition of '+' as "One or more occurrences." Thus $A^+ = AA^*$ [3].
- REP. The REP operator, '*' is defined in the standard to mean "Zero or more occurrences."

c. Clause 11.2.4.2 of the standard states that #PCDATA has an implied '*' indicator.

Thus, $\#PCDATA = \#PCDATA^*$.

Tables 5.3 and 5.4 shows the modifications to the parser to construct the equivalent regular expression r' for m' . It requires that the modifications to the token recognizer and to Figure 4.4 that are described in a. above have been made. The functions OPT(), PLUS(), AND(), SEQ(), OR(), and PCDATA implement the replacements defined in b. and c. above. Output from the modified parser is a regular expression r' such that $L(r') = L(m')$. If m' is the indexed model group from Example 5.3, $m' = (B_1?, (A_1 \& B_2)^*)$, then $r' = ((B_1 | e) ((A_1 B_2) | (B_2 A_1))^*)$.

Table 5.3. Model_group: constructing an equivalent regular expression for an indexed model group.

States	Actions
1	expr = null make a local move
2	expr = symbol current_state = 3 i=1 s _i = content_token()
3	make a local move
4	current_state = 7 i=i+1 s _i = content_token()
5	current_state = 8 s ₂ = content_token()
6	current_state = 9 s ₂ = content_token()
7	group_type = AND make a local move
8	s ₁ = "(" , s ₁ , " " , s ₂ , ")" group_type = OR make a local move
9	s ₁ = "(" , s ₁ s ₂ , ")" group_type = SEQ make a local move
10	if group_type = AND expr = expr, AND(s ₁ , ... s _i) else expr = expr, s ₁ expr = expr, symbol if next_symbol() = (OPT or PLUS or REP) make a local move else if the pushdown list is empty HALT: output expr else return(expr)
11	expr = "(" , expr , " e)" return(expr)
12	expr = expr, expr, "*" return(expr)
13	expr = expr, "*" return(expr)

NOTE 1: Expr is a string, s[] is a list of strings, symbol is the current input symbol, and group_type distinguishes AND groups from OR and SEQ groups.

NOTE 2: The ',' operator denotes string concatenation

NOTE 3: AND(s₁, ... s_i) returns an OR group of all possible permutations of the SEQ groups, (s₁, ... s_i).

Table 5.4. Content_token: constructing an equivalent regular expression for an indexed model group.

States	Actions
1	if next_symbol() = (#PCDATA _i or GI _i) make a local move else current_state = 4 expr = model_group()
2	expr = #PCDATA _i + "^*"
3	return(expr) expr = GI _i if next_symbol() = (OPT or PLUS or REP) make a local move else return(expr)
4	return(expr)
5	expr = "(" + expr + " e)"
6	return(expr) expr = expr + expr + "^*"
7	return(expr) expr = expr + "^*"
	return(expr)

NOTE 1: Expr is a local string variable. GI_i and #PCDATA_i, i ≥ 1, are the current symbols returned by the token recognizer.

NOTE 2: Setting current_state and calling model_group() pushes (content token,current_state) onto the pushdown list. Return() pops the pushdown list.

Step 2. Construct any NFA M' , such that $L(M')=L(r')$. One method is to use Thompson's construction of an NFA from a regular expression [2], which is illustrated in Chapter 2. For any input regular expression s , this method inductively constructs an NFA recognizing $L(s)$ by combining the component NFAs for the subexpressions of s . For example, Figure 5.1 shows an NFA recognizing $r'=((B_1 | e) ((A_1 B_2) | (B_2 A_1))^*)$ constructed using Thompson's construction.

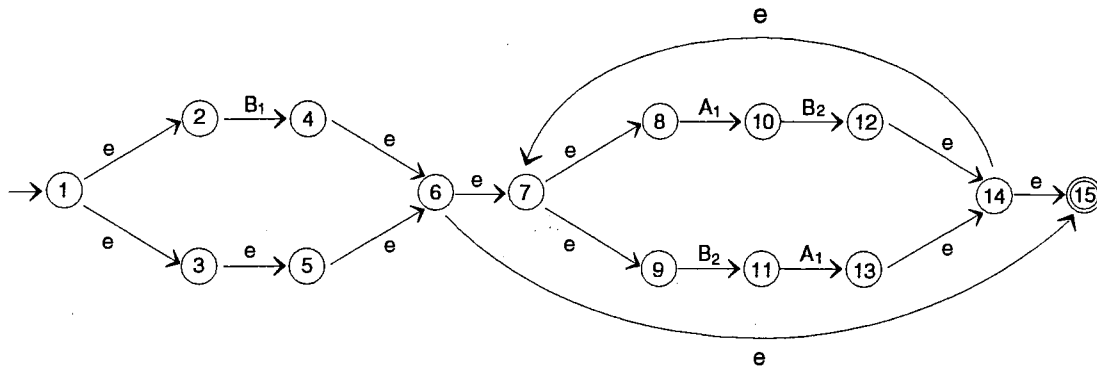


Figure 5.1. The NFA for r'

The requirements for correctness of Thompson's method are that each component NFA for a subexpression must 1. recognize the language defined by the subexpression, 2. have exactly one start state and one final state, and 3. have no arc entering the start state and no arc leaving the final state. Thus, any of the component NFAs can be replaced by any NFA that conforms to these three requirements.

One method for optimizing the space required by the resulting NFA M' , is described following Step 3. However, it does not reduce the peak storage requirements while constructing M' . In particular, using the equivalent regular expressions for AND and PLUS in Step 1 above will result in NFAs that are unnecessarily large when Thompson's construction is applied.

One method for solving this problem requires changes to Step 2 and Step 3. In Step 2 do not modify the parser to replace subexpressions A^+ and AND groups. This implies that Thompson's method must be extended in Step 3 as follows: 1. modify the parser for regular expressions to recognize extended regular expressions containing A^+ and AND groups, and 2. supply constructions for these subexpressions that construct component NFAs that conform to the three requirements for correctness listed above. Constructions that conform to these requirements are shown below.

Construction 1. PLUS(M_1). Let M_1 be an NFA for the subexpression m_1 of M_1 respectively. Construct the NFA $M(m_1\text{PLUS})$ as shown in Figure 5.2.

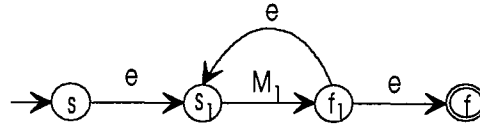


Figure 5.2. The NFA for $M(m_1\text{PLUS})$.

Figure 5.3 and 5.4 illustrate the different results using the two methods for constructing NFAs for A^+ . Figure 5.3 shows the NFA constructed for $(C|A)^+$ by Construction 1, and Figure 5.4 shows the NFA constructed by converting $(C|A)^+$ to $(C|A)(C|A)^*$ and then applying Thompson's construction.

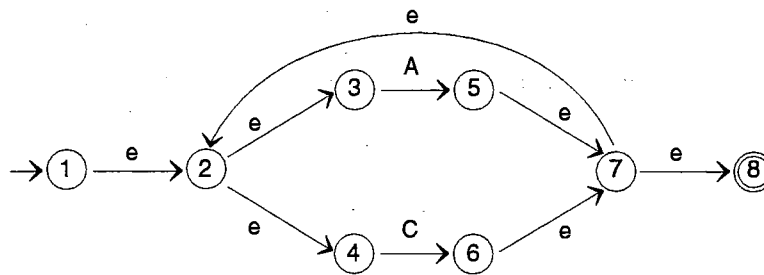


Figure 5.3. The NFA for $(C|A)^+$ by Construction 1.

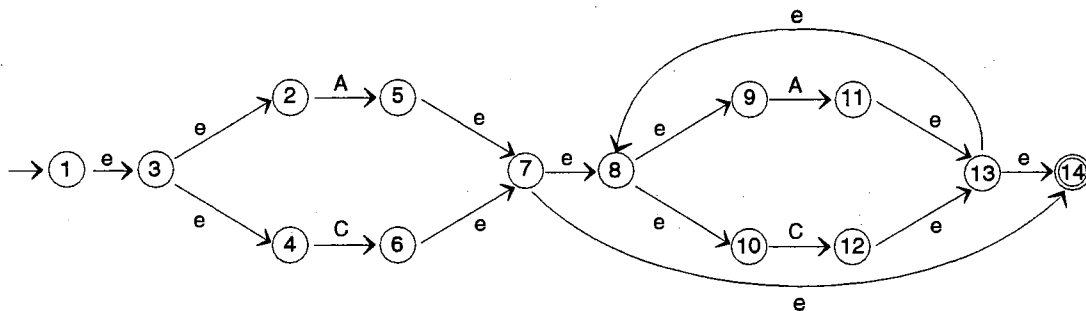


Figure 5.4. The NFA for $(C|A)^+$ by Thompson's construction.

When A in A^+ is large, Construction 1 will result in approximately one half as many states as converting the model group to a regular expression and then applying Thompson's method.

Construction 2. $\text{AND}(M_1, \dots, M_n)$. Let M_1, \dots, M_n be NFAs for the subexpressions m_1, \dots, m_n . The construction for AND cannot be performed incrementally as for SEQ and OR groups because AND is not an associative operator. To construct an NFA, $M(m_1 \text{ AND } \dots m_n)$ use the composite NFAs M_1, \dots, M_n for the subexpressions m_1, \dots, m_n respectively.

Create 2^n new states corresponding to all possible subsets of $\{M_1, \dots, M_n\}$: We call these the *primary states*, and each is denoted by the subset of $\{M_1, \dots, M_n\}$ that it represents. The primary state $p = \{M_j, \dots, M_k\}$ represents the state in which M_j, \dots, M_k have been recognized in the input (no ordering is implied). Thus, the start state of M is $\{\}$ and the final state is $\{M_1, \dots, M_n\}$. For each primary state $p = \{M_j, \dots, M_k\}$, construct a transition on M_i to each primary state $p' = \{M_j, \dots, M_k, M_i\}$, for all M_i , $i = 1 \dots n$, not in p . For each such transition, add a distinct copy of M_i to M . The copies of M_i are distinguished from each other by the state p from which they emanate. Merge the start state of each (M_i, p) into p and merge the final state of each (M_i, p) into p' as follows. To merge the start state q_0 of (M_i, p) into p , remove all transitions $\delta(q_0, a) \rightarrow q'$ from M and add $\delta(p, a) \rightarrow q'$ to M . The start state of (M_i, p) is now p . To merge the final state f of (M_i, p) into p' , for each state q in (M_i, p) , where q may be p , remove all transitions, $\delta(q, a) \rightarrow f$, from M and add $\delta(q, a) \rightarrow p'$ to M . In the resulting NFA, M , each state is unambiguously denoted by (p, M_i, q) , for q in M_i . Figure 5.5 shows the NFA for the model group, $(A \text{ AND } B \text{ AND } C)$ by Construction 2. The NFA constructed by converting $(A \text{ AND } B \text{ AND } C)$ to the equivalent regular expression $((ABC)|(ACB)|(BAC)|(BCA)|(CAB)|(CBA))$, and then applying Thompson's method has 34 states.

The NFAs from Constructions 1 and 2 conform to the three requirements for correctness for Thompson's method. Thus, after Step 2 is complete, regardless of the method used, the NFA M' is constructed such that $L(M') = L(r') = L(m')$.

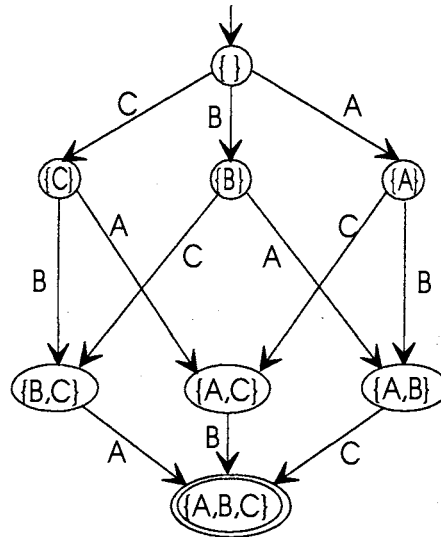


Figure 5.5. The NFA for (A AND B AND C) by Construction 2.

Step 3: Remove the e-transitions, inaccessible states, and useless states from M' using Algorithms 5.2.a-c below.

ALGORITHM 5.2.a. Removal of e-transitions from an NFA.

Input: An NFA, M .

Output: M with no e-transitions

Step 1: Derive a set of states Q' as follows: for each state q in Q , such that $\delta(r,a) \rightarrow q$, for some r and for $a \neq e$, add q to Q' , and add the start state of Q , q_0 , to Q' . Q' is the set of states in Q that can be reached by non e-transitions.

Step 2: For each state q in Q' , compute e-closure of q , the set of all states that can be reached from q on e-transitions only [2].

Step 3: For each state q in Q' and for each state p in e-closure of q , for all occurrences of $\delta(p,a) \rightarrow p'$ for some p' and $a \neq e$, add $(q,a) \rightarrow p'$ to δ .

Step 4: For each state q in Q' , for all p in e-closure of q such that p is in F , add q to F .

Step 5: Remove all e transitions from δ .

Algorithm 2 does not change the language, $L(M)$. $L(M)$ contains no new elements because every state reached by a transition added in Step 3 was reached on e-transitions

anyway, and all states added to F in Step 4 already had e -transitions to final states. No elements are removed from $L(M)$ because all e -transitions removed by Step 5 are replaced by direct transitions in Step 3.

Definition 5.4. Inaccessible states. In an NFA M , a state q is inaccessible iff $\{y \mid (q_0, y) \vdash^* (q, e) \text{ for } q_0 \text{ the start state of } M\} = \emptyset$.

ALGORITHM 5.2.b. Removal of inaccessible states from an NFA.

Input: An NFA, M .

Output: M with no inaccessible states.

Method: This algorithm is derived from a method illustrated in [3].

Step 1: Let $A_0 = \{q_0\}$ such that q_0 is the start state of M , and let $i=1$.

Step 2: Let $A_i = (A_{i-1} \cup A')$ where A' is derived as follows: for all states q in A_{i-1} and for all states p in Q of M , if $\delta(q, b) \rightarrow p$ for some b in Σ , then add p to A' .

Step 3: If $A_i \neq A_{i-1}$, then let $i=i+1$ and repeat Step 2, else let $A=A_i$. A is the set of states that is accessible from the start state q_0 : if q is in A , then there is some string x in Σ^* , such that $(q_0, x) \vdash^* (q, e)$. If there is no state f in F , then $L(M) = \emptyset$.

Step 4: Let $Q=A$ and for all states p in $Q-A$ and b in Σ , remove all $\delta(q, b) \rightarrow p$ from δ .

Algorithm 5.2.b does not change $L(M)$. Because no transitions or are added, no strings are added to $L(M)$. For all states that p that are removed, there is no x in Σ^* such that $(q_0, x) \vdash^* (p, e)$, and thus there can be no string xy in Σ^* such that $(q_0, xy) \vdash^* (p, y) \vdash^* (f, e)$ for some f in F . Figure 5.6 shows the NFA from Figure 5.1 after removing e -transitions.

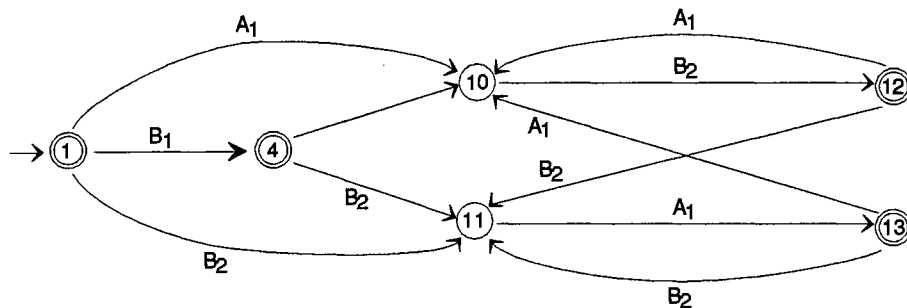


Figure 5.6. Removing e -transitions and inaccessible states.

Definition 5.5. Useless states. In an NFA M , a state q is useless iff $\{ y \mid (q, y) \vdash^* (f, e), f \text{ is in } F \} = \emptyset$.

ALGORITHM 5.2.c: Removing useless states from an NFA. This algorithm is adopted from an algorithm in [3] for removing useless symbols from a context free grammar.

Input: An NFA M .

Output: An NFA with no useless states, recognizing $L(M)$.

Method: First find the useless states, then remove them from Q of M .

Step 1: For each state q in Q of M , construct the set P of all states that can be reached from q : P is the set of all states p such that $(q, xy) \vdash^* (p, x)$ for some x in Σ^* . Construct sets $N_0, N_1 \dots N_i$ until set $N_i = P$ for some i .

- Let $N_0 =$ the empty set and $i=1$
- Let $N_i = \{ q' \mid \delta(q, a) \rightarrow q' \text{ for some } q \text{ in } N_{i-1}, a \text{ in } \Sigma \cup e \} \cup N_{i-1}$.
- If $N_i \neq N_{i-1}$, then set $i=i+1$ and go to Step 2, else $P=N_i$.
- If P contains no final states of M , then q is a useless state.

Step 2: For a useless state q , remove q from Q , and for any a in $(\Sigma \cup e)$ and any p in Q , remove all $\delta(q, a) \rightarrow p$ and all $\delta(p, a) \rightarrow q$ from δ of M .

The language $L(M)$ is unchanged by removing useless states. Removing any states and transitions cannot add any words to $L(M)$, thus there are no new words in $L(M)$. By Definition 5.4, if a state q was removed there was no sequence of moves $(q, y) \vdash^* (f, e)$ for some final state f of M . Thus, in the input NFA M , there was no sequence of moves $(q_0, xy) \vdash^* (q, y) \vdash^* (f, e)$. Thus, by removing q and all transitions into q and from q , there are no words that are removed from $L(M)$. Note that there are no useless states in the NFA in Figure 5.6.

Removing e -transitions, inaccessible states, and useless states does not affect the language recognized by M' . Therefore, $L(M')=L(r')=L(m')$ still holds. \square

Minimizing the number of states in the NFA M' is also a useful result. This can be accomplished simultaneously with the removal of inaccessible and useless states as

follows. Construct the canonical finite automata for M' [3]. This NFA has the least number of states of any NFA that recognizes $L(M')$. This NFA must have no inaccessible or useless states, because removal of these states does not affect the language recognized. Therefore, an inaccessible or useless state implies that there is an NFA that recognizes $L(M')$ but has fewer states, and this is a contradiction.

LEMMA 5.1. *If r' is a regular expression over $(\Sigma \cup \{I\})$ and x is in $L(r')$, then $\text{symbol}(x)$ is in $L(\text{symbol}(r'))$.*

PROOF. Let M' be the NFA constructed from r' by Thompson's construction. Then $L(M')=L(r')$. Replace all occurrences of indexed symbols x in r' with $\text{symbol}(x)$. The result is $\text{symbol}(r')$, a regular expression over Σ . Construct an NFA M from $\text{symbol}(r')$ using Thompson's construction, $L(M)=L(\text{symbol}(r'))$. M is exactly the same as M' except that each transition $\delta(q,x) \rightarrow p$ in M' is replaced by $\delta(q,\text{symbol}(x)) \rightarrow p$ in M .

Consider any string $x=x_1, \dots, x_n$, $n \geq 0$, in $L(r')$. There is a sequence of moves in M' of the form $(q_0, x_1, \dots, x_n) \vdash (q_1, x_2, \dots, x_n) \dots \vdash (q_n, e)$ for q_n in F , and this implies there is a sequence of moves in M of the form $(q_0, \text{symbol}(x_1), \dots, \text{symbol}(x_n)) \vdash (q_1, \text{symbol}(x_2), \dots, \text{symbol}(x_n)) \dots \vdash (q_n, e)$ for q_n in F .

Therefore, if x is in $L(r')$, $\text{symbol}(x)$ is in $L(\text{symbol}(r'))$.

LEMMA 5.2. *If m' is an indexed model group derived from m , then for all words x' in $L(m')$, $\text{symbol}(x')$ is in $L(m)$.*

PROOF.

1. Apply Step 1 of Algorithm 5.2 to m' to derive a regular expression r' , such that $L(m')=L(r')$.
2. Apply Step 1 of Algorithm 5.2 to m to derive a regular expression r , such that $L(m)=L(r)$.
3. By Definition 5.1, m and m' are identical except that each a_i in M' is replaced by $\text{symbol}(a_i)$ in M . The constructions of Step 1 of Algorithm 5.2 are independent of any particular symbols. Thus, the constructions in 1 and 2 above proceed identically,

except that each a_i in some subexpression of r' is replaced by $\text{symbol}(a_i)$ in the corresponding subexpression of r . Thus, $\text{symbol}(r')=r$.

Suppose x is in $L(m')$. Then by 1, x is in $L(r')$, and then by Lemma 3.1, $\text{symbol}(x)$ is in $L(\text{symbol}(r'))$. Thus, by 3, $\text{symbol}(x)$ is in $L(r)$, and therefore by 2, $\text{symbol}(x)$ is in $L(m)$.

ALGORITHM 5.4. Detecting model groups that are ambiguous without lookahead.

Input: a model group m .

Output: YES if m is ambiguous under Definition 5.3; else, output NO.

Method:

Step 1: Use Algorithm 5.1 to derive an indexed model group m' from m .

Step 2: Use Algorithm 5.2 to construct an NFA M' from m' such that $L(M')=L(m')$.

Algorithm 5.2 does not require any particular method of construction for the NFA; only that the four properties hold for the NFA.

Step 3: Traverse the NFA M' . If for any state q in Q' of M' , $\delta'(q, a_i) \rightarrow p$ and $\delta'(q, a_j) \rightarrow r$ for some p and r in Q' , such that $i \neq j$, then m is ambiguous and output YES; else, output NO.

PROOF.

Case 1: Suppose m is ambiguous without lookahead. By Definition 5.3 there is a string xay in $L(m)$ such that a in xay satisfies more than one indexed token in m' without lookahead. Then there is a string xay in $L(m)$ and strings $x'ajy'$ and $x'ajw'$ in $L(m')$ such that $\text{symbol}(x'aj)=xa$ and $\text{symbol}(x'aj)=xa$ and $i \neq j$. Because $x'ajy'$ and $x'ajw'$ are in $L(m')$ and $L(M')=L(m')$, $x'ajy'$ and $x'ajw'$ are in $L(M')$. Thus M' must make sequences of moves $(q_0, x'ajy') \vdash^* (q, ajy')$ and $(q_0, x'ajw') \vdash^* (q, ajw')$ for some q in Q' , such that $\delta'(q, a_i) \rightarrow p$ and $\delta'(q, a_j) \rightarrow r$ for some p and r in Q' and $i \neq j$. Thus, there is some state q in m' such that $\delta'(q, a_i) \rightarrow p$ and $\delta'(q, a_j) \rightarrow r$ for some p and r in Q' , such that $i \neq j$.

Case 2: Suppose there is some state q in M' such that $\delta'(q, a_i) \rightarrow p$ and $\delta'(q, a_j) \rightarrow r$ for some p and r in Q' , such that $i \neq j$. Because Algorithm 5.2 removes all inaccessible states from M' , the state q is accessible from q_0 . Thus there is some string x' such that M' makes

the sequence of moves $(q_0, x') \vdash^* (q, e)$ such that $\delta'(q, a_i) \rightarrow p$ and $\delta'(q, a_j) \rightarrow r$ for some p and r in Q' , such that $i \neq j$. Because there are no useless states in M' , and because $L(M') = L(m')$, then there are strings $x'a_jy'$ and $x'a_jw'$ in $L(M')$ such that $i \neq j$ (y' may equal w').

By Lemma 5.1, if z' is any string in $L(m')$, then $\text{symbol}(z')$ is in $L(m)$. This implies there are strings xay and xaw in $L(m)$ and strings $x'a_jy'$ and $x'a_jw'$ in $L(m')$ such that $\text{symbol}(x'a_j) = xa$ and $\text{symbol}(x'a_j) = xa$, and $i \neq j$. Then, there is a string xay in $L(m)$ and a string $x'a_jy'$ in $L(m')$ such that a in xay satisfies a_j in m' without lookahead and there is a string $x'a_jw'$ in $L(m')$ such that a in xay satisfies a_j in m' without lookahead. Thus, there is a string xay in $L(m)$ such that a in xay satisfies more than one indexed token in m' without lookahead. Therefore, m is ambiguous without lookahead.

Therefore, by 1 and 2 the algorithm is correct. \square

The model group of Example 5.3, $m = (B?, (A \& B)^*)$ is ambiguous. Example 5.3 shows the indexed model group derived from m , $m' = (B_1?, (A_1 \& B_2)^*)$. The equivalent regular expression for m' is $r' = (B_1 \mid e) ((A_1B_2) \mid (B_2A_1))^*$. Figure 5.1 shows the NFA M' constructed from r' using Thompson's method, where $L(M') = L(r') = L(m')$. Figure 5.6 shows this NFA after the removal of e -transitions, inaccessible states, and useless states.

The model group m is ambiguous, because the NFA M' in Figure 5.6 has outgoing transitions from state 1 on both B_1 and B_2 , representing two different indexed tokens in m' .

In this chapter, model groups that are ambiguous without lookahead are defined. The definition is consistent with Clause 11.2.4.3 of the standard and resolves ambiguities in the clause. An algorithm for detecting ambiguous model groups without lookahead is given that is an improvement over existing methods. In the next chapter a definition is given for ambiguity caused by omitting tags (as prohibited by Clause 7.3.1 of the standard), and an algorithm is shown for detecting this kind of ambiguity while parsing the DTD.

6. *Ambiguous DTDs*

Clause 7.3.1 of the standard prohibits ambiguity caused by omitted tags. However, the standard does not precisely define this kind of ambiguity, and it does not provide complete methods for detecting or preventing it. Clause 7.3.1 consists of a set of rules to be applied when creating a document instance; ambiguity is defined in terms of particular words (document instances) in a language, rather than in terms of the language specification (DTD). This is different than other definitions of ambiguity for languages [2,3,4] and for ambiguous content models in Clause 11.2.4.3 [13]. It places the responsibility for preventing ambiguity on data entry rather than on language design. This chapter shows a method for detecting this kind of ambiguity while parsing the DTD.

In the first section an algorithm is shown for constructing a system of regular expression R from a DTD, D , such that $L(R)=L(D)$, and constructing a system of automata S , from R , such that $L(S)=L(R)=L(D)$. In the second section a definition is given for ambiguity caused by omitted tags; ambiguous DTDs are defined rather than ambiguous document instances. Then an algorithm is shown for detecting this kind of ambiguity while parsing the DTD. The definition of ambiguity does not overlap with Definition 5.3 for ambiguous model groups; in particular, ambiguous model groups do not cause a DTD to be ambiguous by the definition given in this chapter. However, it is similar to Definition 5.3 in that it is based on a restricted lookahead in the input.

Only the high level syntax of document instances as defined by DTDs is considered; it is assumed that a token recognizer can distinguish between markup (begin and end tags) and text (`#PCDATA`, `CDATA`, and `RCDATA`). The methods in this chapter do not consider the optional SGML features, `SHORTTAG`, `DATATAG`, and `RANK`, and exceptions. Exceptions are considered in Chapter 7.

6.1. Systems of Automata Recognizing Document Instances

Definition 6.1: Explicit DTD. For each DTD, D , there is an equivalent explicit representation of D , \underline{D} . If D is any DTD, then \underline{D} is a set of productions of the form $A_i \rightarrow \alpha_i$, $i=1, \dots, n$, $n \geq 1$. \underline{D} has exactly one production for each element in the element type list of some element declaration in D . A_i is the name of the element and α_i is an expression defining the content of A_i . The start tag minimization, end tag minimization, and other implicit definitions of the element declaration for A_i are included explicitly in α_i . The expressions for α_i below are italicized to distinguish them.

- A. α_i is derived from the element declaration for A_i as shown in 1-3 below. In the expressions for α_i the begin and end are variables denoting their respective values.
1. If A_i has declared content of "EMPTY", then $\alpha_i = \langle A_i \rangle$. By Clause 7.3.1.1, all elements with declared content of "EMPTY" must have a start tag and by Clauses 7.3 and 11.2.2, they must not have an end tag. By Clause 11.2.3, "EMPTY" elements have no content other than the start tag.
 2. If A_i has declared content of "CDATA" or "RCDATA", then $\alpha_i = \langle A_i \rangle, \textit{data character}^*, \textit{end}$ where $\textit{end} = \langle /A_i \rangle$ if end tag minimization=MINUS and $\textit{end} = \langle /A_i \rangle ?$ if end tag minimization="O". By Clause 7.3.1.1, all elements with declared content must have a start tag, and by Clauses 9.1 and 9.2, "CDATA" and "RCDATA" define occurrences of data character*.
 3. If the content of A_i is defined by a model group, m , then $\alpha_i = \textit{begin}, m, \textit{end}$. If start tag minimization=MINUS, then $\textit{begin} = \langle A_i \rangle$, else $\textit{begin} = \langle A_i \rangle ?$. If end tag minimization=MINUS, then $\textit{end} = \langle /A_i \rangle$, else $\textit{end} = \langle /A_i \rangle ?$. For any occurrence of "PCDATA" in the model group, replace "#PCDATA" by data character*. By Clauses 7.6, 11.2.4, and 11.2.4.2, "PCDATA" is equivalent to "CDATA" and "RCDATA" except for details of token recognition distinguishing between data and markup.
- B. If an element declaration has an element type of $k > 1$ names, then by Clause 11.2.1:

1. Each name is an element name (GI).
2. The content of each element named in the element type is defined by the remainder of the element declaration.
3. An element can occur only once in an element type in a DTD.

Thus, by 2, for each element declaration with an element type of $k > 1$ elements, there is an equivalent DTD in which the element declaration is replaced by k declarations, one for each element, and by 3, there must be at most one element declaration per element name.

Therefore, by A and B, there is an explicit representation of D , \underline{D} , that is equivalent to D , and consists of a set of productions, $A_i \rightarrow \alpha_i$, $i=1, \dots, n$, $n \geq 1$, where some A_i is the DOCTYPE element. Algorithm 6.1 shows how to construct an explicit DTD, and Example 6.1 illustrates an explicit DTD.

Example 6.1: An explicit DTD. Let D be the DTD

```
<DOCTYPE      TOP  [
<!ELEMENT    TOP   - - (A, B?) >
<!ELEMENT    A     - O (B?) >
<!ELEMENT    B     - - (#PCDATA) >
```

Then the explicit DTD, \underline{D} , for D is

```
DOCTYPE = TOP
TOP  →  <TOP>, (A, B?), </TOP>
A    →  <A>, (B?), </A>?
B    →  <B>, (data character*), </B>
```

ALGORITHM 6.1. Constructing an explicit DTD.

Input: A DTD, D .

Output: An explicit representation of D , \underline{D} .

Method: Add translating actions to the parser for DTDs shown in Tables 4.1-4.4.

These actions are shown in Tables 6.1 and 6.2.

Table 6.1. DTD: constructing an explicit DTD

states	actions
1	n=0; local move
2	local move
3/4	DTD.doctype = symbol current_state = 4 /5 repectively productions = element_declaration() i=1; while productions[i] ≠ null n=n+1 dtd.productions[n] = productions[i] i=i+1
5	if next_symbol() = end of input if pushdown list is empty HALT: output DTD else current state = 5 productions = element_declaration() i=1; while productions[i] ≠ null n=n+1 dtd.productions[n] = productions[i] i=i+1

NOTE 1: DTD is an explicit DTD; DTD.doctype is the document type. DTD.productions[] and productions[] are lists of productions, where a production, (production.GI,production.expr). symbol is the current input symbol.

Table 6.2. Element declarations: constructing an explicit DTD.

states	actions
1,2,6,8	local move
3	content_type = null current_state = 4 type_list = element_type()
4	if next_symbol() = ("O" or MINUS) current_state = 5 tag_specs = omitted_tag_minimization()

Table 6.2. continued on next page.

Table 6.2 (cont.). Element declarations: constructing an explicit DTD.

```

-----
        else if next_symbol()=(CDATA or RCDATA or EMPTY)
            current_state = 6
            content_type = declared_content()
        else if next_symbol() = GRPO
            current_state = 7
            m = model_group()
5      if next_symbol()=(CDATA or RCDATA or EMPTY)
            current_state = 6
            content_type = declared_content()
        else if next_symbol() = GRPO
            current_state = 7
            m = model_group()
            m = replace_PCDATA(m)
            content_type = model_group
7      if next_symbol() = (MINUS or PLUS)
            current_state = 8
            exceptions()
        else
            local move
9      if tag_specs.beg = MINUS
            beg_expr = <GI>,
        else if tag_specs.beg = "O"
            beg_expr = <GI>?,
        if tag_specs.end = MINUS
            end_expr = </GI>,
        else if tag_specs.end = "O"
            end_expr = </GI>?,
        if content_type = "EMPTY"
            expr = <GI>
        else if content_type = (CDATA or RCDATA)
            expr = <GI>,data character*,end_expr
        else if content_type = model_group
            expr = beg_expr,data character*,end_expr
        i=1
        while type_list[i] ≠ null
            productions[i].GI = type_list[i]
            productions[i].expr = expr
            i=i+1
        return(productions[])
-----

```

NOTE 1: Element_type() returns a list of one or more element names, omitted_tag_minimization() returns an (tag_spec.beg,tag_spec.end), declared_content() returns CDATA, RCDATA, or EMPTY, model_group() returns a model group in which each "PCDATA" is replaced by data character*. Content_type is CDATA, RCDATA, EMPTY, or model group. Expr, beg_expr, and end_expr are strings.

NOTE 2: The model group SEQ operator ',' denotes string concatenation.

```

-----

```

ALGORITHM 6.2. Construct a system of regular expressions for a DTD.

Input: A DTD, D .

Output: A system of regular expressions, R , such that $L(R)=L(D)$.

Method: The method is to construct R such that R is equivalent to an explicit representation of D , \underline{D} , where each α_i in \underline{D} is replaced by an equivalent regular expression. Rather than develop a second algorithm to parse the α_i 's, R is constructed directly from the non explicit form of D by modifying Algorithm 6.1 as follows:

Step 1: In Table 6.2, let the variables *expr*, *beg_expr*, *end_expr*, *m*, and *production.expr* denote regular expressions, and let the model group SEQ operator ';' denote string concatenation [13].

Step 2: In state 4 of Table 6.2, let the call to *model_group()* return a regular expression for the model group. This construction is shown by Table 5.4. with the indexing removed (replace GI_i and $\#PCDATA_i$ with GI and *data character** respectively).

Step 3: In state 9 of Table 6.2, replace each occurrence of $\langle GI \rangle?$ and $\langle /GI \rangle?$ with $(\langle GI \rangle | e)$ and $(\langle /GI \rangle | e)$ respectively.

The result of these three steps is that the modified version of Algorithm 6.1 will output an explicit form of D , D' , where for each $A_i \rightarrow \alpha_i$, α_i is a regular expression. By Appendix H of the standard [13], model groups have equivalent regular expressions, and thus, it can be shown that $L(D')=L(\underline{D})=L(D)$.

Step 4: For each $A_i \rightarrow \alpha_i$ in D' , add $A_i \rightarrow \alpha_i$ to P of R , and add each A_i to N of R .

Step 5: Let the element, *DTD.doctype*, be N_0 of R , and let Σ of R be the set of all begin tags, end tags, and data characters in α_i , $i=1,...,n$.

Example 6.2. A system of regular expressions for a DTD. Let \underline{D} be the explicit DTD

DOCTYPE = TOP

TOP \rightarrow $\langle \text{TOP} \rangle$, (A, B?), $\langle / \text{TOP} \rangle$

A \rightarrow $\langle \text{A} \rangle$, (B?), $\langle / \text{A} \rangle?$

B \rightarrow $\langle \text{B} \rangle$, (data character*), $\langle / \text{B} \rangle$

Then the system of regular expressions, R , such that $L(R)=L(D)$ is:

1. $N = (TOP, A, B)$
2. $N_0 = TOP$
3. $\Sigma = \{<TOP>, </TOP>, <A>, , , , \text{data characters}\}$
4. $P =$

$TOP \rightarrow <TOP> (A, (B \mid e)) </TOP>$

$A \rightarrow <A> (B \mid e) (\mid e)$

$B \rightarrow (\text{data character}^*) $

ALGORITHM 6.3. Construct a system of finite automata S from a DTD D such that $L(S)=L(D)$.

Input: A DTD, D .

Output: A system of finite automata S , such that S has no useless states, $L(S)=L(D)$, and each NFA in M of S has no inaccessible or useless states.

Method:

Step 1: Construct a system of regular expressions, R , from D such that $L(R)=L(D)$ using Algorithm 6.2.

Step 2: From R constructed in Step 1, construct a system of finite automata S' , such that $L(S')=L(R)$ using Algorithm 3.1. Algorithm 3.1 removes the ϵ -transitions from each NFA in M of S' using Algorithm 5.2.a.

Step 3: Construct a system of finite automata S from S' using Algorithm 3.1, such that $L(S)=L(S')$. Remove the useless states from S using Algorithm 3.6. Algorithm 3.6 uses Algorithm 3.3 and Algorithm 3.5 to remove useless and inaccessible NFAs from S .

Example 6.3. A system of finite automata for a DTD. Consider the system of regular expressions, R , in Example 6.2 constructed from the DTD in Example 6.1 by Algorithm 6.2. Then the system of finite automata derived from R by Steps 2 and 3 of Algorithm 6.3 is:

1. $N = \{TOP, A, B\}$
2. $\Sigma = \{<TOP>, </TOP>, <A>, , , , \text{data characters}\}$
3. $M_0 = TOP$
4. $Q = \{(TOP,1),(TOP,2),(TOP,3),(TOP,4),(TOP,5),(A,1),(A,2),(A,3),(A,4), (B,1),(B,2),(B,3),(B,4)\}$
5. $\Gamma = Q$.
6. The NFAs of M are shown in Figures 6.1-6.3.

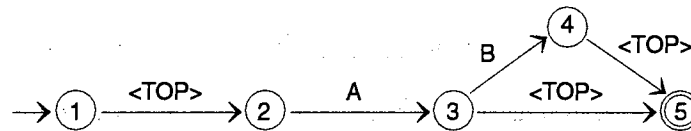


Figure 6.1. The NFA for TOP in Example 6.3.

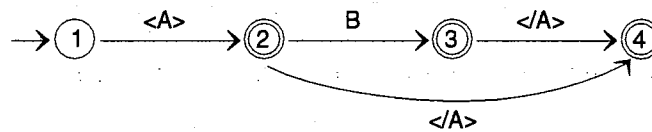


Figure 6.2. The NFA for A in Example 6.3.

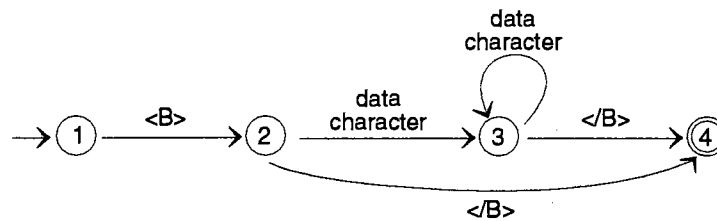


Figure 6.3. The NFA for B in Example 6.3.

6.2. Ambiguity Caused by Omitted Tags

Definition 6.2. Σ and Σ_O of D . If D is a DTD, then Σ of D is the set of all symbols that can occur in a document instance defined by D : the begin tags and end tags of all elements defined in some element declaration of D and the set of data characters defined for D . Σ_O is the set of all begin and end tags in Σ for which the omitted tag minimization is set to "O" in D , except for the end tags of elements with declared content of "EMPTY".

If R is a system of regular expressions constructed from D by Algorithm 6.2, and if S is a system of finite automata constructed from D by Algorithm 6.3, then Σ of D is the same as Σ of R and Σ of S . In the remainder of this chapter, Σ unambiguously refers to Σ of D , R , and S .

Definition 6.3. Completely tagged document instance. For a DTD, D , a document instance w is completely tagged iff w is also a document instance under D' , where D' is the DTD derived from D by setting the omitted tag minimization to minus for all start and end tags except the end tags of elements with declared content of "EMPTY".

Definition 6.4. Correspondence. For a DTD, D , if there is a document instance, xy , x and y in Σ^* , and a completely tagged document instance, $x'y'$, x' and y' in Σ^* , and x can be derived from x' by omitting zero or more occurrences of symbols in Σ_O from x' , then x corresponds to x' . Note that when x corresponds to x' and $y'=e$, x is a document instance and x' is a completely tagged document instance.

Definition 6.5. Ambiguous DTDs by omitted tags. A DTD is ambiguous by omitted tags iff there is any document instance w and completely tagged document instances w' and w'' , such that w corresponds to w' , w corresponds to w'' , and $w' \neq w''$.

Definition 6.6. Ambiguous DTDs by omitted tags without lookahead. A DTD is ambiguous by omitted tags without lookahead iff there is a document instance $w=Vby$, and completely tagged document instances, $w'=Vby'$ and $w''=Vby''$, such that:

1. either b is an input symbol in Σ or $by=by'=by''=e$,
2. V corresponds to V' and V corresponds to V'' , and

3. $V' \neq V''$.

An informal restatement of Definition 6.6 is as follows: a DTD is ambiguous by omitted tags without lookahead if and only if for any document instance defined by the DTD, for each symbol (a begin tag, end tag, or data character) or end of input encountered when parsing the document instance, without looking ahead in the document instance there is only one possible context for the symbol; that is, the prefix of the document instance up to the current symbol can correspond to at most one prefix (followed by the same symbol) of any completely tagged document instance.

Examples 6.4-6.10 illustrate Definition 6.6. Examples 6.4 and 6.5 are two examples that clearly illustrate the difference between Definitions 6.5 and 6.6. Note that in Definition 6.6 for the case when $by=by'=by''=e$, then D is ambiguous by Definition 6.5. That is, D is ambiguous whenever there is ambiguity without lookahead on end of input in some document instance.

THEOREM 6.1. *If a DTD D is not ambiguous by omitted tags without lookahead, then it is not ambiguous by omitted tags.*

PROOF. Suppose D is ambiguous by omitted tags. Then by Definition 6.5, there is a document instance w and completely tagged document instances w' and w'' , such that $w' \neq w''$ and w corresponds to w' and w corresponds to w'' . Let $by=by'=by''=e$, $V=w$, $V'=w'$, and $V''=w''$. Then directly by Definition 6.6, if D is ambiguous then D is ambiguous by omitted tags without lookahead. Then, because P implies Q iff $\neg Q$ implies $\neg P$, if D is not ambiguous by omitted tags without lookahead, it is not ambiguous. \square

Definition 6.7. Tree (adapted from [1]). A tree is a special case of a directed graph. A tree consists of a set of elements called nodes and a set of directed arcs that define a relation on the set of nodes; an arc is an ordered pair of nodes (v,w) . The arc (v,w) can be expressed as $v \rightarrow w$ or as "the arc from v to w ."

A relation parenthood is defined on the nodes in a tree as follows. A node b is the parent of a node c iff there is an arc from b to c . Conversely, a node c is the child of a

node b iff there is an arc from b to c . A node with no children is called a leaf node; all other nodes are called interior nodes. Trees are defined recursively by the following rules:

1. A single node by itself is a tree. This node is called the root of the tree.
2. If n is a leaf node in a tree T , and T_1, \dots, T_k are trees with roots n_1, \dots, n_k , $k \geq 1$, respectively, then a new tree T' can be constructed by adding arcs $(n, n_1), \dots, (n, n_k)$.

If n_1, \dots, n_k , $k \geq 1$, is a sequence of nodes in a tree such that for $i=1, \dots, k-1$, n_i is the parent of n_{i+1} , then there is path from n_1 to n_k . The length of the path from n_1 to n_k is $k-1$, the number of arcs on the path. There is a unique path from the root node to every node in the tree. A path from the root to a leaf node is called a full path. In this paper, a reference to a path is a reference to a full path unless noted otherwise.

Trees can have labels associated with each arc and/or each node; the labels can be values of any data type. Algorithm 6.4 constructs trees that have a label for each arc. If n_1, \dots, n_k , are nodes on a path then there is a label, b_i , on each arc $n_i \rightarrow n_{i+1}$, $i=1, \dots, k-1$. The sequence of labels on the path from n_1 to n_k is defined to be $b_1 \dots b_k$. The sequence of labels on a path may also be referred to as the label on the path.

Definition 6.8. *e-closure tree.* For any system of finite automata S that contains no local e -moves and for any state (M_i, q) in S , the e -closure tree for (M_i, q) is a tree in which the nodes of the tree represent states in Q of S and the root represents (M_i, q) . The paths of the tree represent all possible sequences of e -moves from the state (M_i, q) ; each state (node), (M_j, p) , has exactly one child for each state (M_k, r) that can be reached from (M_j, p) on one e -move (a push or a pop). The leaf nodes in an e -closure tree are the states that have no e -moves; some paths in an e -closure tree may be nonterminating (they have no leaf node).

An abbreviated e -closure tree is an e -closure tree in which any path is terminated by the second occurrence of some state on the path. Thus, because there are a finite number of states in Q of S , all paths in an abbreviated e -closure tree must terminate.

Definition 6.9. lookahead(1) tree (LA(1) tree). An LA(1) tree is an extension of an e-closure tree; each state (M_j, p) has exactly one child for each state (M_k, s) that can be reached by a single e-move. A node for state (M_j, p) also has one child for each b in Σ such that $\Delta((M_j, p), b, e) \rightarrow ((M_j, s), e)$ for some (M_j, s) in Q . The child is denoted by b . If (M_j, p) is a final state (M_0, f) for some f in F_0 , then (M_0, f) has one child representing end of input. The child for end of input is denoted by Z . Thus, the leaf nodes for an LA(1) tree are either an input symbol b in Σ or Z representing end of input. Some paths in an LA(1) tree may be nonterminating.

An abbreviated LA(1) tree, denoted ALA(1), is an LA(1) tree in which any state (M_j, p) that occurs on a path from the root for the second time has no children that are states. However, the node for (M_j, p) does have children denoting the next input symbol or end of input (leaf nodes). Note that for each LA(1) tree, T , there is exactly one ALA(1) tree, T' , and it may be true that $T=T'$.

Algorithm 6.4 constructs ALA(1) trees for systems of finite automata constructed by Algorithm 6.3. The arcs of these trees have labels that represent tags that are omitted in a move from the parent to the child. Examples 6.4-6.10 each show a system of finite automata S constructed by Algorithm 6.3 and the ALA(1) tree for one state in S .

ALGORITHM 6.4. Detecting DTDs that are ambiguous by omitted tags without lookahead.

Input: A DTD, D .

Output: If D is ambiguous by omitted tags without lookahead output YES, else output NO.

Method:

Step 1: Construct a system of finite automata S from D using Algorithm 6.3.

Step 2: For each state (M_i, p) in S such that for some q in Q_i and some a in Σ , $\delta(q, a) \rightarrow p$, or for $(M_i, p) = (M_0, q_0)$, construct an ALA(1) tree for (M_i, p) . The paths of the

tree represent the possible sequences of moves on the next input symbol b in Σ or on the first input symbol b when $(M_j, p) = (M_0, q_0)$.

The root of the tree represents (M_j, p) , the interior nodes represent states, and the leaf nodes represent the next input symbol, end of input, or a state that has already occurred on the path and which has no transitions on input symbols.

Step 3: For each interior node, (representing some state (M_j, r)), construct the children of the node, which represent the possible moves from (M_j, r) as described in cases a-e below.

Each arc in the tree from (M_j, r) to an input symbol b in Σ , to Z for end of input, or to a state (M_k, s) in Q is assigned a label: a begin tag $\langle N_j \rangle$, an end tag $\langle /N_j \rangle$, $\langle N_j \rangle \langle /N_j \rangle$, or ϵ representing no label. The sequence of labels on a path are the begin and end tags that are omitted in the sequence of moves represented by the path. The labels for arcs are assigned as described in cases a-e as follows:

a. Local Moves. For each transition from r of M_j on some input symbol b to a state t in Q_j , $\delta(r, b) \rightarrow t$, create a child node b for (M_j, r) .

If $(M_j, r) = (M_j, q_0)$ where q_0 is the start state of M_j and if b is in any symbol in Σ except $\langle N_j \rangle$, then add $\langle N_j \rangle$ to the arc from (M_j, q_0) to b . For any α in Γ^* , y in Σ^* , and $b \neq \langle N_j \rangle$, in the move $((M_j, q_0), by, \alpha) \vdash ((M_j, t), y, \alpha)$ the begin tag $\langle N_j \rangle$ is omitted.

b. Pushdown moves (pushes). For each transition from r of M_j on some N_k in N to some t in Q_j , $\delta(r, N_k) \rightarrow t$, create a child (M_k, q_0) for (M_j, r) .

If $(M_j, r) = (M_j, q_0)$ where q_0 is the start state of M_j , add the label $\langle N_j \rangle$ to the arc from (M_j, q_0) to (M_k, q_0) . For any α in Γ^* , any y in Σ^* , and $b \neq \langle N_j \rangle$, in the move $((M_j, q_0), y, \alpha) \vdash ((M_k, q_0), y, (M_j, t)\alpha)$ the begin tag $\langle N_j \rangle$ is omitted.

c. Pushdown moves (pops). If $r = f$ is a final state in F_j , for all M_k in M and t and q in Q_k such that there is a transition from q to t on N_j , $\delta(q, N_j) \rightarrow t$, create a child (M_k, t) for (M_j, f) .

If f is in F_j and f is not entered by a transition on $\langle N_j \rangle$, then add $\langle N_j \rangle$ to the arc from (M_j, f) to (M_k, t) . For any α in Γ^* and y in Σ^* , in the move $((M_j, f), y, (M_k, t)\alpha) \vdash ((M_k, t), y, \alpha)$ the end tag $\langle N_j \rangle$ is omitted.

- d. Pushdown moves (pops) from start states. This is a special case of case c, and should be performed instead of case c whenever $(M_j, r) = (M_j, q_0)$, such that q_0 is the start state of (M_j) and q_0 is also in F_j .

Add $\langle N_j \rangle \langle N_j \rangle$ to the arc from (M_j, q_0) to (M_k, t) . In the move $((M_j, q_0), y, (M_k, t)\alpha) \vdash ((M_k, t), y, \alpha)$ both $\langle N_j \rangle$ and $\langle N_j \rangle$ are omitted. This move can only occur when the omitted tag minimization for the start and end tag are "O" and when the content of the element N_j is optional as defined by the model group or declared content for N_j . In this case e is in $L(M_j)$.

- e. Accepting moves. If $(M_j, r) = (M_0, f)$ for some f in F_0 , create a child Z for (M_j, r) where Z denotes end of input.

If f is not entered by a transition on $\langle N_0 \rangle$, add $\langle N_0 \rangle$ to the arc from (M_0, f) to Z .

In this accepting move, the end tag $\langle N_0 \rangle$ is omitted.

For cases a-e, if (M_j, r) has already occurred on the path from the root, create children for (M_j, r) for local moves and accepting moves (Steps a and e), but do not create children for pushdown moves (Steps b, c, and d). Thus, the tree is an ALA(1) tree.

Step 4: All labels assigned to arcs in Step 3 are strings in Σ_O^* (the set of all begin and end tags of elements in D for which the omitted tag minimization is set to "O"). For each tree constructed by Steps 2-3, concatenate the labels on each path from the root to a leaf node. The result for each path is a string V in Σ_O^* . V is called the sequence of labels on the path. If there are any two paths in the tree, from the root to two leaf nodes for b in Σ or from the root to two leaf nodes Z such that the sequences of labels on the two paths are not equal, then halt and output YES. If this condition is not met for any tree constructed by Steps 2-3, then output NO.

Note that the tree is checked for two paths to leaf nodes Z with different sequences of labels on the path. However, by the results of Lemma 6.1, Theorem 6.2, and Lemma 6.6, detecting two paths to Z with different sequences of labels is not necessary for detecting ambiguity. Thus, Step 3.e may be omitted in Algorithm 6.4. However, by Definition 6.6, a DTD is ambiguous without lookahead if there are two different sequences of moves on end of input, and thus Step 3.e is kept in Algorithm 6.4 to illustrate completeness. Examples 6.4-6.10 show the tree construction by Algorithm 6.4. For each DTD, D , only the tree of one state in S is shown (to illustrate ambiguity).

Example 6.4. This example shows that when an element can satisfy two element tokens in different model groups, the DTD is ambiguous. Consider the DTD, D :

```
<!DOCTYPE A [  
  <!ELEMENT A  - -  (B | C)  >  
  <!ELEMENT B  O -  (C)  >  
  <!ELEMENT C  - O  "EMPTY"  > ]
```

Then a system of finite automata recognizing $L(D)$ is

1. $N = \{A, B, C\}$
2. $\Sigma = \{<A>, , , , <C>\}$
3. $N_0 = A$
4. $Q = \{(A,1), (A,2), (A,3), (A,4), (A,5), (B,1), (B,2), (B,3), (B,4), (C,1), (C,2)\}$
5. $\Gamma = Q$.
6. Δ is defined by M .
7. The NFAs in M are shown in Figures 6.4-6.6.

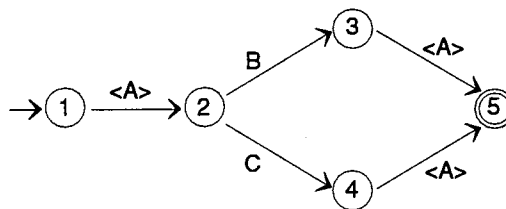


Figure 6.4. The NFA for A in Example 6.4.

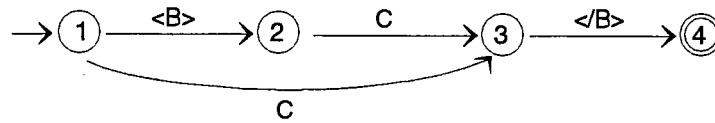


Figure 6.5. The NFA for B in Example 6.4.

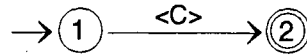


Figure 6.6. The NFA for C in Example 6.4.

The ALA(1) tree constructed for (A,2) by Algorithm 6.4 is shown in Figure 6.7. D is ambiguous because there are two paths in the tree to leaf nodes for <C> with different sequences of labels (and e).

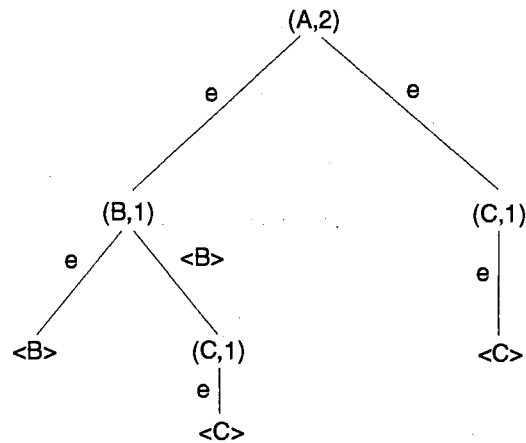


Figure 6.7. The ALA(1) tree for (A,2) in Example 6.4.

Table 6.3 shows a document instance w for D that corresponds to completely tagged document instances w' and w'' .

Table 6.3. Correspondence for Example 6.4.

V=<A>, V'=<A>, V"=<A>, b=<C>, Y=, Y'=, Y"=.		

W	W'	W''

<A>	<A>	<A>
<C>	<C>	
		<C>

Example 6.5. This example illustrates ambiguity caused by omitted begin tags.

Consider the DTD, D:

```
<!DOCTYPE A [
  <!ELEMENT A - - (B | C) >
  <!ELEMENT B O - (#PCDATA) >
  <!ELEMENT C O - (#PCDATA) > ]
```

Then a system of finite automata recognizing D is:

1. $N = \{A, B, C\}$
2. $\Sigma = \{\langle A \rangle, \langle /A \rangle, \langle B \rangle, \langle /B \rangle, \langle C \rangle, \langle /C \rangle, \text{data characters}\}$
3. $N_0 = A$
4. $Q = \{(A,1), (A,2), (A,3), (A,4), (A,5), (B,1), (B,2), (B,3), (B,4)\}$
5. $\Gamma = Q$.
6. Δ is defined by M.
7. The NFAs in M are shown in Figures 6.8-6.10.

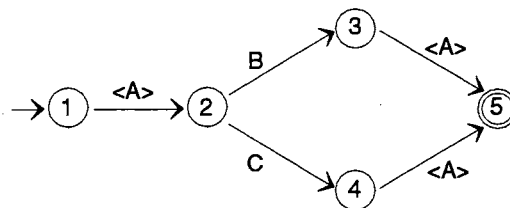


Figure 6.8. The NFA for A in Example 6.5.

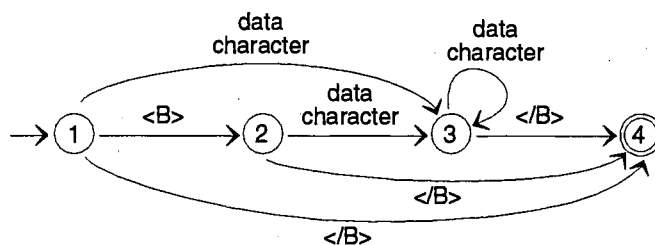


Figure 6.9. The NFA for B in Example 6.5.

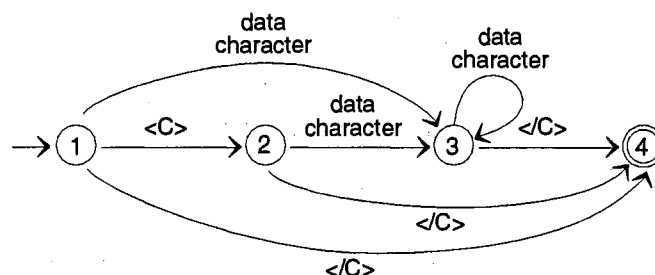


Figure 6.10. The NFA for C in Example 6.5.

The ALA(1) tree constructed for (A,2) by Algorithm 6.4 is shown in Figure 6.11. D is ambiguous because there are two paths in the tree to leaf nodes for data character, and the paths have different sequences of labels (and <C>).

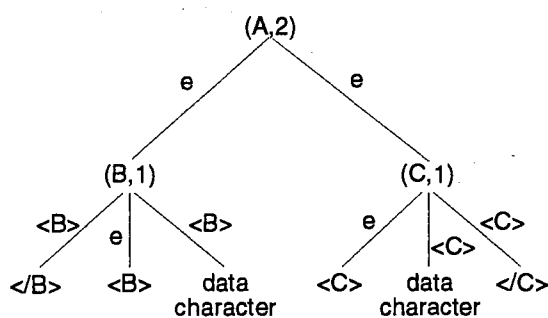


Figure 6.11. The ALA(1) tree for (A,2) in Example 6.5.

Table 6.4 shows a document instance w for D that corresponds to completely tagged document instances w' and w'' .

Table 6.4. Correspondence for Example 6.5.

V=<A>, V'=<A>, V"=<A><C>, b=data character, y=, y'=, y"=</C>.		

w	w'	w''

<A>	<A>	<A>
data character		<C>
	data character	data character
		</C>

Example 6.6. This example illustrates ambiguity caused by omitted end tags. Consider the DTD, D:

```
<!DOCTYPE A [
  <!ELEMENT A - - (B, #PCDATA) >
  <!ELEMENT B - O (#PCDATA) > ]
```

Then a system of finite automata recognizing D is:

1. $N = \{A, B\}$
2. $\Sigma = \{\langle A \rangle, \langle /A \rangle, \langle B \rangle, \langle /B \rangle, \text{data character}\}$
3. $N_0 = A$
4. $Q = \{(A,1), (A,2), (A,3), (A,4), (A,5), (B,1), (B,2), (B,3), (B,4)\}$
5. $\Gamma = Q$.
6. Δ is defined by M.
7. The NFAs in M are shown in Figures 6.12 and 6.13.

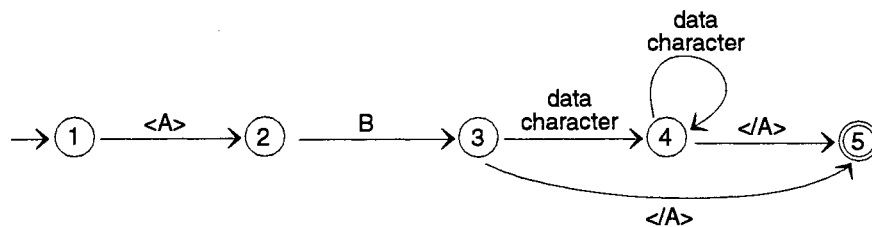


Figure 6.12. The NFA for A in Example 6.6.

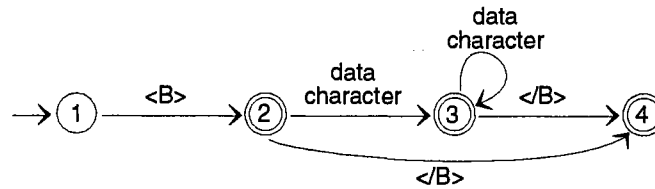


Figure 6.13. The NFA for B in Example 6.6.

The ALA(1) tree constructed for (B,2) by Algorithm 6.4 is shown in Figure 6.14. D is ambiguous because there are two paths to leaf nodes for data character with different label sequences (and e).

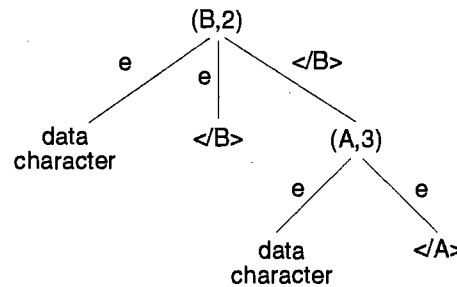


Figure 6.14. The ALA(1) tree for (B,2) in Example 6.6.

Table 6.5 shows a document instance w for D that corresponds to completely tagged document instances w' and w'' .

Table 6.5. Correspondence for Example 6.6.

$V = \langle A \rangle \langle B \rangle$, $V' = \langle A \rangle \langle B \rangle$, $V'' = \langle A \rangle \langle B \rangle \langle /B \rangle$,		
$b = \text{data character}$, $y = \langle /A \rangle$, $y' = \langle /B \rangle \langle /A \rangle$, $y'' = \langle /A \rangle$.		

w	w'	w''

$\langle A \rangle$	$\langle A \rangle$	$\langle A \rangle$
$\langle B \rangle$	$\langle B \rangle$	$\langle B \rangle$
data character	data character	$\langle /B \rangle$
$\langle /A \rangle$	$\langle /B \rangle$	data character
	$\langle /A \rangle$	$\langle /A \rangle$

The standard specifies that any occurrence of #PCDATA (data character*) satisfies only one token. However, this is a disambiguating rule applied by a parser. This example shows that this kind of ambiguity can be detected statically, while parsing the DTD.

Example 6.7. Ambiguity caused by elements that are optional and that can be empty.

Consider the DTD, D:

```
<!DOCTYPE A [  
  <!ELEMENT A - - (B?, C) >  
  <!ELEMENT B O O (#PCDATA) >  
  <!ELEMENT C - O "EMPTY" > ]
```

Then a system of finite automata recognizing $L(D)$ is:

1. $N = \{A, B\}$
2. $\Sigma = \{\langle A \rangle, \langle /A \rangle, \langle B \rangle, \langle /B \rangle, \langle C \rangle, \langle /C \rangle, \text{data characters}\}$
3. $N_0 = A$
4. $Q = \{(A,1), (A,2), (A,3), (A,4), (A,5), (B,1), (B,2), (B,3), (B,4), (C,1), (C,2)\}$
5. $\Gamma = Q$.
6. Δ is defined by M.
7. The NFAs in M are shown in Figures 6.15-6.17.

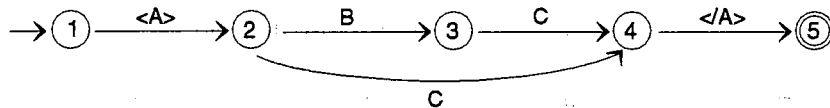


Figure 6.15. The NFA for A in Example 6.7.

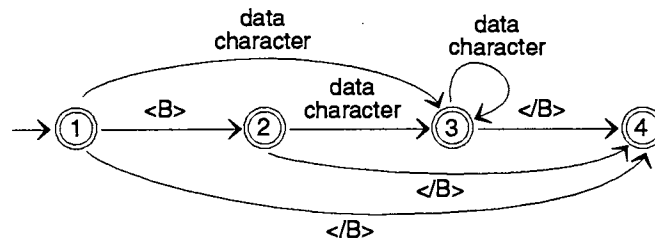


Figure 6.16. The NFA for B in Example 6.7.

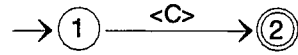


Figure 6.17. The NFA for C in Example 6.7.

The ALA(1) tree constructed for (A,2) by Algorithm 6.4 is shown in Figure 6.18. D is ambiguous because there are two occurrences of paths to leaf nodes for <C> in the tree and each has a different sequence of labels, (, and e).

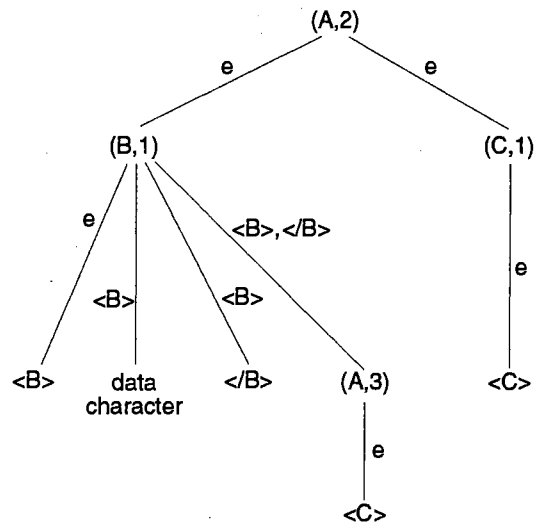


Figure 6.18. The ALA(1) tree for (A,2) in Example 6.7.

Table 6.6 shows a document instance w for D that corresponds to completely tagged document instances w' and w'' .

Table 6.6. Correspondence for Example 6.7.

 $V = \langle A \rangle$, $V' = \langle A \rangle$, $V'' = \langle A \rangle \langle B \rangle \langle /B \rangle$, $b = \langle C \rangle$,
 $Y = \langle /A \rangle$, $Y' = \langle /A \rangle$, $Y'' = \langle /A \rangle$.

w	w'	w''
$\langle A \rangle$	$\langle A \rangle$	$\langle A \rangle$
$\langle C \rangle$	$\langle C \rangle$	$\langle B \rangle$
$\langle /A \rangle$	$\langle /A \rangle$	$\langle /B \rangle$
		$\langle C \rangle$
		$\langle /A \rangle$

Example 6.8. Ambiguity caused by omitted begin tags and recursion in the DTD.

Consider the DTD, D:

```
<!DOCTYPE A [
  <!ELEMENT A  O - (A | B) >
  <!ELEMENT B  - O "EMPTY" > ]
```

Then a system of finite automata recognizing $L(D)$ is:

1. $N = \{A, B\}$
2. $\Sigma = \{<A>, , \}$
3. $N_0 = A$
4. $Q = \{(A,1), (A,2), (A,3), (A,4), (A,5), (B,1), (B,2)\}$
5. $\Gamma = Q$.
6. Δ is defined by M.
7. The NFAs of M are shown in Figures 6.19-6.20.

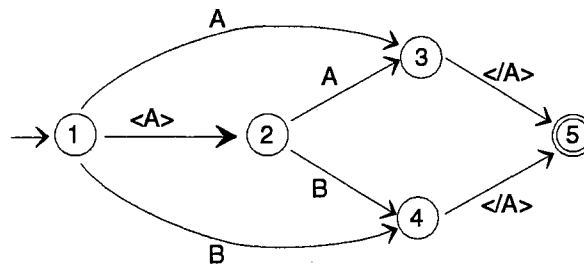


Figure 6.19. The NFA for A in Example 6.8.

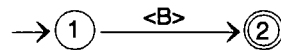


Figure 6.20. The NFA for B in Example 6.8.

The ALA(1) tree constructed for (A,1) by Algorithm 6.4 is shown in Figure 6.21. D is ambiguous because there are two occurrences of paths to leaf nodes for $<A>$, and each path has a different sequence of labels ($<A>$ and e). Recursion is not inherently ambiguous. This is illustrated by setting the omitted tag minimization to minus for A in D.

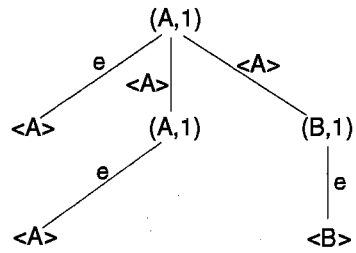


Figure 6.21. The ALA(1) tree for (A,1) in Example 6.8.

Table 6.7 shows a document instance w for D that corresponds to completely tagged document instances w' and w'' .

Table 6.7. Correspondence for Example 6.8.

$V=<A>, \quad V'=<A>, \quad V''=<A><A>, \quad b=,$		
$y=, \quad y'=, \quad y''=.$		

w	w'	w''

$<A>$	$<A>$	$<A>$
$$	$$	$<A>$
$$	$$	$$
		$$
		$$

Example 6.9. Ambiguity caused by omitted end tags and recursion in the DTD.

Consider the DTD, D :

```
<!DOCTYPE A [
  <!ELEMENT A - O (A | B) >
  <!ELEMENT B - O "EMPTY" > ]
```

Then a system of finite automata recognizing $L(D)$ is:

1. $N = \{A, B\}$
2. $\Sigma = \{<A>, , \}$
3. $N_0 = A$
4. $Q = \{(A,1), (A,2), (A,3), (A,4), (A,5), (B,1), (B,2)\}$
5. $\Gamma = Q$.

6. Δ is defined by M .
7. The NFAs in M are shown in Figures 6.22-6.23.

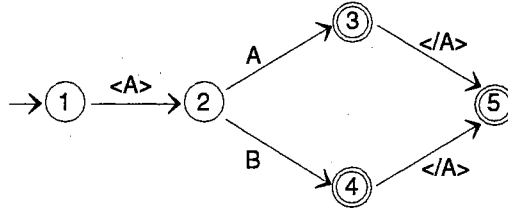


Figure 6.22. The NFA for A in Example 6.9.

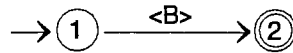


Figure 6.23. The NFA for B in Example 6.9.

The ALA(1) tree constructed for $(A,3)$ by Algorithm 6.4 is shown in Figure 6.24. D is ambiguous because there are two occurrences of paths to leaf nodes for $\langle /A \rangle$ in the tree, and each path has a different sequence of labels, ($\langle /A \rangle$ and e). Note that there are also two different sequences of labels on the paths to Z . Lemma 6.6 shows that if there is a tree with two paths to leaf nodes for Z with different sequences of labels, there will always be some tree constructed for S that has two paths to leaf nodes for some b in Σ , with different sequences of labels on the paths. Also note that the node for $(A,3)$ terminates because it is the second occurrence of a state on a path.

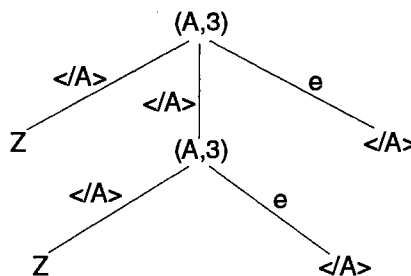


Figure 6.24. The ALA(1) tree for $(A,3)$ in Example 6.9.

Table 6.8 shows a document instance w for D that corresponds to completely tagged document instances w' and w'' .

Table 6.8. Correspondence for Example 6.9.

 $V = \langle A \rangle \langle A \rangle \langle B \rangle$, $V' = \langle A \rangle \langle A \rangle \langle B \rangle$, $V'' = \langle A \rangle \langle A \rangle \langle B \rangle \langle /A \rangle$,
 $b = \langle /A \rangle$, $y = e$, $y' = \langle /A \rangle$, $y'' = e$.

w	w'	w''
$\langle A \rangle$	$\langle A \rangle$	$\langle A \rangle$
$\langle A \rangle$	$\langle A \rangle$	$\langle A \rangle$
$\langle B \rangle$	$\langle B \rangle$	$\langle B \rangle$
$\langle /A \rangle$	$\langle /A \rangle$	$\langle /A \rangle$
	$\langle /A \rangle$	$\langle /A \rangle$

 Note that $w' = w''$. When the $b = \langle /A \rangle$ in w is encountered, it cannot be determined without lookahead whether it is the end of the inner $\langle A \rangle$ or the end of the outer $\langle A \rangle$.

Example 6.10. Ambiguous model groups. The element declaration for E is an example of an ambiguous content model from Clause 11.2.4.3 and is an ambiguous model group without lookahead by Definition 3.2. Consider the DTD, D :

```
<!DOCTYPE E [
  <!ELEMENT E  O - ((A, B?), B) >
  <!ELEMENT A  - O "EMPTY" >
  <!ELEMENT B  - O "EMPTY" > ]
```

Then a system of finite automata recognizing $L(D)$ is:

1. $N = \{E, A, B\}$
2. $\Sigma = \{\langle E \rangle, \langle /E \rangle, \langle A \rangle, \langle B \rangle\}$
3. $N_0 = E$
4. $Q = \{(E, 1), (E, 2), (E, 3), (E, 4), (E, 5), (E, 6), (B, 1), (B, 2), (A, 1), (A, 2)\}$
5. $\Gamma = Q$.
6. Δ is defined by M .
7. The NFAs in M are shown in Figures 6.25-6.27.

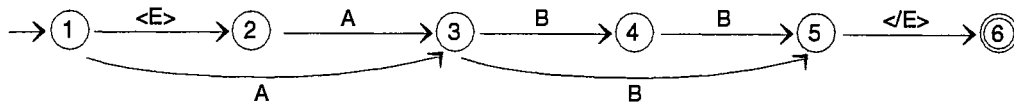


Figure 6.25. The NFA for E in Example 6.10

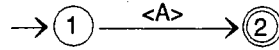


Figure 6.26. The NFA for A in Example 6.10.

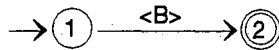


Figure 6.27. The NFA for B in Example 6.10.

The ALA(1) tree constructed for (E,3) by Algorithm 6.4 is shown in Figure 6.28.

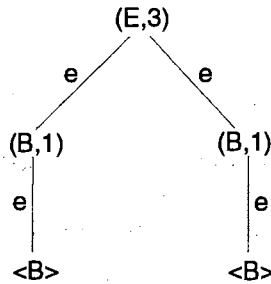


Figure 6.28. The ALA(1) tree for (E,3) in Example 6.10.

There are two paths to leaf nodes for . However, the paths have the same sequence of labels, e. The trees for the other states of S also do not satisfy the requirements for ambiguity. Thus D is not ambiguous.

Lemma 6.1 supports Theorem 6.2 that follows. It is used to simplify the proof of Theorem 6.3, and it can be used to simplify Algorithm 6.4. It shows that for any DTD, that if there is a document instance with an ambiguity on end of input, then there is also some document instance with an ambiguity on an input symbol in Σ .

LEMMA 6.1. *For a DTD, D , If there is a document instance V and completely tagged document instances, V' and V'' , such that V corresponds to V' and to V'' and $V' \neq V''$, then there is a document instance $w = Vb$ and completely tagged document instances, $w' = V'b$ and $w'' = V''b$, such that V corresponds to V' and to V'' , $V' \neq V''$, and b is an input symbol in Σ .*

PROOF. Let V be a document instance and V' and V'' be completely tagged document instances such that V corresponds to V' and to V'' , and $V' \neq V''$. Let $V = v_1, \dots, v_n$, $n \geq 0$ and let $V' = v'_1, \dots, v'_k$ and $V'' = v''_1, \dots, v''_m$. Because V' and V'' are completely tagged, $k \geq 2$ and $m \geq 2$, and $v'_k = v''_m = \text{</DOCTYPE>}$.

Case A: Suppose $v_n \neq \text{</DOCTYPE>}$. Let $b = \text{</DOCTYPE>}$. Then because V can be derived from V' and from V'' , Vb is a document instance that can be derived from V' and from V'' by not omitting $b = v'_k = v''_m$. Let $V' = v'_1, \dots, v'_{k-1}$ and $V'' = v''_1, \dots, v''_{m-1}$. Thus, $w = Vb$ is a document instance, $w' = V'b$ and $w'' = V''b$ are completely tagged document instances, V corresponds to V' and to V'' , and $V' \neq V''$.

Case B: Suppose $v_n = \text{</DOCTYPE>}$. Let $V = v_1, \dots, v_{n-1}$ and $b = \text{</DOCTYPE>}$. Then $Vb = V$ is in $L(D)$ and Vb corresponds to V' and to V'' . Let $V' = v'_1, \dots, v'_{k-1}$ and $V'' = v''_1, \dots, v''_{m-1}$. Then because $v'_k = v''_m = \text{</DOCTYPE>}$ and $V' \neq V''$, then $V' \neq V''$. Thus, $w = Vb$ is a document instance, $w' = V'b$ and $w'' = V''b$ are completely tagged document instances, and V corresponds to V' and to V'' . \square

The results of Lemma 6.1 and Theorem 6.1 are similar but not identical. Theorem 6.1 shows that if a DTD D is ambiguous, then by Definition 6.6 (for the case of $\text{by} = \text{by}' = \text{by}'' = e$), D is ambiguous without lookahead. Lemma 6.1 shows that if D is ambiguous, then there is a document instance in $L(D)$ in which there is an ambiguity on an input symbol b in Σ . Lemma 6.1 is used in Theorem 6.2 that follows to show a simplified equivalent definition for Definition 6.6.

THEOREM 6.2. *A DTD, D , is ambiguous by omitted tags without lookahead (Definition 6.6) iff there is a document instance V by and completely tagged document instances, V' by and V'' by, such that:*

1. b is an input symbol in Σ .
2. V corresponds to V' and to V'' , and
3. $V' \neq V''$,

PROOF. If D is ambiguous by omitted tags without lookahead, then all conditions of Definition 6.6 hold such that either b is in Σ or $by=by'=by''=e$.

Case A: Lemma 6.1 shows that if conditions 2 and 3 of Definition 6.6 hold for $by=by'=by''=e$, then conditions 2 and 3 also hold for some b in Σ .

Case B: If conditions 2 and 3 of Definition 6.6 hold and b is in Σ , then D is ambiguous without lookahead directly by Definition 6.6. \square

Theorem 6.2 shows an alternate definition for Definition 6.6. It simplifies the proof of correctness for Algorithm 6.4 that is shown in Lemmas 6.2-6.7 and Theorem 6.3. The definition in Theorem 6.2 is not given as the primary definition for DTDs that are ambiguous by omitted tags without lookahead because it does not illustrate that ambiguity can occur on end of input.

LEMMA 6.2. *Let S be a system of finite automata constructed from a DTD, D , by Algorithm 6.3. Let (M_j, q) , (M_j, r) , and (M_k, t) be in Q of S , α and β be in Γ^* , y be in Σ^* , and b be in $(\Sigma \cup e)$. Let a be $\langle N_j \rangle$, $\langle /N_j \rangle$, or $\langle N_j \rangle \langle /N_j \rangle$ in Σ_O^* . Then S makes a sequence of moves $((M_j, q), aby, \alpha) \vdash^+ ((M_j, r), by, \alpha) \vdash ((M_k, t), y, \beta)$ iff $((M_j, q), by, \alpha) \vdash ((M_k, t), y, \beta)$ and in any tree constructed by Algorithm 6.4, if there is an arc from (M_j, q) to (M_k, t) the label on the arc is a .*

PROOF. There are 5 cases to be considered. These are given in Propositions 6.1-6.5 that follow. Propositions 6.1-6.5 correspond to cases a-e of Step 3 of Algorithm 6.4 respectively.

PROPOSITION 6.1. *Let the omitted tag minimization for the begin tag of N_j in D be "O", let q_0 be the start state of M_j , and let b be in Σ , $b \neq \langle N_j \rangle$. Then $((M_j, q_0), \langle N_j \rangle by, \alpha) \vdash ((M_j, r), by, \alpha) \vdash ((M_j, t), y, \alpha)$, iff $((M_j, q_0), by, \alpha) \vdash ((M_j, t), y, \alpha)$ and in any tree constructed by Algorithm 6.4 if there is an arc from (M_j, q_0) to a leaf node for b , the label on the arc is $\langle N_j \rangle$.*

PROOF. Because the begin tag for N_j is "O" in the DTD, then in the NFA constructed for M_j in Algorithm 6.3, for some r and t in Q_j , $\delta(q_0, \langle N_j \rangle) \rightarrow r$ and $\delta(r, b) \rightarrow t$ for b in Σ iff $\delta(q_0, b) \rightarrow t$. This result is direct by applying Thompson's construction to $(\langle N_j \rangle | e) s$ for any regular expression s . The NFA constructed is illustrated in Figure 6.29; note that the notation $M(s)$ refers to the NFA for some regular expression s .

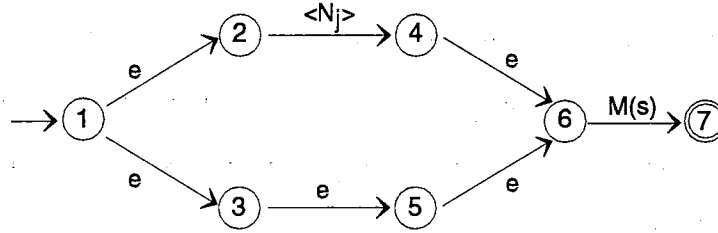


Figure 6.29. Thompson's construction for $((\langle N_j \rangle | e) s)$

In Figure 6.29 let $r=4$ and $(M_j, q_0)=1$. After Algorithm 5.2.a for removing e -transitions, $\delta(1, \langle N_j \rangle) \rightarrow 4$, and for all $\delta(6, b) \rightarrow t$ in $M(s)$ and b in Σ^* , $\delta(1, b) \rightarrow t$ and $\delta(4, b) \rightarrow t$ in $M((\langle N_j \rangle | e) s)$. Then for all b in Σ , S can make all moves from (M_j, q_0) that it makes from (M_j, r) . Thus, $((M_j, q_0), \langle N_j \rangle by, \alpha) \vdash ((M_j, r), by, \alpha) \vdash ((M_j, t), y, \alpha)$, iff $((M_j, q_0), by, \alpha) \vdash ((M_j, t), y, \alpha)$.

In the move $((M_j, q_0), by, \alpha) \vdash ((M_j, t), y, \alpha)$, the begin tag $\langle N_j \rangle$ is omitted. This corresponds to Step 3.a of Algorithm 6.4. Thus if there is an arc from (M_j, q_0) to a leaf node b in any tree constructed by Algorithm 6.4, the label on the arc is $\langle N_j \rangle$.

PROPOSITION 6.2. *Let the omitted tag minimization for the begin tag of N_j in D be "O", let q_0 be the start state of M_j , and let N_k be in N . Then $((M_j, q_0), \langle N_j \rangle y, \alpha) \vdash ((M_j, r), y, \alpha) \vdash ((M_k, q_0), y, (M_j, t)\alpha)$, iff $((M_j, q_0), y, \alpha) \vdash ((M_k, q_0), y, (M_j, t)\alpha)$ and in any tree constructed by Algorithm 6.4 if there is an arc from (M_j, q_0) to (M_k, q_0) the label on the arc is $\langle N_j \rangle$.*

PROOF. Because the begin tag for N_j is "O" in the DTD, in the NFA constructed for M_k in Algorithm 6.3, for some r and t in Q_j , $\delta(q_0, \langle N_j \rangle) \rightarrow r$ and $\delta(r, N_k) \rightarrow t$ for N_k in N iff $\delta(q_0, N_k) \rightarrow t$. The NFA constructed by Thompson's construction is the same as the NFA in case A above. After Algorithm 5.2.a for removing e-transitions, $\delta(1, \langle N_j \rangle) \rightarrow 4$, and for all $\delta(6, N_k) \rightarrow t$ in $M(s)$ and N_k in N , $\delta(1, N_k) \rightarrow t$ and $\delta(4, N_k) \rightarrow t$ in $M((\langle N_j \rangle | e) s)$. Then for all N_k in N , S can make all e-moves (pushes) from (M_j, q_0) that it makes from (M_j, r) . Thus, $((M_j, q_0), \langle N_j \rangle y, \alpha) \vdash ((M_j, r), y, \alpha) \vdash ((M_k, q_0), y, (M_j, t)\alpha)$, iff $((M_j, q_0), y, \alpha) \vdash ((M_k, q_0), y, (M_j, t)\alpha)$.

In the move $((M_j, q_0), y, \alpha) \vdash ((M_k, q_0), y, (M_j, t)\alpha)$ the begin tag $\langle N_j \rangle$ is omitted. This corresponds to Step 3.b of Algorithm 6.4. Thus if there is an arc from (M_j, q_0) to (M_k, q_0) in any tree constructed by Algorithm 6.4, the label on the arc is $\langle N_j \rangle$.

PROPOSITION 6.3. *Let the omitted tag minimization for the end tag of N_j in D be "O" and let f and f'' be in F_j . Then for all (M_k, t) such that there is a transition into (M_k, t) on N_j , $((M_j, f), \langle N_j \rangle y, (M_k, t)\alpha) \vdash ((M_j, f''), y, (M_k, t)\alpha) \vdash ((M_k, t), y, \alpha)$, iff $((M_j, f), y, (M_k, t)\alpha) \vdash ((M_k, t), y, \alpha)$ and in any tree constructed by Algorithm 6.4 if there is an arc from (M_j, f) to (M_k, t) the label on the arc is $\langle N_j \rangle$.*

PROOF. Because the omitted tag minimization is marked "O" for the end tag of N_j , in the NFA constructed for M_j by Algorithm 6.3, there must be states f and f'' in F_j , such that $\delta(f, \langle N_j \rangle) \rightarrow f''$ and f'' has no outgoing transitions. This is direct by Thompson's construction for $(s (\langle N_j \rangle | e))$ for any regular expression s . The NFA is of the form illustrated by Figure 6.30.

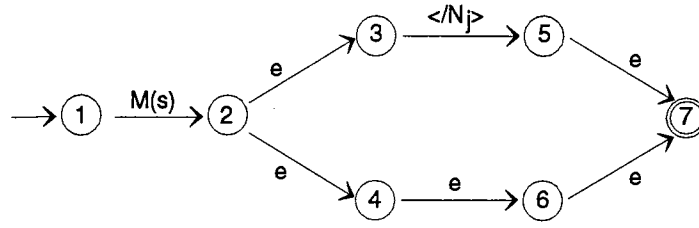


Figure 6.30. Thompson's construction for $(s (<N_j>le))$.

Then after Algorithm 5.2.a for e-removal, state 5 is a final states of $M(s (<N_j>le))$ and there are no transitions leaving state 5, and for any state f in $M(s)$ such that state 2 is in e-closure of f [2], then $\delta(f, <N_j>) \rightarrow 5$, and f and 5 are a final states in $M(s (<N_j>le))$.

Because f and f'' (5) are both in F_j , (M_j, f) and (M_j, f'') can make all of the same e-moves (pops), and thus $((M_j, f), <N_j> y, (M_k, t) \alpha) \vdash ((M_j, f''), y, (M_k, t) \alpha) \vdash ((M_k, t), y, \alpha)$, iff $((M_j, f), y, (M_k, t) \alpha) \vdash ((M_k, t), y, \alpha)$.

In the move $((M_j, f), y, (M_k, t) \alpha) \vdash ((M_k, t), y, \alpha)$, the end tag $<N_j>$ is omitted. This corresponds to Step 3.c of Algorithm 6.4. Then in any tree constructed by Algorithm 6.4, if there is an arc from (M_j, f) to (M_k, t) , the label on the arc is $<N_j>$.

PROPOSITION 6.4. *Let the omitted tag minimization for the begin and end tag of N_j in D be "O" and q_0 be the start state of M_j and q_0 and f be in F_j . Then for p and f in Q_j , $((M_j, q_0), <N_j> <N_j> y, (M_k, t) \alpha) \vdash ((M_j, p), <N_j> y, (M_k, t) \alpha) \vdash ((M_j, f), y, (M_k, t) \alpha) \vdash ((M_k, t), y, \alpha)$ iff $((M_j, q_0), y, (M_k, t) \alpha) \vdash ((M_k, t), y, \alpha)$ and in any tree constructed by Algorithm 6.4 if there is an arc from (M_j, q_0) to (M_k, t) the label on the arc is $<N_j> <N_j>$.*

PROOF. Because the omitted tag minimization for both the start tag and end tag of N_j are "O", and q_0 is the start tag of N_j and is also in F_j , then in the construction of M_j by Algorithm 6.3, it must be true that e is in M_j , as defined by the declared content or model group for N_j in D . Thompson's construction for $((<N_j>le) s (<N_j>le))$ for any regular expression s such that e is in $L(s)$ is shown in Figure 6.31. The notation $M(s)$ in Figure 6.31 refers to the component NFA for s , the model group or declared content of M_j .

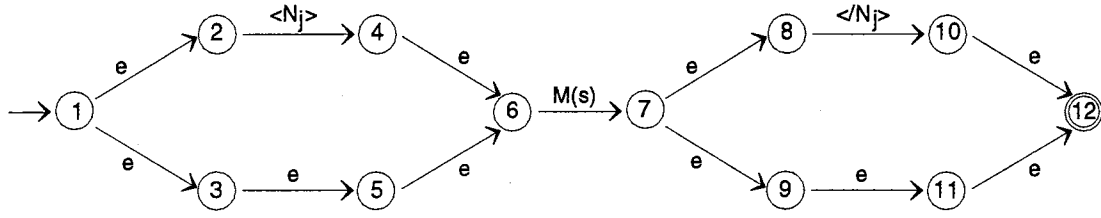


Figure 6.31. Thompson's construction for $((\langle N_j \rangle | e) s (\langle /N_j \rangle | e))$.

Thus, after e -removal, if e is in $L(s)$, then 7 is in e -closure of 6 [2] and thus 1 is e -closure of 12. Then 1 is a final state of $((\langle N_j \rangle | e) s (\langle /N_j \rangle | e))$.

Then because q_0 is a start tag and also an end tag, (M_j, q_0) can make the same e -moves (pops) as (M_j, f) . Thus, $((M_j, q_0), \langle N_j \rangle \langle /N_j \rangle y, (M_k, t)\alpha) \vdash ((M_j, p), \langle /N_j \rangle y, (M_k, t)\alpha) \vdash ((M_j, f), y, (M_k, t)\alpha) \vdash ((M_k, t), y, \alpha)$ iff $((M_j, q_0), y, (M_k, t)\alpha) \vdash ((M_k, t), y, \alpha)$.

In the move $((M_j, q_0), y, (M_k, t)\alpha) \vdash ((M_k, t), y, \alpha)$, both the start and end tag are omitted. This corresponds to Step 3.d in Algorithm 6.4. Then in any tree constructed by Algorithm 6.4, if there is an arc from (M_j, q_0) to (M_k, t) , the label on the arc is $\langle N_j \rangle \langle /N_j \rangle$.

PROPOSITION 6.5. *Let the omitted tag minimization for the end tag of N_0 in D be "O". Then $((M_0, f), \langle /N_0 \rangle, \alpha) \vdash ((M_0, f'), e, \alpha)$ for f and f' in F_0 iff in any tree constructed by Algorithm 6.4 if there is an arc from (M_0, f) to Z the label on the arc is $\langle /N_0 \rangle$.*

PROOF. Let the omitted tag minimization for the end tag of N_0 in D be "O". The result is shown by Thompson's construction for $(s (\langle /N_j \rangle | e))$ for any regular expression s , which is illustrated in Figure 6.30. Thus, $((M_0, f), \langle /N_0 \rangle, \alpha) \vdash ((M_0, f'), e, \alpha)$ iff f and f' are in F_0 .

Then in any tree constructed by Algorithm 6.4, if (M_0, f) occurs as a node in the tree, there is an arc from the node to Z for end of input. This corresponds to Step 3.e of Algorithm 6.4. In this case the end tag $\langle /N_0 \rangle$ is omitted. The label on the arc from (M_0, f) to Z is $\langle /N_0 \rangle$.

LEMMA 6.3. *Let S be a system of finite automata constructed from a DTD, D , by Algorithm 6.3, $(M_{g,s})$ and $(M_{h,t})$ be in Q of S , α and β be in Γ^* , x and u be in Σ^* . If $((M_{g,s}),xu,\alpha) \vdash^* ((M_{h,t}),u,\beta)$, then for some x' in Σ^* , $((M_{g,s}),x'u,\alpha) \vdash^* ((M_{h,t}),u,\beta)$ and there are no omitted tags in this sequence of moves.*

PROOF. Let (M_j,q) , (M_j,r) , and (M_k,t) be in Q of S , α' and β' be in Γ^* , y be in Σ^* , b be in $(\Sigma \cup e)$, and a be in Σ_O^+ . Consider $((M_{g,s}),xu,\alpha) \vdash^* ((M_{h,t}),u,\beta)$. In this sequence of moves, for each occurrence of a move $((M_j,q),by,\alpha') \vdash ((M_k,t),y,\beta')$ in which a is omitted, as shown in Lemma 6.2 there is a corresponding sequence of moves $((M_j,q),aby,\alpha') \vdash^+ ((M_j,r),by,\alpha') \vdash ((M_k,t),y,\beta')$, in which a is not omitted. Then, if there are n moves in which a tag is omitted in $((M_{g,s}),xu,\alpha) \vdash^* ((M_{h,t}),u,\beta)$, by n applications of Lemma 6.2 (Propositions 6.1-6.5), it must be true for some x' in Σ^* , that $((M_{g,s}),x'u,\alpha) \vdash^* ((M_{h,t}),u,\beta)$.

LEMMA 6.4. *For each M_i in M of S and for each state p in M_i , if there are states q and r in Q_i (q may equal r) and a and b in $(\Sigma \cup N)$ such that $\delta(q,a) \rightarrow p$ and $\delta(r,b) \rightarrow p$, then $a=b$.*

PROOF. The following proof shows that each state in Q_i is entered by transitions on only one symbol in $\Sigma_i \subseteq (\Sigma \cup N)$, and that this property holds at each step of the construction of M_i in Algorithm 6.3.

1: By Step 2 of Algorithm 6.2, M_i is constructed from a regular expression over Σ_i using Thompson's construction [2]. The property holds for all NFAs constructed by Thompson's construction as follows.

Basis: For each symbol b in a regular expression r , the basis step constructs an NFA consisting of a start state q , a final state f , and a single transition, $\delta(q,b) \rightarrow f$.

Inductive steps: For the inductive steps $M(r|s)$, and $M(r^*)$, only e transitions are added. For the inductive step $M(rs)$, the only transitions added are of the form, $\delta(f,b) \rightarrow p$ for the final state f of $M(r)$ iff q is the start state of $M(s)$ and $\delta(q,b) \rightarrow p$.

Thus, each state p in Q_i is entered by transitions on one and only one b in Σ_i .

2. Step 3 of Algorithm 6.3 removes ϵ -transitions from each M_i using Algorithm 5.2.a. Each transition added in this algorithm is of the form $\delta(q,a) \rightarrow p'$, such that $\delta(p,a) \rightarrow p'$ for some q, p, p' in Q_i and a in Σ_i . Thus the property holds after removing ϵ -transitions.
 3. In Step 4 of Algorithm 6.3, useless and inaccessible NFAs are removed from S using Algorithms 3.3 and 3.5. Removing these NFAs does not affect the remaining NFAs except in Step 4 of Algorithm 3.3 and Step 3 of Algorithm 3.5. In these steps for some N_k that is useless or inaccessible, all transitions on N_k in each remaining M_i in M are removed; there are no transitions added to M_i . Then the inaccessible and useless states are removed from each M_i by Algorithms 5.2.b and 5.2.c. In these algorithms, no transitions are added to M_i . Thus, the property holds after Step 4 of Algorithm 6.3.
- By 1-3 for all M_i , each state p in Q_i is entered by transitions on only one symbol in Σ_i .

LEMMA 6.5. *Let S be a system of finite automata constructed by Algorithm 6.3 and let T be an $LA(1)$ tree for some state (M_i, p) in Q of S , such that Algorithm 6.4 constructs an $ALA(1)$ tree for (M_i, p) . If there is any state that is repeated on a path in T , then for some state (M_g, t) in S there is an $ALA(1)$ tree constructed for (M_g, t) by Algorithm 6.4 that has two different paths from the root to leaf nodes for some input symbol b , and the sequences of labels on the two paths to b are not equal.*

PROOF. Suppose there is some state that is repeated on a path in T . Then there must be some state (M_j, r) that is the first occurrence of a repeated state on the path, where it may be true that $(M_j, r) = (M_i, p)$. Consider the $ALA(1)$ tree, T' , constructed for (M_i, p) by Algorithm 6.4. Because (M_j, r) is the first occurrence of a repeated state on a path in T , (M_j, r) is repeated on a path in T' . Then for any y in Σ^* and some α, α' , and β in Γ^* $((M_i, p), y, \alpha) \vdash^* ((M_j, r), y, \alpha') \vdash^+ ((M_j, r), y, \beta)$, where $((M_i, p), y, \alpha)$ may be $((M_j, r), y, \alpha')$. Then (M_j, r) must be either:

- A. (M_j, q_0) , where q_0 is the start state of M_j .
- B. (M_j, f) , for some f in F_j , $f \neq q_0$.
- C. (M_j, r) such that r is not q_0 of M_j or f in F_j .

Case A: Consider $(M_j, r) = (M_j, q_0)$. The next e-move from (M_j, q_0) must be a push or a pop. A push is a transition from (M_j, q_0) on some N_k ; for δ of M_j , $\delta(q_0, N_k) \rightarrow s$. Then for α in Γ^* , $((M_j, q_0), y, \alpha) \vdash ((M_k, q_0), y, (M_j, s)\alpha)$. Then by Step 3.b of Algorithm 6.4, this implies that $\langle N_j \rangle$ will be the label on the arc from (M_j, q_0) to (M_k, q_0) . A pop must be an e-move to some state (M_k, p') in S such that for some α in Γ^* , $((M_j, q_0), y, (M_k, p')\alpha) \vdash ((M_k, p'), y, \alpha)$. By Step 3.d of Algorithm 6.4, this implies that $\langle N_j \rangle \langle N_j \rangle$ will be the label on the arc from (M_j, q_0) to (M_k, p') .

The path from the root (M_i, p) to the first occurrence of (M_j, q_0) has some sequence of labels V , such that V may be e . Then as shown above, the sequence of labels on the path from the first (M_j, q_0) to the second (M_j, q_0) must be some V_1 containing at least $\langle N_j \rangle$ or $\langle N_j \rangle \langle N_j \rangle$. Thus, the path from (M_i, p) to the second occurrence of (M_j, q_0) must be VV_1 . Then $V_1 \neq e$ implies $V \neq VV_1$.

Because (M_j, q_0) is the start state of M_j , by the construction of M_j it must be true that $\Delta((M_j, q_0), \langle N_j \rangle, e) \rightarrow ((M_j, s), e)$ for some s in M_j . Then by Step 3.a of Algorithm 6.4, the leaf node $\langle N_j \rangle$ is a child of each node for (M_j, q_0) in the tree, and the label on the path from each (M_j, q_0) to $\langle N_j \rangle$ is e . Thus, the path from the root to the child node $\langle N_j \rangle$ of the first occurrence of (M_j, q_0) will be V and the path from the root to the child node $\langle N_j \rangle$ of the second occurrence of (M_j, q_0) will be VV_1 .

Therefore, there are two paths in the ALA(1) tree for (M_i, p) to leaf nodes for $\langle N_j \rangle$ and the sequences of labels on the two paths are not equal. Thus, for Case A, $(M_g, t) = (M_i, p)$.

Case B: Consider $(M_j, r) = (M_j, f)$, for some f in F_j , $f \neq q_0$. Because (M_j, f) is not the start state of M_j , the only way that (M_j, f) can be reached is by a transition on some symbol in $(\Sigma \cup N)$. By Lemma 6.4, there can only be transitions into (M_j, f) on one symbol in $(\Sigma \cup N)$. Then there cannot be a transition into (M_j, f) on some b in Σ because there could not be a sequence of e-moves $((M_j, f), y, \alpha) \vdash^+ ((M_j, f), y, \beta)$ for some α, β in Γ^* , and this would contradict the assumption that (M_j, f) was a state repeated on a path. Then, (M_j, f) must be entered only by transitions on one input symbol N_k in N . Thus, in any sequence

of moves to (M_j, f) , the last move in the sequence must be a pop from a state (M_k, f') where f' is in F_k . Then any sequence of moves from (M_j, f) to (M_j, f) must be of the form: $((M_j, f), y, \alpha) \vdash^* ((M_k, f'), y, (M_j, f)\beta) \vdash ((M_j, f), y, \beta)$ for some α and β in Γ^* and y in Σ^* .

By Lemma 6.4, (M_k, f') can be entered only by transitions on one symbol in $(\Sigma \cup N)$. Then (M_k, f') must be a final state of M_k that is not entered by a transition on some b in Σ , because this would imply that there could not be a sequence of e-moves: $((M_j, f), y, \alpha) \vdash^* ((M_k, f'), y, (M_j, f)\beta) \vdash ((M_j, f), y, \beta)$. Thus (M_k, f') is not entered by a transition on $\langle N_k \rangle$, and by Step 3.c of Algorithm 6.4, the arc from (M_k, f') to (M_j, f) must have a label $\langle N_k \rangle$.

The path from the root (M_i, p) to the first occurrence of (M_j, f) has some sequence of labels V , such that V may be e . The sequence of labels on the path from the first (M_j, f) to the second (M_j, f) must be some V_1 containing at least $\langle N_k \rangle$. Thus, the path from (M_i, p) to the second occurrence of (M_j, f) must be VV_1 . Then $V_1 \neq e$ implies $V \neq VV_1$.

Because (M_j, f) is not entered by a transition on some b in Σ it is not entered by a transition on $\langle N_j \rangle$. Then by the construction of M_j , because f is in F_j , there must be a transition from (M_j, f) on $\langle N_j \rangle$. The construction of M_j is illustrated in Figure 6.30. Thus, by Step 3.a of Algorithm 6.4, a leaf node $\langle N_j \rangle$ must be a child of each occurrence of (M_j, f) in the tree, and the label on each arc from (M_j, f) to $\langle N_j \rangle$ is e .

Thus, the path from the root to the child node $\langle N_j \rangle$ of the first occurrence of (M_j, f) will be $V\langle N_j \rangle$ and the path from the root to the child node $\langle N_j \rangle$ of the second occurrence of (M_j, f) will be $VV_1\langle N_j \rangle$. Then because $V \neq VV_1$, $V\langle N_j \rangle \neq VV_1\langle N_j \rangle$. Thus there are two different paths in the ALA(1) tree for (M_i, p) to leaf nodes for $\langle N_j \rangle$ and the sequences of labels on the two paths are not equal. Thus, for Case B, $(M_g, t) = (M_i, p)$.

Case C: Consider (M_j, r) such that r is not q_0 of M_j or f in F_j . Because (M_j, r) is not a final state of M_j , the first move in the sequence of e-moves, $((M_j, r), y, \alpha) \vdash^+ ((M_j, r), y, \beta)$, must be a push to some state (M_k, q_0) as follows: $((M_j, r), y, \alpha) \vdash ((M_k, q_0), y, (M_j, s)\alpha) \vdash^+ ((M_j, r), y, \beta)$ for some (M_j, s) and for some α and β in Γ^* .

If the begin tag for M_k , $\langle N_k \rangle$ is not optional, then the next move from (M_k, q_0) cannot be an e-move, and it cannot be true that $((M_j, r), y, \alpha) \vdash^+ ((M_j, r), y, \beta)$, and this contradicts the assumption that there is a path from (M_j, r) to (M_j, r) in the ALA(1) tree for (M_i, p) . Thus, the begin tag for M_k , $\langle N_k \rangle$, must be optional and it must be true that $((M_j, r), y, \alpha) \vdash ((M_k, q_0), y, (M_j, s)\alpha) \vdash^+ ((M_j, r), y, \beta) \vdash ((M_k, q_0), y, (M_j, s)\beta)$. Because (M_j, r) is the first repeated state on some path in the ALA(1) tree for (M_i, p) , there must be no other repeated states in this sequence of moves. Then, there must be no repeated states other than (M_k, q_0) in the sequence of moves $((M_k, q_0), y, (M_j, s)\alpha) \vdash^+ ((M_j, r), y, \beta) \vdash ((M_k, q_0), y, (M_j, s)\beta)$.

Because (M_k, q_0) is the start state of M_k , there will be an ALA(1) tree constructed for it by Algorithm 6.4, and by the above result there is a path in the tree such that (M_k, q_0) is the first repeated state on the path. Then because q_0 is the start state of M_k , the proof in Case A holds for the special case of the root $= (M_k, q_0)$. Thus, there is an ALA(1) tree constructed for (M_k, q_0) such that there are two paths in the tree to leaf nodes for $\langle N_k \rangle$, and the sequences of labels on the paths are not equal. Thus for Case C, $(M_g, t) = (M_k, q_0)$.

Therefore, for cases A, B, and C, if there is any occurrence of a state that is repeated on a path in some LA(1) tree, then for some state (M_g, t) in S there is an ALA(1) tree constructed by Algorithm 6.4 such that there are two paths from the root to leaf nodes for an input symbol b and the sequences of labels on the two paths are not equal.

LEMMA 6.6. *Let S be a system of finite automata constructed from a DTD by Algorithm 6.3 and (M_i, p) be a state in S for which an ALA(1) tree is constructed by Algorithm 6.4. If there are two paths in the tree for (M_i, p) from the root to leaf nodes for Z such that the sequences of labels on the two paths are not equal, then there is some state (M_g, t) of S , where (M_g, t) may be (M_i, p) , for which a tree is constructed and there are two paths in the tree to leaf nodes for some b in Σ such that the sequences of labels on the paths are not equal.*

PROOF. In any ALA(1) tree constructed by Algorithm 6.4, for any path to a leaf node Z , the parent of Z must be a final state (M_0, f) for some f in F_0 . In the ALA(1) tree for (M_i, p) , let the parents of the two leaf nodes for Z be (M_0, f) and (M_0, f'') , f and f'' in F_0 . By the construction of the NFA, M_0 , either (M_0, f) is entered by a transition on $\langle N_0 \rangle$ or it is not entered by a transition on N_0 , and either (M_0, f'') is entered by a transition on $\langle N_0 \rangle$ or it is not entered by a transition on $\langle N_0 \rangle$.

Case A: Suppose (M_0, f) is entered by a transition on $\langle N_0 \rangle$. Then by Lemma 6.4, it is only entered by transitions on $\langle N_0 \rangle$. Thus, there are no e-moves to (M_0, f) , and it can only occur in the ALA(1) tree for (M_i, p) if it is the root. Then, the occurrence of (M_0, f'') in the tree for (M_i, p) must be on some path from the root. Because (M_0, f) is entered by $\langle N_0 \rangle$ it is not the start state of M_0 . Thus, the first move in the sequence of moves from (M_0, f) to (M_0, f'') must be a pop. Because (M_0, f) is the root and (M_0, f'') is a node in the tree that is not the root, then for all y in Σ^* and for some α and β in Γ^* , $((M_0, f), y, \alpha) \vdash^+ ((M_0, f''), y, \beta)$. Then because (M_0, f'') is also a final state in F_0 , it makes the same e-moves (pops) as (M_0, f) , and thus for some α' and β' in Γ' , $((M_0, f''), y, \alpha') \vdash^+ ((M_0, f''), y, \beta')$. Thus, the state (M_0, f'') must be repeated on a path in the LA(1) tree for (M_0, f) . Because (M_0, f) is entered by a transition on $\langle N_0 \rangle$, there is an ALA(1) tree constructed for (M_0, f) by Algorithm 6.4. Then by Lemma 6.5, there is some state (M_g, t) in S , such that there is an ALA(1) tree constructed for (M_g, t) by Algorithm 6.4 and there are two paths in the tree to leaf nodes for b in Σ , such that the sequences of labels on the two paths are not equal. Thus, for this case, $(M_g, t) \neq (M_i, p)$.

Case B: The proof for (M_0, f) in A holds for (M_0, f'') entered by a transition on $\langle N_0 \rangle$.

Case C: Suppose (M_0, f) and (M_0, f'') are both entered by transitions on $\langle N_0 \rangle$. This implies a contradiction. By the proofs for A and B, (M_0, f) and (M_0, f'') cannot both be the root of the tree for (M_i, p) . Thus, it cannot be true that both (M_0, f) and (M_0, f'') are entered by transitions on $\langle N_0 \rangle$.

Case D: Suppose (M_0, f) and (M_0, f'') are not entered by transitions on $\langle N_0 \rangle$. Then by Step 3.e of Algorithm 6.4, the arcs from (M_0, f) and (M_0, f'') to Z must both have a label $\langle N_0 \rangle$, and because the sequence of labels on the two paths from (M_i, p) to Z are not equal, the sequence of labels on the path from (M_i, p) to (M_0, f) is not equal to the sequence of labels on the path from (M_i, p) to (M_0, f'') . Let these paths be denoted V and V' respectively. Because (M_0, f) and (M_0, f'') are final states that are not entered by a transition on $\langle N_0 \rangle$, by the construction of M_0 they must each have a child $\langle N_0 \rangle$ in the tree, and by Step 3.a of Algorithm 6.4 the label on the arcs from (M_0, f) to $\langle N_0 \rangle$ and from (M_0, f'') to $\langle N_0 \rangle$ is e . Then the sequence of labels from the root (M_i, p) to the child $\langle N_0 \rangle$ of (M_0, f) is V and the sequence of labels from the root to the child $\langle N_0 \rangle$ of (M_0, f'') is V' . Thus, the tree for (M_i, p) has two paths to leaf nodes for $\langle N_0 \rangle$ in Σ , and the sequences of labels on the paths are not equal. In this case, $(M_g, t) = (M_i, p)$.

Therefore, by A-D, if there are two paths in the tree for (M_i, p) from the root to leaf nodes for Z such that the sequences of labels on the two paths are not equal, then there is some state (M_g, t) of S for which a tree is constructed and such that there are two paths in the tree to leaf nodes for b in Σ , and the sequences of labels on the paths are not equal. \square

The following lemma shows an equivalent definition for Definition 6.6 that is derived from the equivalent definition shown in Theorem 6.2. It is used in Theorem 6.3, which shows the proof of correctness for Algorithm 6.4; Algorithm 6.4 detects ambiguity between an input symbol a in Σ (or a start configuration) and the next input symbol b in Σ .

LEMMA 6.7. *A DTD D is ambiguous by omitted tags without lookahead iff there is a document instance $xaby$ and completely tagged document instances $xaVby'$ and $xaV''by''$ such that:*

1. b is an input symbol in Σ ,
2. either a is an input symbol in Σ or $xa = e$.
3. V' and V'' are in Σ^* , and
4. $V' \neq V''$.

PROOF. The proof is given for the equivalent definition of ambiguity in Theorem 6.2.

Case A: Suppose D is ambiguous by omitted tags without lookahead. By Theorem 6.2 there is a document instance $w=Vby$, and completely tagged document instances, $w=V'by'$ and $w=V''by''$, b is an input symbol in Σ , V corresponds to V' and to V'' , and $V' \neq V''$.

Case A.1: Suppose $V=e$. Then V corresponds to V' and V'' implies that V' and V'' are in Σ_O^* . Let $xa=e$. Then $xaby$ is a document instance and $xaV'by'$ and $xaV''by''$ are completely tagged document instances.

Case A.2: Suppose $V=v_1 \dots v_n$, $n \geq 1$. Let $V'=v'_1 \dots v'_k$ and $V''=v''_1 \dots v''_m$. Because V corresponds to V' and to V'' , then $k \geq 1$ and $m \geq 1$. Consider the first i such that $v_i \neq v'_i$ or $v_i \neq v''_i$. Because $V' \neq V''$, then either:

- a. $v_i = v'_i$ and $v_i \neq v''_i$,
- b. $v_i \neq v'_i$ and $v_i = v''_i$, or
- c. $v_i \neq v'_i$ and $v_i \neq v''_i$.

Case A.2.a/b: The proof for a also holds for b. Because $v'_1 = v''_1 = \langle \text{DOCTYPE} \rangle$, then $i > 1$. Let $xa = v_1 \dots v_{i-1}$, $b = v_i = v'_i$, and $V' = e$. Because V corresponds to V'' , Vb corresponds to $V''b$, and thus there must be a first occurrence of $b = v''_j$ in V'' for some j , $j > i$. Let $V'' = v''_j \dots v''_{j-1}$. Then $V'' \neq e$ and thus $V'' \neq V'$. Let $y = v_{i+1} \dots v_n by$, $y' = v'_{i+1} \dots v'_k by'$ and $y'' = v''_{j+1} \dots v''_m by''$. Then $xaby = Vby$ is a document instance and $xaV'by' = V'by'$ and $xaV''by'' = V''by''$ are completely tagged document instances, and V' and V'' are in Σ_O^* .

Case A.2.c: Let $v_i \neq v'_i$ and $v_i \neq v''_i$.

Case A.2.c.1: Suppose $v'_i \neq v''_i$. Let $xa = v_1 \dots v_{i-1}$ (xa may be e), $b = v_i$, and $y = v_{i+1} \dots v_n$. Because V corresponds to V' and to V'' and by the conditions for i , there must be some next occurrence of $b = v_i$ in V' , v'_h , $h > i$, and some next occurrence of b in V'' , v''_j , $j > i$. Let $V' = v'_i \dots v'_h$, $V'' = v''_i \dots v''_j$, Let $y = v_{i+1} \dots v_n by$, $y' = v'_{h+1} \dots v'_k by'$, and $y'' = v''_{j+1} \dots v''_m by''$. Because $v'_i \neq v''_i$, $V' \neq V''$. Then $xaby = Vby$ is a document instance, $xaV'by' = V'by'$ and $xaV''by'' = V''by''$ are completely tagged document instances, and V' and V'' are in Σ_O^* .

Case A.2.c.2: Suppose $v'_i = v''_i$. Let $b = v'_i = v''_i$. Then because V corresponds to V' and to V'' , b is in Σ_O and $v_1, \dots, v_{i-1}bv_i, \dots, v_n$ corresponds to V' and to V'' (by not omitting b).

Step 1: The occurrence of b is now at position i in V and the old v_i is at position $i+1$. Consider a new V , $V = v_1, \dots, v_{i-1}bv_i, \dots, v_n$. Let $i = i+1$. V still corresponds to V' and to V'' .

Repeat Step 1 until one of the cases for A.2 is true for the new V and i . The only case that remains to be proved is A.2.c.2: Consider $V' = v'_1, \dots, v'_k$ and $V'' = v''_1, \dots, v''_m$. Because $\min(k, m)$ is finite and because $V' \neq V''$, then it must eventually be true that either $v'_i \neq v''_i$ or $i = \min(k, m)$. If $v'_i \neq v''_i$, then the proof for Case A.2.c.1 holds. If $i = \min(k, m)$, by the conditions for i either $k > m$ or $m > k$. The same proof holds for either case: let $m > k$. Because Vb corresponds to $V'b$ and by the conditions for i , then $n = k$ and $V = V'$. Let $xa = V$, $V' = e$, and $V'' = v''_{i+1}, \dots, v''_m$. Then $xab = Vb$ corresponds to $xaV'b = V'b$ and to $xaV''b = V''b$, and thus V' and V'' are in Σ_O^* .

Case B: Suppose there is a document instance $xaby$ and completely tagged document instances $xaV'by'$ and $xaV''by''$ such that b is an input symbol in Σ , either a is an input symbol in Σ or $xa = e$, V' and V'' are in Σ_O^* , and $V' \neq V''$. Let $V = xa$, $V' = xaV'$, and $V'' = xaV''$. Then there is a document instance Vby and completely tagged document instances $V'by'$ and $V''by''$, such that V corresponds to V' and to V'' , b is in Σ , and $V' \neq V''$. Then by Theorem 6.2, D is ambiguous by omitted tags without lookahead.

Therefore, by A and B the result holds.

THEOREM 6.3. *Algorithm 6.4 detects all DTDs and only DTDs that are ambiguous by omitted tags without lookahead (Definition 6.6).*

PROOF. The proof is given for the equivalent definition of DTDs that are ambiguous by omitted tags shown in Lemma 6.7.

Case A: To show that Algorithm 6.4 detects all ambiguous DTDs it must be shown for each ambiguous DTD, D , if S is a system of finite automata constructed by Algorithm 6.3, then there is some state (M_i, p) in S , such that Algorithm 6.4 constructs an ALA(1) tree for (M_i, p) and there are paths in the tree to two leaf nodes for b in Σ , and the sequences of

labels on the two paths are not equal, or there are paths in the tree to two leaf nodes Z for end of input, and the sequences of labels on the two paths are not equal.

Suppose A DTD is ambiguous by omitted tags without lookahead. Then by Lemma 6.7 there is a document instance $xaby$ and completely tagged document instances $xaV'by'$ and $xaV''by''$ such that b is an input symbol in Σ , either a is an input symbol in Σ or $xa=e$, V' and V'' are in Σ_O^* , and $V' \neq V''$.

If a is an input symbol, then for some (M_i, q) and (M_i, p) in Q of S , and α in Γ^* , S makes the following sequence of moves: $((M_0, q_0), xaby, e) \vdash^* ((M_i, q), aby, \alpha) \vdash (M_i, p), by, \alpha$. If $xa=e$, let $(M_i, p) = (M_0, q_0)$ and $\alpha=e$. Consider the state (M_i, p) . Because it is either a state entered by a transition on some input symbol a in Σ or it is the start state of S , Algorithm 6.4 constructs a tree for (M_i, p) . Consider the LA(1) tree for (M_i, p) . Either it has a repeated state on some path or it does not have a repeated state.

Case A.1: If there is some state that is repeated on a path in the LA(1) tree for (M_i, p) , then by Lemma 6.5, for some state (M_g, t) in S , Algorithm 6.4 constructs an ALA(1) tree for (M_g, t) , such that there are two paths in the tree to leaf nodes for b , and the sequences of labels on the two paths are not equal.

Case A.2: If there is no state that is repeated on a path in the LA(1) tree for (M_i, p) . Then each path in the tree terminates and by Definition 6.9 the ALA(1) tree for (M_i, p) is the same as the LA(1) tree. Consider the ALA(1) tree constructed for (M_i, p) . Because $xaby$ is a document instance, there is a path in the tree for (M_i, p) to a leaf node for b . Because V' is in Σ_O^* , if $V' = v'_1 \dots v'_n$, $n \geq 0$, then for $i=1, \dots, n$, v'_i is omitted in the sequence of moves accepting b that is represented by the path in the tree. Then by Lemma 6.2, v'_i , $i=1, \dots, n$ will occur in order on arcs on the path from (M_i, p) to b and no other labels will occur on the arcs of this path. Thus, there is a path to a leaf node b in the tree for (M_i, p) and the sequence of labels on the path is V' . The same proof holds for V'' . Then since $V' \neq V''$, the result is complete.

By A.1 and A.2, there is some state (M_i, p) or (M_g, t) in S , such that Algorithm 6.4 constructs an ALA(1) tree for the state and there are paths in the tree to two leaf nodes for b in Σ , and the sequences of labels on the two paths are not equal. Therefore, Algorithm 6.4 detects all DTDs that are ambiguous by omitted tags without lookahead.

Case B: Algorithm 6.4 detects only ambiguous DTDs. Let D be a DTD and S be a system of finite automata constructed from D by Algorithm 6.3. If Algorithm 6.4 outputs YES for a DTD, D , then there is some state (M_i, p) in Q of S such that there is a tree constructed for (M_i, p) by Algorithm 6.4 and there are paths in the tree from the root to two leaf nodes for b in Σ or to two leaf nodes for Z (denoting end of input), and the sequences of labels on the paths are not equal. By Lemma 6.6, if the tree for (M_i, p) has paths to two leaf nodes Z , such that the sequences of labels on the path are not equal, then there is some state (M_g, t) in S , where (M_g, t) may be (M_i, p) , such that Algorithm 6.4 constructs a tree for (M_g, t) and there are paths from the root to two leaf nodes for b in Σ , and the sequences of labels on the two paths are not equal. The following proof for (M_i, p) also holds for (M_g, t) . Thus, it is only necessary to consider the case of two paths to leaf nodes for b in Σ to show that D is ambiguous by omitted tags without lookahead.

Let the sequences of labels on the two paths to b be V' and V'' . The following proof shows that for V' , there is a document instance $xaby$ and a completely tagged document instance $xaV''by'$, such that either a is an input symbol in Σ or $xa=e$, b is an input symbol in Σ , and V' is in Σ_O^* . The same proof holds for V'' . The proof consists of 3 parts:

1. Because a tree is constructed for (M_i, p) by Algorithm 6.4, either $(M_i, p) = (M_0, q_0)$ or (M_i, p) is entered by a transition on some symbol a in Σ . If (M_i, p) is entered by a transition on a , then because there are no useless states in S , there must be some string $x'aby$ in $L(D)$ and some α in Γ^* such that $((M_0, q_0), x'aby, e) \vdash^* ((M_i, q), aby, \alpha) \vdash^* ((M_i, p), by, \alpha)$. Then by Lemma 6.3 there is some x in Σ^* such that $((M_0, q_0), xaby, e) \vdash^* ((M_i, q), aby, \alpha) \vdash^* ((M_i, p), by, \alpha)$ and there are no tags omitted in this sequence of moves. If $(M_i, p) = (M_0, q_0)$, then $xa=e$ and $((M_0, q_0), xaby, e) \vdash^0 ((M_i, p), by, e)$. Thus, for either

- $xa=e$ or a in Σ , $((M_0, q_0), xaby, e) \vdash^* ((M_i, p), by, \alpha)$, and there are no tags omitted in this sequence of moves.
2. Consider the leaf node b on the path with the sequence of labels V' . Because b is a leaf node in the tree for (M_i, p) , then for some (M_j, s) and (M_j, r) in Q of S and β in Γ^* there must be a sequence of moves $((M_i, p), by, \alpha) \vdash^* ((M_j, r), by, \beta) \vdash ((M_j, s), y, \beta)$. Then because V' is the sequence of labels on the path from (M_i, p) to b , $xaV'by$ is in $L(D)$ and there is a sequence of moves $((M_i, p), V'by, \alpha) \vdash^* ((M_j, r), by, \beta) \vdash ((M_j, s), y, \beta)$ in which there are no omitted tags.
 3. Because $xaV'by$ is in $L(D)$ and because there are no useless states in S , $((M_j, s), y, \alpha) \vdash^* ((M_0, f), e, e)$ for some f in F_j . Then by Lemma 6.3, there is some y' in Σ^* such that $((M_j, s), y', \alpha) \vdash^* ((M_0, f), e, e)$ and there are no omitted tags in this sequence of moves. By the results from 1-3, $((M_0, q_0), xaV'by', e) \vdash^* ((M_i, q), aV'by', \alpha) \vdash ((M_i, p), V'by', \alpha) \vdash^* ((M_j, r), by', \beta) \vdash ((M_j, s), y', \beta) \vdash^* ((M_0, f), e, e)$, and there are no omitted tags in this sequence of moves. Then $xaVby'$ is in $L(D')$ where D' is derived from D by setting the omitted tag minimization to minus for all elements. Therefore, $xaV'by'$ is a completely tagged document instance. Thus, for V' , there is a document instance $xaby$ and a completely tagged document instance $xaV'by'$, such that either a is an input symbol in Σ or $xa=e$, b is an input symbol in Σ , and V' is in Σ_O^* . The same proof holds for V'' . Then because $V' \neq V''$, if Algorithm 6.4 outputs YES for a DTD, D , by Lemma 6.7 D is ambiguous by omitted tags without lookahead.

Therefore by A and B, Algorithm 6.4 detects all DTDs and only DTDs that are ambiguous by omitted tags without lookahead. \square

In this chapter a method is shown for constructing a system of finite automata recognizing $L(D)$ for a DTD, D , when exceptions are not considered. A definition is given for DTDs that are ambiguous by omitted tags without lookahead, and an algorithm is shown for detecting these DTDs. In the next chapter, these methods are extended to include exceptions.

7. Exceptions

In this chapter, the effect of exceptions on $L(D)$ is shown by Algorithm 7.1 that enumerates the ways that exceptions can apply to model groups in a particular DTD. Then, Algorithm 7.2 uses Algorithm 7.1 to construct a system of finite automata, S' , recognizing $L(D)$ when exceptions are considered. S' is an extension of S , the system of finite automata constructed by Algorithm 6.3. Examples are shown that illustrate that Algorithm 6.4 can be applied to S' to detect DTDs that are ambiguous by omitted tags when exceptions are considered. Algorithm 7.1 is also used to implement the optional SGML feature, `VALIDATE EXCEPTIONS`, for reporting exceptions that attempt to remove a required element from a model group.

7.1. Preliminary Definitions

The following three definitions are from the glossary of the standard [13].

4.130 exceptions: A parameter of an element declaration that modifies the effect of the element's *content model*, and the content models of elements occurring within it, by permitting inclusions and prohibiting exclusions.

4.131 exclusions: Elements that are not allowed anywhere in the content of an element or its subelements even though the applicable content model or inclusions would permit them optionally.

4.157 inclusions: elements that are allowed anywhere in the content of an element or its subelements even though the applicable model does not permit them.

The subelements of an element A are defined to be only the elements that occur immediately within A (at the first level of nesting) [13]. Thus, in Definitions 4.131 and 4.157, either the phrase "or in its subelements" is redundant, or the phrase "anywhere in the content of an element" is not precise. However, the example shown in Section B.11.2 of the tutorial annex B [13] shows that the exclusions for an element A apply to elements that are nested below the immediate subelements of A . The algorithms that follow are

based on this interpretation: the exceptions for an element A apply anywhere in the content of an A, including in any elements that occur within A at any level of nesting.

In addition to the above definitions, Clause 11.2.5 of the standard states: "At any point in a document instance, if an element is both an applicable inclusion and an exclusion, it is treated as an exclusion."

Definition 7.1. Ancestors of elements. An element A is an ancestor of an element B and B is a descendent of A iff B occurs within A at some level of nesting.

Definition 7.2. Local exceptions. The inclusions declared in the content model of an element A are local inclusions for A. The exclusions declared in the content model of an element A are local exclusions for A. The local inclusions and local exclusions for A are the local exceptions for A.

Example 7.1. Local exceptions. Consider the element declaration:

```
<!ELEMENT A - - (A | B) - (C) + (X,Y) >
```

The local exclusions for A are {C} and the local inclusions for A are {X,Y}.

Definition 7.3. Inherited exceptions. The inherited inclusions of an element A are the union of the local inclusions of all ancestors of A. The inherited exclusions of an element A are the union of the local exclusions of all ancestors of A. Note that a local inclusion, X, for some ancestor B of A is still an inherited inclusion of A, even if X is a local exclusion for some ancestor C of A. This definition is used in the algorithms that follow and does not imply a contradiction to the rule in Clause 11.2.5, which states that exclusions override inclusions. Inherited exceptions are illustrated in Example 7.2.

Example 7.2. Inherited exceptions. Consider the DTD:

```
<!DOCTYPE A [
  <!ELEMENT A - - (B) - (Y) + (X) >
  <!ELEMENT B - - (C) - (X) + (Z) >
  <!ELEMENT C - - (#PCDATA) - (Z) + (Y)> ]
```

The inherited exclusions for C are {X,Y} and the inherited inclusions for C are {X,Z}.

Definition 7.4. Applicable exceptions. The applicable inclusions for A are the inherited inclusions for A plus the local inclusions of A. The applicable exclusions of A are the inherited exclusions of A plus the local exclusions for A. The applicable exceptions are denoted as $- \{ \}$ and $+ \{ \}$ for exclusions and inclusions respectively. In Example 7.2, the applicable exclusions for C are $- \{ X, Y, Z \}$ and the applicable inclusions are $+ \{ X, Y, Z \}$.

Definition 7.5. Effective exceptions. The effective inclusions for an element A are the applicable inclusions for A minus the applicable exclusions. The effective exclusions are the applicable exclusions. Thus, the effective exceptions are those that apply to the model group of the element as defined by Clause 11.2.5 of the standard. In Example 7.2, the effective inclusions for C are $\{ \}$ and the effective exclusions are $\{ X, Y, Z \}$.

Each element A defined in a DTD either has declared content or is defined by a content model (model group and optional exceptions). The content of A elements in document instances may vary depending on the context in which they occur, because exceptions apply anywhere within an element, including within any descendent of the element (which may be A). Model groups can introduce recursion into a DTD and inclusions imply another level of recursion; by definition, an included element A can contain an A unless otherwise excluded. Exclusions are frequently used to limit recursion [10].

Definition 7.6. Dynamic content model. For a DTD, D, a dynamic content model (DCM) for an element A defined in D, is the model group for A and a set of applicable exceptions for A that apply to some occurrence of A in some document instance defined by D. For the purposes of this definition, all elements are assumed to be defined by a model group as follows: for elements with declared content of CDATA or RCDATA, the content definition is equivalent to a model group ($\#PCDATA$). Thus, these elements are considered to be defined by the model group, ($\#PCDATA$). The model group for elements with declared content of "EMPTY" is NULL. Exceptions do not apply to elements with declared content [13]. Thus, the applicable exceptions for all DCMs with

declared content are empty. Because exceptions do not apply to elements with declared content, they have only one DCM.

Each element in a DTD has a finite number of DCMs. This is clear as follows: let Y be the set of all combinations of inclusions that occur in D and let Z be the set of all combinations of exclusions. Then, because Y and Z are finite, $(Y \times Z)$ is a finite set of ordered pairs. The DCMs of each element must be a subset of $(Y \times Z)$, and therefore is a finite set. The DCMs for an element are distinguished from each other by their respective sets of applicable exceptions. For notational purposes, a unique index is assigned to each DCM as illustrated in Example 7.3.

Example 7.3. Dynamic content models (DCMs). Consider the DTD, D :

```
<!DOCTYPE A [
  <!ELEMENT A - - (B | C) >
  <!ELEMENT B - - (C) +(X) >
  <!ELEMENT C - - (#PCDATA) >
  <!ELEMENT X - - (#PCDATA) -(X) > ] >
```

The DCMs for D are shown in Table 7.1.

Table 7.1. Dynamic content models for Example 7.3.

name/ index	model group	applicable inclusions	applicable exclusions
A_1	(B C)	{}	{}
B_1	(C)	{X}	{}
C_1	(#PCDATA)	{}	{}
C_2	(#PCDATA)	{X}	{}
X_1	(#PCDATA)	{X}	{X}

7.2 The Effect of Exceptions on Content Models

Algorithm 7.1 that follows shows how exceptions affect $L(D)$. In particular, a tree is traversed such that each node in the tree is a DCM for some element in D . The nodes of the tree enumerate all of the DCMs for all elements named in the element type of some element declaration in D . The tree shows the context in which each DCM can occur in

relation to other DCMs of D . The name of each node in the tree is the element name plus the index of the DCM, and each node has labels (attributes) for the applicable inclusions and the applicable exclusions. The leaf nodes of the tree are one of the following:

1. DCMs that have no children. This can occur in one of the following ways:
 - a. the element has declared content.
 - b. the element has a model group that has no GIs (element names), and there are no effective inclusions.
 - c. all GI's in the model group of the element and any included elements are excluded by effective exclusions.
2. DCMs that have already occurred as some node in the tree.

For both of these cases the current path is terminated. The tree is traversed rather than explicitly constructed. As the tree is traversed, a list of the DCMs of D is maintained. The list is used to terminate paths when a DCM has already occurred in the tree, and this guarantees that the algorithm terminates. The list of the DCMs is extended so that for each DCM, there is a list of the children of the DCM in the tree. For DCMs that are leaf nodes (they have already occurred) the list of children is obtained from the list of children for the first occurrence of the DCM; by the definition of DCMs, the children of all occurrences of a particular DCM in the tree must be the same. This list is then used by Algorithm 7.2 to construct a system of finite automata recognizing $L(D)$.

ALGORITHM 7.1. Constructing a DCM list for a DTD.

Input: A DTD, D .

Output: A list of all DCMs of D . For each DCM in the list, the list entry contains:

1. the name of the element.
2. an index that distinguishes the DCM from other DCMs for the same element.
3. the applicable inclusions and applicable exclusions.
4. The children of the DCM in the DCM tree traversed by this algorithm.

Method: Use the recursive function `dcm_tree` shown in Table 7.2.

Table 7.2. Construct a DCM list for a DTD.

The initial call to `dcm_tree()` is made with the name of the DOCTYPE element and the inherited exceptions are `{}`. `dcm_tree()` returns the index of the current DCM.

D - the input DTD

`dcm_list` -

`dcm_list.name` - the name of the element of the DCM
`dcm_list.index` - an index that distinguishes the DCM for the element from other DCMs of the same element
`dcm_list.inclusions` - applicable inclusions
`dcm_list.exclusions` - applicable exclusions
`dcm_list.children[]` - list of the children of the DCM in the tree; entries in the list are an element name and an index that identifies the DCM.

`dcm_list` is initialized to NULL.

`dcm_tree (A, inherited exceptions)`

A - the name of the current element
inherited exceptions - inherited_inclusions and
inherited_exclusions

BEGIN

If A has declared content

{ `insert_dcm` adds the DCM to the list if not already there and returns 1 as the index - the applicable exceptions and the child list are null }

`index = insert_dcm(A, {}, {}, {})`

`return(1)`

else

{ A is defined in D by a content model - construct the applicable exceptions - the local exceptions are globally available in D }

`appl_inclusions = inherited_inclusions union`
`local inclusions`

`appl_exclusions = inherited_exclusions union`
`local exclusions`

{ look up the index of the DCM in `dcm_list[]`. If not already in the list, `index=NULL` }

`index=lookup_dcm(A,appl_inclusions,appl_exclusions)`

if `index` is not NULL

{ the DCM has already occurred in the tree }
`return(index)`

else

{ if `index=null` the DCM has not already occurred }

{ derive the effective exceptions }

`eff_inclusions = appl_inclusions - appl_exclusions`

Table 7.2 continued on next page

Table 7.2 (cont.). Construct a DCM list for a DTD.

```

-----
    eff_exclusions = appl_exclusions
    { determine the set of all possible subelements of
      A in the current context - GIs() returns the GIs
        in the model group for A in D}
    subelements = (GIs() U eff_inclusions)
                  - eff_exclusions
    if subelements = {}
        child_list = NULL
    else
        { construct a list of subelements of the DCM,
          each list entry is an ordered pair (name of
            child, index of child)
          for each element, child, in subelements
              child_index = dcm_tree(child,appl_inclusions,
                appl_exclusions)
              insert_child(child_list, child, child_index)
          end for
        end else
        { now insert the DCM into the list - the index
          assigned is the next available index for a DCM
            for A }
        index = insert_dcm(A, appl_inclusions,
          appl_exclusions, child_list)
        return(index)
    end else
end else
END
-----

```

The function, `dcm_tree()` must halt because there are a finite number of DCMs for each DTD, D, and each path in the tree is terminated when a DCM is encountered that is already in the tree. By the definitions and assumptions for exceptions and by the definition of DCMs, the tree contains all of the DCMs in D and only DCMs in D. The DCMs enumerate all possible ways that exceptions can effect model groups in D, because every possible set of effective exceptions can be derived from some set of applicable exceptions.

Example 7.4. Constructing a DCM tree for a DTD D. Consider the DTD, D:

```

<!DOCTYPE  A  [
<!ELEMENT  A  - -  (B | C)  +(X) >
<!ELEMENT  B  - -  (A | #PCDATA)  -(X) +(C) >
<!ELEMENT  C  - -  (B)  -(C) >
<!ELEMENT  X  - -  "EMPTY" >

```

The DCM tree for D is shown in Figure 7.1. The applicable exceptions are not shown for nodes that are repeated occurrences of other DCMs in the tree.

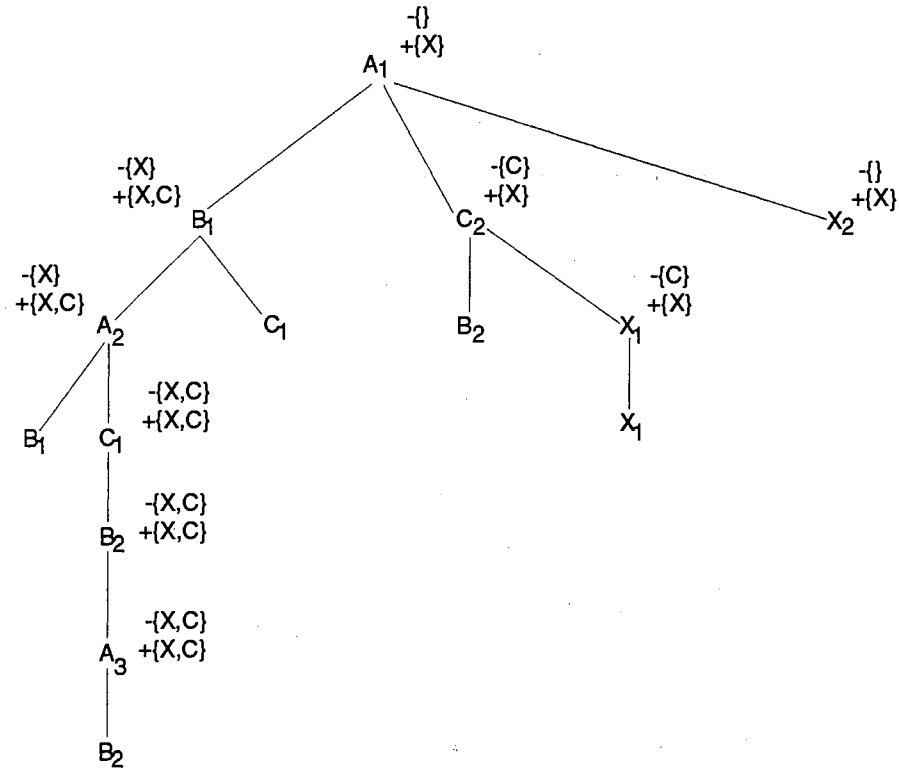


Figure 7.1. The DCM tree for Example 7.4.

The following example illustrates why it is necessary to define DCMs in terms of the applicable exceptions rather than in terms of the effective exceptions.

Example 7.5. A DCM tree for a DTD. Consider the DTD, D:

```

<!DOCTYPE A [
<!ELEMENT A - - (B | D) >
<!ELEMENT B - - (C) + (X) >
<!ELEMENT C - - (A) - (X) >
<!ELEMENT D - - (E) + (X) >
<!ELEMENT E - - "EMPTY" >
<!ELEMENT X - - (#PCDATA) - (X) > ]>

```

The DCM tree for D is shown in Figure 7.2. If DCMs were defined by the effective exceptions, then because A_1 and A_2 have the same effective exceptions, the leftmost path in the tree would terminate at A_2 , and thus the DCMs D_1 and B_2 would not be included in the tree. Note that E_1 has no exceptions because it has declared content of "EMPTY".

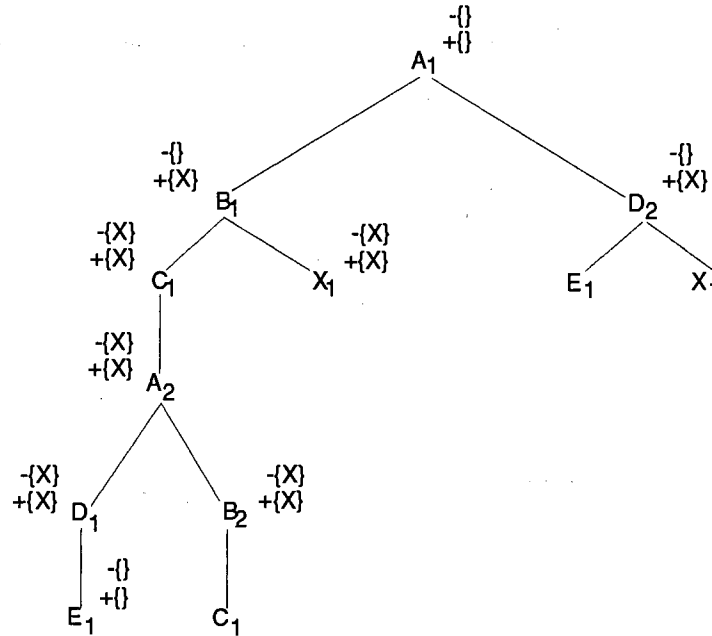


Figure 7.2. The DCM tree for Example 7.5.

7.3 A System of Finite Automata for Document Instances

The DCM list output from Algorithm 7.1 can be used to construct a system of finite automata for D when exceptions are considered. If S is a system of finite automata recognizing $L(D)$ constructed by Algorithm 6.3, then a system of finite automata S' can be constructed from S, such that S' recognizes $L(D)$ when exceptions are considered. This is implemented in Algorithm 7.2, which constructs S' by creating one NFA for each DCM for each element in D. The NFA for each DCM is created by replicating the NFA in M of S for the element of the DCM. These NFAs are then modified to recognize the language

defined by the model group of the element, when the effective exceptions (derived from the applicable exceptions) are applied..

ALGORITHM 7.2. Constructing a system of finite automata for $L(D)$.

Input: A DTD, D , and the DCM list for D from Algorithm 7.1.

Output: A system of finite automata, S' , such that $L(S')=L(D)$ when exceptions are considered.

Method:

Step 1: Construct a system of finite automata, S , recognizing $L(D)$ using Steps 1-2 of Algorithm 6.3. Do not remove useless states with Step 3 of Algorithm 6.3.

Step 2: For each element A defined in some element declaration of D , if A has declared content, then there is one DCM for A and there are no applicable exceptions and no children for A . Add the name A_1 to N' of S' , add the NFA for A in M of S to M' of S' , and let A_1 be the name of the NFA. If A is defined by a content model (model group), then for each DCM for A in the DCM list do the following:

- a. For every element B in the effective (applicable) exclusions of the DCM for A , if B does not occur on some arc in the NFA for A , do nothing. If B occurs on some arc in the NFA for A remove all states entered by a transition on A . By Lemma 6.4, for every state in the NFA, the state is entered by transitions on at most one input symbol in Σ of the NFA. This removes all of the strings containing B and only strings containing B from $L(m)$, where m is the model group for A .
- b. Derive the effective inclusions for the DCM from the applicable inclusions minus the applicable exclusions. For every element B in the effective inclusions for A , add a new state r for each state q in the NFA for A , except for the final state entered by a transition on $\langle/A\rangle$ and for the start state if the omitted tag minimization is minus. Add the following transitions to the NFA for each pair of states, q and r : 1. $\delta(q,B)\rightarrow r$. 2. $\delta(r,B)\rightarrow r$. 3. for all p in Q and c in Σ of the NFA, such that $\delta(q,c)\rightarrow p$, add $\delta(r,c)\rightarrow p$ (except for the case when q is the start state and c is the begin tag). This construction

allows a B to occur anywhere in the content of an A defined by the DCM. It does not allow a B element to precede the start tag or follow the end tag.

- c. For the modified NFA for A created in steps a-b, name the NFA A_i where i is the index of the DCM for A, and add the name A_i to N' , the names of the NFAs of S' . In the NFA for A, replace each element B in N of S, with B_j , where j is the index of B in the child list of the DCM for A. To replace B with B_j do the following: add B_j to Σ of the NFA, and for all states q and p replace all occurrences of $\delta(q,B) \rightarrow p$ with $\delta(q,B_j) \rightarrow p$.

Step 3: Let M'_0 of S' be A_1 , where A is the DOCTYPE element of D and let $\Sigma' = \Sigma$ of S. M' and N' of S' are constructed in Step 2. Q' , Γ' , and Δ' are completely defined by M' .

Step 4. Remove all useless states from S' using Algorithm 3.6.

Example 7.6. A system of finite automata constructed by Algorithm 7.2. Consider the DTD, D:

```
<!DOCTYPE A [
<!ELEMENT A O - (B | C) >
<!ELEMENT B O - (A) - (C) + (X) >
<!ELEMENT C - O "EMPTY" >
<!ELEMENT X - - (#PCDATA) - (X) > ]>
```

The DCM tree constructed for D by Algorithm 7.1 is shown in Figure 7.3.

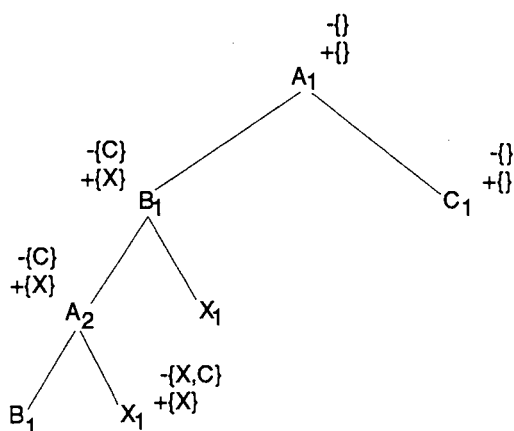


Figure 7.3. The DCM tree for Example 7.6.

The system of finite automata, S , constructed for D when exceptions are not considered by Steps 1-2 of Algorithm 6.3 is:

1. $N = \{A, B, C, X\}$
2. $\Sigma = \{\langle A \rangle, \langle /A \rangle, \langle B \rangle, \langle /B \rangle, \langle C \rangle, \langle X \rangle, \langle /X \rangle, \text{data characters}\}$
3. $M_0 = A$
4. Q, Γ , and Δ are completely defined by the NFAs of M as shown in Figures 7.4-7.7.

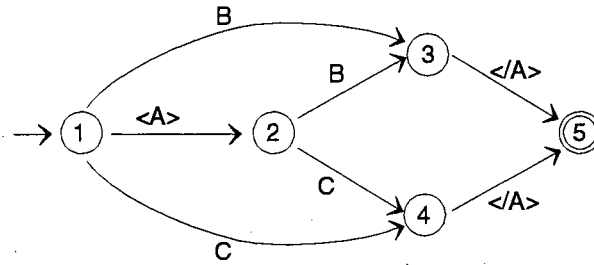


Figure 7.4. The NFA for A in Example 7.6

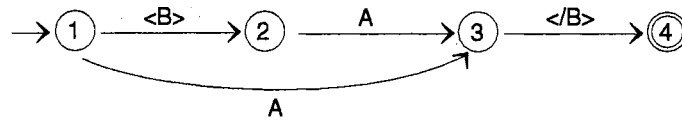


Figure 7.5. The NFA for B in Example 7.6

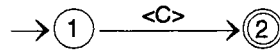


Figure 7.6. The NFA for C in Example 7.6



Figure 7.7. The NFA for X in Example 7.6

The system of finite automata, S' , constructed by Algorithm 7.2 for D .

1. $N' = \{A_1, A_2, B_1, C_1, X_1\}$
2. $\Sigma' = \{\langle A \rangle, \langle /A \rangle, \langle B \rangle, \langle /B \rangle, \langle C \rangle, \langle X \rangle, \langle /X \rangle, \text{data characters}\}$
3. $M_0' = A_1$
4. Q' , Γ' , and Δ' are completely defined by the NFAs of M' as shown in Figures 7.8-7.10.

The NFAs for C_1 and X_1 are shown by Figures 7.6 and 7.7 respectively. They remain unchanged except for the name of the NFA, because C has declared content and X has no GI's in its model group and no effective inclusions.

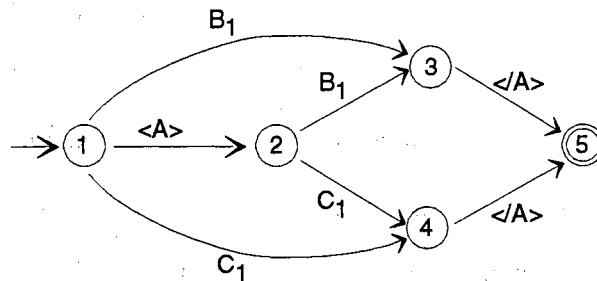


Figure 7.8. The NFA for A_1 in Example 7.6.

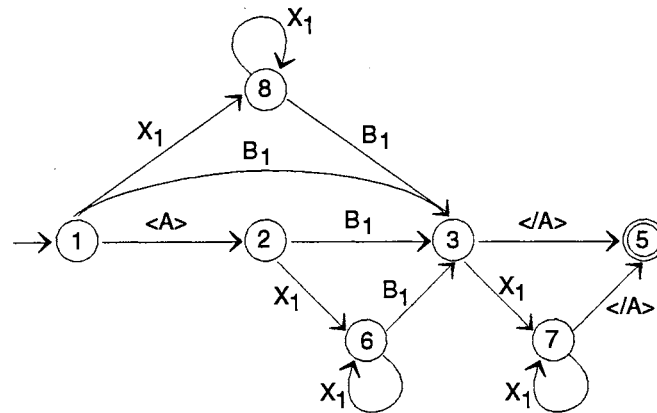


Figure 7.9. The NFA for A_2 in Example 7.6.

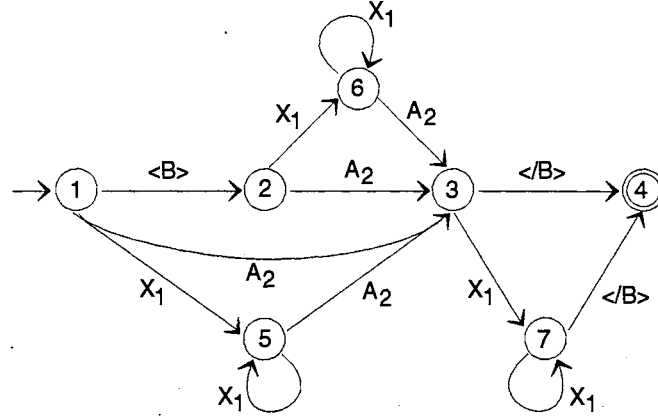


Figure 7.10. The NFA for B_1 in Example 7.6.

Note that the following document instance is in $L(S)$ but is not in $L(S')$.

```

<A>
<B>
<A>
<C>
</A>
</B>
</A>

```

7.4. Exceptions and Ambiguity

Exceptions do not affect Definition 5.3 for ambiguous model groups without lookahead. Although the DCMs constructed by Algorithm 7.1 may be useful for extending this definition, this is not considered in this dissertation.

Exceptions are not excluded from Definition 6.6 for DTDs that are ambiguous without lookahead. Examples 7.7-7.9 show that this definition applies when exceptions are considered. The examples also show that if Algorithm 6.4 is modified by replacing the construction of S in Step 1 with the construction of S' (by Algorithm 7.2), then the modified algorithm detects ambiguous DTDs by omitted tags under Definition 6.6.

Example 7.7. A DTD that is ambiguous for S and still is ambiguous for S' . Consider Example 7.6. The ALA(1) tree constructed for $(A,1)$ of S is shown in Figure 7.11; it shows that D is ambiguous when exceptions are not considered. There are two paths in the tree to leaf nodes for $\langle A \rangle$ with different sequences of labels, $\langle A \rangle \langle B \rangle$ and e .

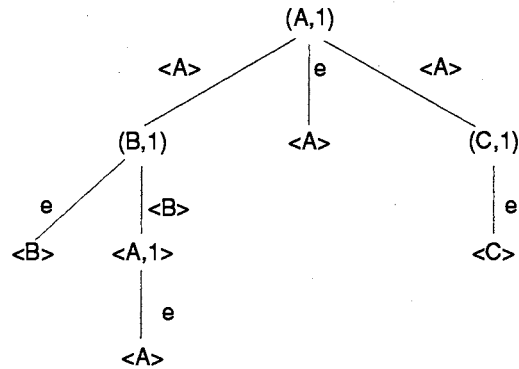


Figure 7.11. The ALA(1) tree for $(A,1)$ in Example 7.6.

The ALA(1) tree for $(A_1,1)$ in Figure 7.12 shows that D is ambiguous when exceptions are considered. There are still two paths to $\langle A \rangle$ with sequences of labels $\langle A \rangle \langle B \rangle$ and e .

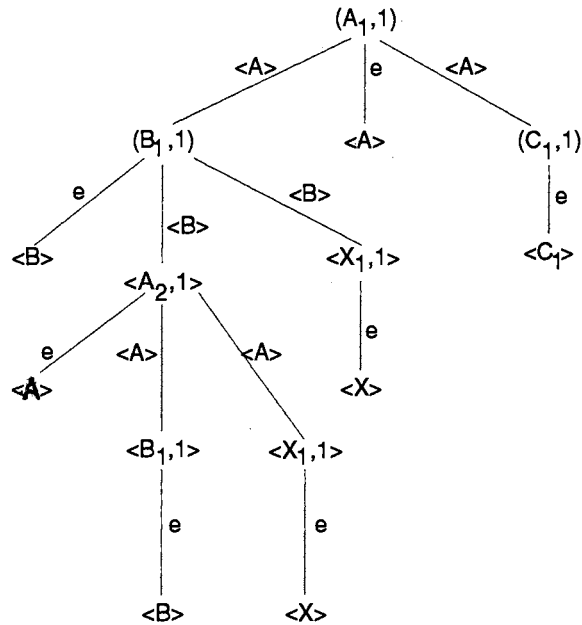


Figure 7.12. The ALA(1) tree for $(A_1,1)$ in Example 7.6.

Example 7.8. This example illustrates that inclusions can introduce ambiguity caused by omitted tags into a DTD. Consider the DTD, D:

```
<!DOCTYPE A [
  <!ELEMENT A - - (B) +(X) >
  <!ELEMENT B O - (C) -(X) >
  <!ELEMENT C - O "EMPTY" >
  <!ELEMENT X O - (C) -(X) > ]>
```

When exceptions are not considered, the system of finite automata, S, constructed for D by Algorithm 6.3 is:

1. $N = \{A, B, C, X\}$
2. $\Sigma = \{\langle A \rangle, \langle /A \rangle, \langle B \rangle, \langle /B \rangle, \langle C \rangle, \langle X \rangle, \langle /X \rangle\}$
3. $M_0 = A$
4. $Q, \Gamma,$ and Δ are completely defined by the NFAs of M as shown in Figures 7.13-7.15.

The NFA for X is shown in Figure 7.16, although it is not in M, because it is removed by Algorithm 6.3 as an inaccessible NFA. D is not ambiguous by omitted tags (Definition 6.6) when exceptions are not considered.

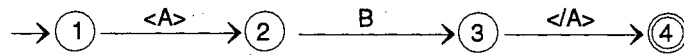


Figure 7.13. The NFA for A in Example 7.7.

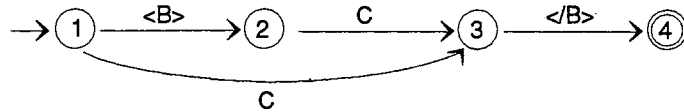


Figure 7.14. The NFA for B in Example 7.7.

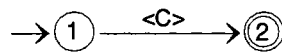


Figure 7.15. The NFA for C in Example 7.7.

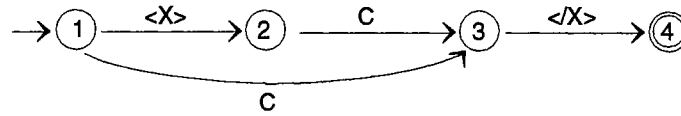


Figure 7.16. The NFA for X in Example 7.7.

The DCM tree constructed for D by Algorithm 7.1 is shown in Figure 7.17.

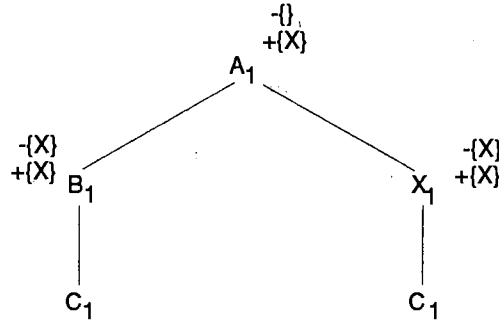
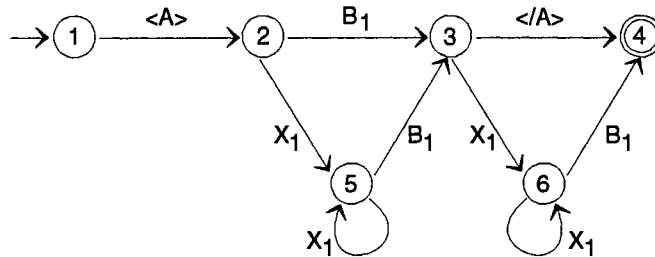


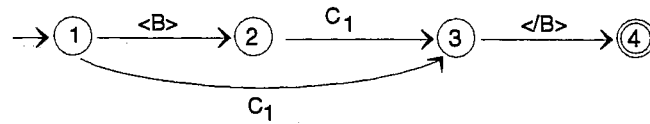
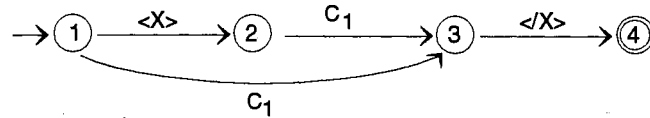
Figure 7.17. The DCM tree for Example 7.7.

The system of finite automata, S' , constructed for D when exceptions are considered by Algorithm 7.2 is:

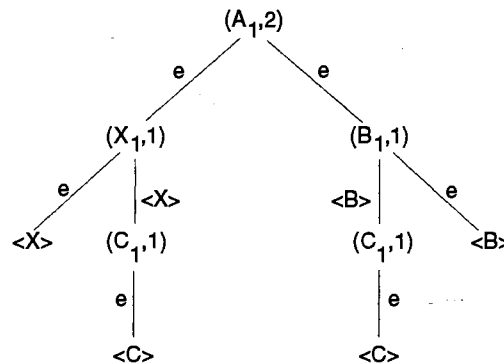
1. $N' = \{A_1, B_1, C_1, X_1\}$
2. $\Sigma' = \{\langle A \rangle, \langle /A \rangle, \langle B \rangle, \langle /B \rangle, \langle C \rangle, \langle X \rangle, \langle /X \rangle, \text{data characters}\}$
3. $M_0' = A_1$
4. Q' , Γ' , and Δ' are completely defined by the NFAs of M' as shown in Figures 7.18-7.20.

The NFA for C_1 and remains unchanged from the NFA for C shown in Figure 7.15.

Figure 7.18. The NFA for A_1 in Example 7.7.

Figure 7.19. The NFA for B_1 in Example 7.7.Figure 7.20. The NFA for X_1 in Example 7.7.

D is ambiguous by omitted tags when exceptions are not considered. When S' is constructed under Algorithm 7.2, the useless NFA for X in S is not removed. Then, the ALA(1) tree for $(A_1, 2)$ in Figure 7.21 shows that D is ambiguous. There are two paths in the tree to leaf nodes for $\langle C \rangle$ and they have different sequences of labels, $\langle X \rangle$ and $\langle B \rangle$.

Figure 7.21. The ALA(1) tree for $(A_1, 2)$ in Example 7.7.

Example 7.9. When exceptions are not considered the following DTD is ambiguous by omitted tags without lookahead. When exceptions are considered, it is not ambiguous.

```

<!DOCTYPE A [
  <!ELEMENT A - - (B | C) - (C) >
  <!ELEMENT B O - (C | D) >
  <!ELEMENT C - O "EMPTY" >
  <!ELEMENT D - O "EMPTY" > ] >
  
```

7.5. Exception Validation

Clause 11.5.2.1 of the standard specifies that it is an error to try to remove an element that is required: "Exclusions modify the affect of model groups to which they apply by precluding options that would otherwise have been available...It is an error if an exclusion attempts to modify the affect of model groups in any other way..." SGML parsers are not required to report this error unless they specify YES for the VALIDATE EXCEPTIONS feature. It is easy to detect occurrences of this in single element declarations such as

```
<!ELEMENT A - - (B, D, E+) -(D) >
```

The element D is required and thus it is an error to try to exclude it. However, as illustrated by Algorithm 7.1, applying only local exceptions is not sufficient for a complete implementation of the VALIDATE EXCEPTIONS feature as defined in the standard. No methods have been shown for implementing this feature while parsing the DTD.

The DCMs for a DTD constructed by Algorithm 7.1 can be used for a complete implementation of the VALIDATE EXCEPTIONS features as follows:

1. Construct a DCM tree for the DTD.
2. For each DCM apply the applicable exclusions of the DCM (the effective exclusions), to the NFA in M of S for the DCM element, using Step 2.a of Algorithm 7.2.
3. For the NFA derived in Step 2.a (after the excluded elements have been removed), use Algorithm 5.2.b to remove the inaccessible states of the NFA. If there is no final state remaining in the NFA, a required element has been removed, and an exception error has occurred. Note that when this occurs the NFA is useless by Definition 3.4. Thus, it will be removed by Step 4 of Algorithm 7.2 that removes useless states..

This chapter shows how exceptions can affect $L(D)$. An algorithm is shown for enumerating the ways that exceptions can affect model groups in a DTD. Then a system of finite automata is constructed recognizing $L(D)$ when exceptions are considered, and it is shown by example that Algorithm 6.4 for detecting DTDs that are ambiguous by omitted tags still applies.

8. *Conclusions and Future Work*

8.1. Conclusions

Although SGML is the first standard for text processing and document design to gain widespread acceptance, there is one factor that is limiting its success: the lack of a formal language model. This is described by Sperberg-McQueen in [21] as follows: "But the biggest problem we face, I think, is that we need a clear formulation of a formal model for SGML. If we get such a formal model, we will be able to improve the strength of SGML in several ways."

This dissertation provides a formal model for SGML; systems of regular expressions and a class of recognizers, systems of finite automata, are defined. These are variations of language specifications found in the literature [15,26]. An algorithm is presented for constructing a system of automata, S , from a system of regular expressions, R , such that $L(S)=L(R)$. This language model applies to SGML in two ways.

A system of finite automata is constructed from the syntax productions that define DTDs, and then from this system of finite automata a parser is constructed for DTDs. Using this method, it is probable that a parser can be constructed for the entire system of syntax productions that define SGML; this includes the hierarchical components above DTDs, as well as the components of DTDs that are not considered in this dissertation because they are not relevant to the high level syntax of documents defined by DTDs. There are no descriptions in the literature of this method for parsing SGML.

The language model also applies to the set of documents defined by a DTD, the document instances. Algorithms are shown for constructing a system of finite automata recognizing the document instances defined by a DTD (when all required components of DTDs that affect the high level syntax of document instances are considered). This is the first description in the literature for including the exceptions and omitted tag minimization features of SGML in a static language model.

The definitions of ambiguity in DTDs in Clauses 11.2.4.3 and 7.3.1 of the standard are not precise, and no complete methods have been shown for detecting ambiguity as prohibited by Clause 7.3.1. The lack of a formal language model for DTDs is the primary reason that these problems have not been solved.

An implementation independent definition is given for ambiguous model groups that is derived from [6] and is modified to conform closely to Clause 11.2.4.3. Based on this definition, an algorithm is given for detecting ambiguous model groups. The algorithm is generalized and is easier to understand than existing methods for detecting ambiguous model groups [6,16]. Two specific methods are shown that significantly reduce the space required during construction of the NFA, and due to the generality of the algorithm the resulting NFA may be optimized for space [3].

Ambiguity caused by omitted tags is prohibited by Clause 7.3.1, but the term *ambiguity* in the clause is not defined. Ambiguous documents are prohibited as opposed to prohibiting ambiguous DTDs. A set of rules is given for preventing ambiguous documents, but the rules are incomplete and unnecessarily restrictive as shown in [25]. Even if a complete set of rules could be defined and applied while creating documents, this would not be sufficient for the domain SGML is designed to serve; the responsibility for correcting ambiguities would be on data entry personnel who may have the least technical training [9,11,12]. A definition is given for DTDs that are ambiguous by omitted tags, and an algorithm is shown for detecting this kind of ambiguity while parsing the DTD. This algorithm solves a significant problem in implementing SGML, preventing ambiguity in the DTD design phase [9,11,12,17]. It is the first method shown for detecting this kind of ambiguity.

The algorithms for removing useless and inaccessible NFAs from a system of finite automata can be used to show which elements declared in the DTD do not affect the language defined by the DTD. The algorithm for enumerating the ways that exceptions affect model groups in a DTD can be used for DTD analysis and to implement the

VALIDATE EXCEPTIONS feature of SGML. Thus, these algorithms will be useful tools in DTD design and verification. No methods have been shown in the literature for solving either of these problems.

The two revised definitions of ambiguity formalize the existing definitions, they maintain consistency with them where feasible, and they are independent of any particular language model. They are also independent of each other, and they clearly distinguish the kinds of ambiguity that can occur in DTDs as a property of the DTD component for which they are defined.

The language model, the revised definitions of ambiguity, and the algorithms for detecting ambiguity resolve important open questions regarding the implementation of SGML. Thus, the results of this dissertation will be useful to the publishing industry and to the government which have a significant initial investment in SGML applications.

8.2. Recommendations for Future Work

In Chapter 4 a parser is constructed for DTDs from the system of finite automata for DTDs. This should be investigated to see if an algorithm can be developed for constructing parsers for document instances from the systems of finite automata constructed for document instances.

The algorithms in this dissertation were developed to show solvability. The analysis of the complexity of these algorithms is considered as future work.

References

1. Aho A. V., Hopcroft J. E., and Ullman J. D. *Data Structures and Algorithms*. Addison-Wesley, 1983.
2. Aho A. V., Sethi R., and Ullman J. D. *Compilers: Principles, Tools, and Techniques*. Addison-Wesley, 1986.
3. Aho A. V. and Ullman J. D. *The Theory of Parsing, Translation, and Compiling Volume 1: Parsing*. Prentice-Hall Inc., N.J., 1972.
4. Book R., Even S., Greibach S., and Ott G. Ambiguity in Graphs and Expressions. *IEEE Transactions on Computers*. Volume C-20(2) (February 1971), pp. 149-153.
5. Brueggemann-Klein A. Regular Expressions into Finite Automata. *Proceedings of LATIN '92: 1st Latin American Symposium on Theoretical Informatics*. (April 6-10, 1992, S'ao Paulo, Brazil), Springer-Verlag, Berlin, 1992, pp. 87-98.
6. Brueggemann-Klein A. Unambiguity of SGML Content Models. working paper. Universitat Freiburg, Institut fur Informatik, August, 1992.
7. Bryan M. *SGML: An Author's Guide to the Standard Generalized Markup Language*. Addison-Wesley, 1988.
8. Cover R. Annotated Bibliography and List of Resources Standard Generalized Markup Language ISO 8879:1986 (SGML). Version 2.1, Revised February 1992, <TAG> *The SGML Newsletter*. Volume 5, (3-5), (March-May 92).
9. Davis W. W. OMITTAG Minimization. <TAG> *The SGML Newsletter*. Volume 5(2) (Feb 92), pp. 4-5.
10. Goldfarb C. F. *The SGML Handbook*. Clarendon Press, Oxford, 1990.
11. Graf J. M. Ambiguity in the Instance. <TAG> *The SGML Newsletter*. Issue 7, (1988), pp. 6-9.
12. Heath J. and Welsh L. Difficulties in Parsing SGML. *Proceedings of ACM Conference on Document Processing Systems*, (Dec 5-9, 1988, Santa Fe, New Mexico), ACM, New York, 1988, pp. 71-77.
13. International Standard ISO 8879 Information Processing - Text and office systems - Standard Generalized Markup Language (SGML), International Organization for Standardization, Switzerland, 1986.
14. Johnsonbaugh R. *Discrete Mathematics*. Macmillan, N.Y., 1990.
15. Lalonde W. R. Regular Right Part Grammars and Their Parsers. *Communications of the ACM*, Volume 20(10) (October 1977), pp. 731-741.
16. Matzen R. M., George K. M., and Hedrick G. E. A Model for Studying Ambiguity in SGML Element Declarations. *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*. (February 14-16, Indianapolis, Indiana), ACM, N.Y., 1993, pp. 668-676.

17. McFadden J. R. and Wilmott S. Ambiguity in the Instance: An Analysis. <TAG> *The SGML Newsletter*. Issue 9 (March-April 89), pp. 3-5
18. Price L. A. Graphic Representation of Content Models. <TAG> *The SGML Newsletter*. Issue 10 (July 89), pp. 12-16.
19. Price L. A. and Schneider J. Evolution of an SGML Application Generator. *Proceedings of ACM Conference on Document Processing Systems*, (Dec 5-9, 1988, Santa Fe, New Mexico), ACM, New York, 1988, pp. 51-60.
20. Requests for contributions for review of ISO 8879 (SMGL). <TAG> *The SGML Newsletter*. Volume 5(2) (Feb 92), pp. 9.
21. Sperbeg-McQueen C. M. Closing Remarks at SMGL '92: the quiet revolution. notes from the closing address given at the SGML '92 conference, (October 25-29, 1992, Boston MA, Graphics Communications Association), posted on Internet newsgroup: comp.text.sgml, November 2, 1992, message-ID: <92307.171247U35395@uicvm.uic.edu>.
22. The Innerview DTD. TMS Inc., Stillwater, OK, 1992.
23. van Herwijnen E. *Practical SGML*. Kluwer Academic Publishers, 1990.
24. Waldt D. C. The Inclusion and Exclusion Confusion. <TAG> *The SGML Newsletter*. Issue 12 (December 89), pp. 1-5.
25. Warmer J. and van Egmond S. The Implementation of the Amsterdam SGML Parser. *Electronic Publishing*, Volume 2(2) (July 89), pp. 65-90.
26. Woods W. A. Transition Network Grammars for Natural Language Analysis. *Communications of the ACM*, Volume 13(10) (October 1970), pp. 591-606.

²
VITA

Richard W. Matzen

Candidate for the Degree of

Doctor of Philosophy

Thesis: A FORMAL LANGUAGE MODEL FOR DETECTING
AMBIGUITY IN SGML

Major Field: Computer Science

Biographical:

Personal Data: Born in Rochester, New York, May 12, 1948, the son of Walter T. and Virginia Matzen.

Education: Graduated from Richardson High School, Richardson Texas, in May 1966; Received Bachelor of Science Degree in Computer Science with math minor from the University of Central Arkansas in August, 1984; Received Master of Science Degree in Computer Science from Oklahoma State University in August 1987; Completed requirements for Degree of Doctor of Philosophy, Oklahoma State University, December 1993.

Academic Honors: Presidents Honor Roll and Dean's List, University of Central Arkansas, 1982-1984; McAlester Scottish Rite Fellowship, Oklahoma State University, 1985-1986; Phi Kappa Phi, Oklahoma State University, 1987.

Professional Experience: Teaching Assistant, Department of Computer Science, Oklahoma State University, August 1984 to May 1987; Software Engineer, TMS Inc., Stillwater Oklahoma, February 1987 to present.

Professional Organizations: member, Association for Computing Machinery (ACM); member, ACM Special Interest Group for Applied Computing.