



GMD Research Series

GMD –
Forschungszentrum
Informationstechnik
GmbH

Yi Xu

An Incremental Approach to Document Structure Recognition

© GMD 1998

GMD – Forschungszentrum Informationstechnik GmbH
Schloß Birlinghoven
D-53754 Sankt Augustin
Germany
Telefon +49 -2241 -14 -0
Telefax +49 -2241 -14 -2618
<http://www.gmd.de>

In der Reihe GMD Research Series werden Forschungs- und
Entwicklungsergebnisse aus der GMD zum wissenschaftlichen, nicht-
kommerziellen Gebrauch veröffentlicht. Jegliche Inhaltsänderung des
Dokuments sowie die entgeltliche Weitergabe sind verboten.
The purpose of the GMD Research Series is the dissemination
of research work for scientific non-commercial use.
The commercial distribution of this document is prohibited,
as is any modification of its content.

Anschrift der Verfasserin/Address of the author:

Yi Xu
debis Systemhaus
Göbelstraße 1-3
D-64293 Darmstadt
E-mail: yxu@debis.com

Die vorliegende Veröffentlichung entstand im/

The present publication was prepared within:

Institut für Integrierte Publikations- und Informationssysteme (IPSI)
Integrated Publication and Information Systems Institute
<http://www.darmstadt.gmd.de/ipsi>

Die Deutsche Bibliothek - CIP-Kurztitelaufnahme:

Xu, Yi:

An incremental approach to document structure recognition /
Yi Xu. GMD – Forschungszentrum Informationstechnik GmbH. -
Sankt Augustin : GMD – Forschungszentrum Informationstechnik, 1998
(GMD Research Series ; 1998, No. 16)
Zugl.: Darmstadt, Techn. Univ., Diss., 1998
ISBN 3-88457-340-3

ISSN 1435-2699

ISBN 3-88457-340-3

Abstract

Keywords: document structure recognition and machine learning

Most of the electronic documents available from today's huge number of electronic information sources have an implicit structure. In order to manipulate, exchange, and archive these documents, it is important to extract their logical structure and to make it explicitly available.

Many researches have noted the importance of document logical structure recognition, yet we still lack an easy method for recognizing the implicit structure of electronic documents. The two most widely used methods are: recognizing structure by hand, or through structure recognition programs. Due to the large number of documents, the manual approach is tedious and error-prone although in principle it is very simple. Writing a complete recognition program is much more effective, but it requires significant intellectual effort. To combine the advantages of both methods, this thesis presents an approach to automate the learning of recognition grammars from manually structured examples.

The approach uses two techniques from the field of machine learning: Version space – to abstract from the concrete contents of the structured examples in order to recognize examples with different content, and grammatical inference – to generalize the syntactic structure of the structured examples in order to recognize examples with slightly deviating structure. These two techniques are embedded into an incremental structure learning system – *MarkItUp!* – which allows for a convenient refinement of a recognition grammar towards new examples with unanticipated structure.

This dissertation presents the design, analysis, and implementation of *MarkItUp!*. The characteristics of *MarkItUp!* are as follows. (1) it supports a simple way for the user to obtain a suitable recognition grammar; (2) it uses incremental learning so that the recognition grammar can be efficiently modified using additional structured examples. Experimental results on combining the version-space method with a grammatical inference approach in the learning cycle are also presented.

Kurzfassung

Schlüsselworte: Dokumentstrukturerkennung und maschinellen Lernen

Verschiedene elektronische Informationsquellen bieten ihre Dokumente in unterschiedlicher Form an. Insbesondere ihre Struktur ist oft nur in anbieterspezifischem Format verfügbar. Für die weitere Bearbeitung, den Austausch und die Archivierung muß diese Struktur extrahiert werden. Diese Dissertation entwickelt einen Ansatz zur automatischen Erkennung der Struktur von elektronischen Dokumenten auf Basis von nur wenigen, manuell strukturierten Beispieldokumenten. Dazu wird eine regel-orientierte Sprache zur Spezifikation von Erkennungsprogrammen eingeführt. Auf dieser Basis werden Techniken des maschinellen Lernens – Versionsraum und Grammatik-Inferenz – entwickelt, die Erkennungsprogramme aus Beispielen generieren.

Acknowledgements

I am most grateful to my advisor, Prof. Dr. Erich J. Neuhold, for providing me the research opportunity in GMD-IPSI. His insightful comments on this work have been a principal reason for its success. Further more, his feedback has been the major force behind my development as a researcher. I also appreciate my second advisor, Prof. Dr.-Ing. Dr. h.c. José L. Encarnação, for reviewing this dissertation and helpful discussions.

A number of other people deserve special acknowledgements. Peter Fankhauser, my group leader, guided me into a new research area. Discussions with him helped me understand what it was that I was doing. Discussions with Helena Ahonen, Ralph Busse, Bertin Klein, and especially Weimin Chen helped me make my thesis clear, even when I thought it was already clear.

I would like to thank Lothar Rostek who supports a good environment of SMALLTALK which helped to code my implementation successfully. I also appreciate the kind help of Ute Sotnik, Andreas Stenger, Ute Kischel, Elisabeth Trautrim, Peter Schoendorf, and Ernst Mink during our years at GMD-IPSI.

This work was done with the financial and technical support of GMD-IPSI. The support by the Computer Science Department, Darmstadt University of Technology is greatly appreciated.

I must thank Prof. Longxiang Zhou who recommended GMD-IPSI to me, whose encouragement over the years made the completion of this work possible. Final thanks go to my parents, whose understanding and support over 10 thousand kilometers away made this work possible.

Table of Contents

Chapter 1 Introduction	1
1.1 Problem Domain and Overall Goals	1
1.2 Overall Approach	3
1.2.1 Low-Level Recognition	4
1.2.2 High-Level Recognition	4
1.2.3 Learning by Marking up	5
1.3 Contribution	6
1.4 A Guide to this Dissertation	7
Chapter 2 Preliminaries	9
2.1 Formal Languages	9
2.1.1 Strings and Languages	9
2.1.2 Regular Expressions and Finite Automata	10
2.1.3 Binary Relation of Regular Expressions	12
2.2 Graphs	12
2.2.1 Directed Graphs and Undirected Graphs	13
2.2.2 Directed Acyclic Graphs	13
2.2.3 Permutation Graphs	14
2.3 Document Structures	15
2.3.1 Syntactic Structures	15
2.3.2 Layout Structure	15
2.3.3 Logical Structure	16
2.3.4 Relations Between Layout and Logical Structure	16
2.4 Markup	17
2.5 Document Description Language – SGML	18
2.5.1 SGML Markup	19
2.5.2 SGML DTD	19
2.5.3 SGML Parser	20
2.5.4 The Marking Up (Tagging) Process	21
Chapter 3 An Approach to Document-Structure Recognition	23
3.1 DREAM	23
3.1.1 Structure Description of a DSD	24
3.1.2 Recognition Styles of DSDs	26
3.1.2.1 Regular Expressions in DSDs	26
3.1.2.2 Functions in DSDs	27
3.1.3 A Complete Example of a DSD	28
3.2 System Overview of MarkItUp!	29
3.2.1 Starting the MarkItUp! System	31
3.2.2 Structure Editor	31
3.2.3 Scanner	37
3.2.4 Learning	37

3.3 Demonstration of MarkItUp!	38
3.4 Summary	43
Chapter 4 Learning	45
4.1 Learning Problems and Learning Levels	45
4.2 Learning at Content Level	46
4.2.1 Goals, Problems and Overall Approach	46
4.2.2 Concepts and Binary Relation	47
4.2.3 Determining Concepts	48
4.2.4 Ordering Concepts	49
4.2.4.1 Concept Base	49
4.2.4.2 Linear Ordering List of Concepts	50
4.2.5 Learning from Strings	51
4.2.5.1 Learning from Copy-Strings	52
4.2.5.2 Learning from Cut-Strings	55
4.2.5.3 Examples for String Abstraction	55
4.3 Learning at Structure Level	56
4.3.1 Goals, Problems and Overall Approach	56
4.3.2 Representation of Document Logical Structure	57
4.3.3 Learning Logical Structure by Rewrite Rules	58
4.3.3.1 Unification and Simplification Rules	59
4.3.3.2 Abstraction Rules	62
4.3.4 Applying the Learning Rules	64
4.3.4.1 Control Strategies for Applying the Learning Rules	64
4.3.4.2 Some Examples for the Learning Rules	65
4.4 Summary	68
Chapter 5 Sequence of Arbitrary Ordering	69
5.1 Problem and Goal	69
5.2 Basic Concepts and Notations	70
5.3 Inferring a General Expression	71
5.3.1 Constructing a Permutation Graph	71
5.3.2 Union Graph of PGs	72
5.3.3 Deriving a General Expression from an AG	75
5.4 Summary	79
Chapter 6 Implementation	81
6.1 System Architecture of MarkItUp!	81
6.2 User Interface	81
6.2.1 Structure Editor	82
6.2.2 Concept Editor	88
6.3 From Marked-up Example to a Grammar	93
6.4 Implementation of Learning Component	93
6.4.1 Grouping Cut-Strings with Nonterminals	94
6.5 DREAM Grammar Generator	97
6.6 Learning Strategies	99
6.6.1 Fallback Rule	99
6.6.2 Exhaustive vs. Partial Learning	100
6.6.2.1 Learning from an Entire Example	101

6.6.2.2 Learning from a Partial Example	103
6.7 Experimental Evaluation of the System	106
6.8 Summary	109
Chapter 7 Related Work	111
7.1 Editing-By-Example	111
7.1.1 Function Approaches	111
7.1.2 Procedural Approaches	112
7.2 Inductive Learning and Learning Methods	113
7.2.1 Grammatical Inference	113
7.2.2 Version Spaces	115
7.3 Application to Wrapping Semi-Structured Data	116
Chapter 8 Conclusion	119
Bibliography	123
Appendix: List of Figures and Tables	129

Introduction

Today's many electronic information sources offer masses of electronic documents. These documents usually have only *implicit* structures. For further manipulation, exchange and archiving, it is important to *extract* their implicit structure. This dissertation presents an approach to automatically recognize the structure of electronic documents on the basis of a few manually structured example documents. For this purpose this thesis introduces a dedicated language for specifying recognition programs, and shows how machine learning techniques can be used to generate such recognition programs from examples. The developed concepts have been implemented in the framework of the system *MarkItUp!*.

1.1 Problem Domain and Overall Goals

With widely available computers, documents are not only thought as a medium to be printed and read, but also as a structure to be communicated, retrieved, archived in data bases, etc. To take advantage of the already existing tools for the above purposes, two issues have become important: (a) standardized representation of documents, and (b) document structure recognition, with which arbitrary documents can be turned into a standardized form. The first issue is a general goal, and the second issue is a means to accomplish that goal.

Document structure recognition thus aims at extracting information from documents and at converting the extracted information into a representation language which models the original document as accurately and concisely as possible. A document can be generally viewed as having two kinds of structures: a logical structure and a layout structure. The logical structure separates a document into *logical* elements, such as the title and author of a document, while the layout structure consists of *formatting* elements, such as pages, columns, and paragraphs. Document structure recognition refers to both aspects: document logical structure recognition and layout structure recognition. Furthermore, a document can have different representations, for example, it can be represented by paper sheets from a laser printer, or can exist as an electronic source sitting in computer memory or on magnetic devices. The documents existing as electronic sources are machine readable to smaller or larger extent, e.g. pixel representation,

versus explicitly structured and described. With the increasing popularity of the Internet, the number of electronic sources is growing quickly. How to translate a paper form document into a machine-readable document is beyond the scope of this thesis. This thesis focuses on the documents that are already machine readable.

Comprehending a consistently structured document is easier than comprehending an unstructured document since the reader can concentrate on the contents and organization of the document without worrying about its layout. More generally, in a publication cycle, which comprises writing and reading documents, in addition to copying, distributing and archiving of the documents, a structured document has further advantages which are not concerned with creation and editing alone: Because of the high level of abstraction of the document model used, many different kinds of processes can be applied to structured documents. For example, information necessary for a document retrieval system, database systems, hyper-document systems or individualized printed document systems can work efficiently on such documents.

However, document producers usually use different and inconsistent formatting conventions to express the layout structure, even within one source or document. For example, in a bibliographic document, a line starting with either the character “!” or the capital characters “AU” expresses that the content in the line denotes an author name. Furthermore, the first name and the last name of the author could have different ordering, that is, the first name is followed by a blank and by the last name, or the last name is followed by a comma, by a blank, and by the first name. Similar inconsistent structuring conventions can be found in documents retrieved from public databases or received via electronic mails.

Documents can be classified into several *classes* on the basis of their formatting forms, for instance, the bibliography documents can be divided into several different classes since each person has his own habit to format his bibliography document. However a *collection* of a document class consists of many subdocuments with *similar* structures.

To arrive at a consistent standard structure which can be processed by a wide range of applications, it is necessary to recognize such formatting conventions, and to map them into a standardized form. Towards this end the following three problems have to be tackled.

- (a) What kind of document structure should be captured?

The aim of this thesis is to convert documents from their original format, that is, their *layout* structure, to a standard *logical* structure. Thus the focus is on document *logical structure* recognition.

- (b) Which formal description method is to be used for representing the recognized structure?

Apart from the many proprietary description languages used by commercial document production systems, there exist two main standard languages. ODA (*Office Document Architecture*) and SGML [5] (*Standard Generalized Markup Language*). H. Brown [10] discusses similarities and differences between the two description languages. Choosing which one depends on the concrete applications. In this thesis SGML is chosen, because it specifies document structures with well-defined grammars, which can be used as a good basis for implementing machine learning techniques to generate such grammars semi-automatically, as well as appropriate parsers to use these grammars for document structure recognition.

- (c) How can the structure of a document be captured?

The structure of documents can either be captured manually, or automatically by means of recognition programs. Manually structuring a huge number of documents is highly repetitive, cumbersome, time-consuming, error-prone, and expensive. In addition, the documents have different contents and similar structures, and their logical structures are nested structures rather than flat structures. All of these factors increase the complexity of the logical structure recognition for the human. Of course, writing complete recognition programs to recognize the logical structure requires significant intellectual effort. Thus the main goal of this dissertation is to find a semi-automatic approach to bridge the gap between structuring manually and writing a recognition program.

1.2 Overall Approach

Capturing information from documents is the goal of document structure recognition. This ranges from the identification of the layout structure, the recognition of the (logical) structure of documents, and to the (largely domain dependent) extraction of semantic content.

Document structure recognition can be classified along two view points, *document processing* and *recognition level*.

From the viewpoint of document processing, the document structure recognition can be divided into two activities:

- *Document analysis* to extract the geometric (layout) structure from a document;
- *Document understanding* to map the geometric (layout) structure into a logical structure of document.

From the viewpoint of recognition level, two main levels can be distinguished:

- The *low-level recognition* or the document layout structure recognition;
- The *high-level recognition* or the document logical structure recognition.

This thesis discusses both recognition levels, but especially concentrates on high level recognition.

1.2.1 Low-Level Recognition

Low-level recognition aims at obtaining a symbolic representation of the document regarded initially as an image in document pages, e.g. scanned images. This includes decomposing the image into regions of text and non-text by breaking the text regions on the page into text blocks and image blocks, and text blocks into text lines, recognizing characters and words, and identifying the format characteristics such as font type and size.

Most efforts on analyses and transformations of documents have concentrated on the document layout structure recognition, such as separating text and graphics in the documents and recognizing characters in the text [45, 53, 24, 12].

1.2.2 High-Level Recognition

Over the past few years it has become apparent that low-level recognition does not suffice, but also higher levels of recognition are required [33, 43]. For this purpose, the *hierarchical structure* and *sequence relation* in a document are extracted and described by means of a standard high level language. Recognizing this level of structure also can be regarded as the final aim and the last stage of document structure recognition.

In actual production level documents the logical structure is usually expressed by means of layout and format. This information has now to be extracted and transformed, in order to arrive at a coherent logical structure for a particular document. Since there is no one-to-one mapping between layout structure and logical structure, such an extraction is difficult. In addition, for different types of documents and even different description languages of documents' structures, the formatting rules are different.

Practically all approaches aiming at logical structure recognition utilize some form of rule knowledge to specify the relationship between layout and logical structure. For example, Ingold [28] proposes a method for deducing the logical structure from the layout structure of a document, using precise rules that interpret the layout structure in terms of font-information

and geometrical information. Toyoda, Noguchi and Nishimura [48] develop a method for the extraction of articles in Japanese newspapers. For this purpose, they identify six general formatting rules for the layout of Japanese newspapers, and on this basis develop an algorithm for extracting newspaper articles. Another, more flexible approach to recognize the logical structure of documents is implemented by the CAROL system which is an automatic cataloging system to be used in libraries [43]. A printed document (usually the header page of some scientific article) is input into the system by OCR (optical character recognition) in order to derive a well-defined output format with additional layout information. This information is used for recognizing the logical structure of the documents by means of recognition rules. To allow the user to treat different document types, there is a learning mechanism which can generate a set of new rules for a specific document type from examples.

All these approaches focus on the layout information to determine the logical structure. In most cases, however, this does not suffice. Thus these approaches are extended by utilizing content information. In addition, the extended approach in this thesis is not restricted itself to *flat* structures, but uses the full power of SGML to treat also documents with a nested structure.

1.2.3 Learning by Marking up

SGML represents the structure of documents by so-called *markups*, which are dedicated labels splitting the content of a document into its logical elements. The transformation of an electronic document into an SGML-compliant form comprises the following three steps: (1) specifying the document's logical structure, (2) determining the processing rules which will produce the structure desired for the document, and (3) inserting the markups into the document according to these rules.

As stated above, there are two alternatives to carry out these steps. One is to manually structure the documents with a normal editor, on the basis of an initially specified goal structure. The other is to use programs to transform a partially inconsistently formatted document into a consistently structured document.

To a non-programmer structuring documents manually with an editor may seem easier than to program a translator. However, it is highly burdensome to manually structure a large amount of documents. Of course it is productive in the sense that every step makes tangible progress towards the solution of the problem, but it is tedious and repetitive all the same.

Writing recognition programs is not as easy as manually structuring documents. These recognition programs can be implemented by editing macro commands, such as an *emacs macro* [46], or recognition rules, such as FastTag, IMSYS [52], and DREAM (*Document Structure REcognition And Markup*) [20]. Although using macros or recognition rules (a recognition grammar) based on a complete definition of the structure can overcome the above problems, it involves a fair amount of work that requires significant intellectual effort due to structural differences and formatting inconsistencies among the subdocuments. In addition to writing such a grammar, the user has to debug it, parse it, check to see whether it does the right thing, and then debug it again if necessary. Once this development process is completed, the program can be used to convert an arbitrary number of subdocuments of a collection. But writing such a program is not a simple task, and adjusting it directly to new document classes is nearly impossible.

This thesis aims at recognizing the structure of electronic documents (e.g. collections) that have similar *implicit* structures (e.g. BibTeX, electronic mail, folders). For this kind of electronic document collections the manual determination of the structure of a few example subdocuments can be used as a basis to generate recognition programs for structuring the other subdocuments. The system *MarkItUp!* developed in the framework of this dissertation follows exactly this approach, using techniques from machine learning. It provides a structure editor, with which an initial example for a particular collection is manually structured (marked up) by the user. The system accepts the marked-up example and generates a recognition grammar, which can recognize similar examples. On the basis of this grammar the system tries to mark up another example selected from the same collection. The tentatively marked-up example can be accepted or rejected by the user. If it is rejected, because the example deviates from the previous examples and the result of the tentatively marking leads to an undesired consequence, the user corrects this example and asks the system to learn the corrected example. After learning the example, the system synthesizes a new recognition grammar which includes the structural deviations occurring in the new example. After a few such learning steps, the generated grammar usually comprises most of the logical structure of all remaining subdocuments. Thus they can be automatically structured with very few further user corrections.

1.3 Contribution

The main contribution of this thesis lies in the effective combination of two approaches to machine learning – *version-space* (see Section 7.2.2) and *grammatical inference* (see Section 7.2.1), and their adaptation to the field of document structure recognition. The algorithms de-

veloped on this basis are used to abstract the concrete strings in the documents (see Section 4.2) and to learn the logical structure of the documents (see Section 4.3). The grammatical inference approach is further refined to allow for inferring general expressions involving arbitrary ordering (see Chapter 5).

The developed learning algorithms are embedded into a flexible and friendly user interface, implemented in the programming language Smalltalk (see Section 6.2). The user interface supports utilities for a variety of tasks, such as organizing predefined recognition patterns, manual mark up of examples, etc., in a uniform framework.

The approach described in this thesis thus fills the gap between structuring manually and structuring by a programming approach. It supports an easy way for the user who wants to structure similar on-line electronic documents and it can be applied in many areas where such “repetitive” documents occur.

The *MarkItUp!* system is fully implemented and forms an operational front-end to the DREAM parser (see Section 3.1).

1.4 A Guide to this Dissertation

The remainder of this thesis is organized as follows. Chapter 2 introduces some basic concepts which are used in the following chapters. In Chapter 3 the overall approach of *MarkItUp!* is presented. The parser of *MarkItUp!* – the DREAM parser generator is discussed first, and then the overall learning cycle of *MarkItUp!* is described. Finally, some examples demonstrate the learning process that has been discussed in the above learning cycle. Chapter 4 concentrates on the learning strategies in *MarkItUp!* and discusses how to derive grammars from marked-up examples by abstraction at the content level and how to unify and abstract these grammars at the structure level. Chapter 5 discusses the sequences with arbitrary ordering in a document collection, that is, these sequences describe how the structures of subdocuments in the document collection may have different ordering. Chapter 6 depicts the system architecture of *MarkItUp!* and detail functions, learning strategies, or major implementation algorithms of each component in the architecture with/without examples. Chapter 7 surveys related work in the areas of editing-by-example techniques and machine learning approaches. Finally, Chapter 8 illustrates some results, gives limitations of the learning approach and discusses the future work.

Preliminaries

To derive a general recognition grammar from document examples, two areas are built: Machine learning and document structure recognition. To discuss the problem of these areas, the following definitions and theorems should be taken into account. They can be separated into two parts corresponding to the above two areas.

The learning methods refer to inductive learning. The concepts of formal languages and graphs form the representational basis for the learning approaches.

Besides the machine learning approach to recognition, this thesis concentrates on the result of document structure recognition, that is, how to represent the structured document, rather than the processing of document structure recognition, that is, how to really abstract the logical structure from the document. Therefore, this chapter also introduces SGML – a standard document description language and its related concepts and notions, such as document structures, markup, etc.

To associate the definitions and theorems with their use in this thesis, a short explanation is given, at the beginning or end of some sections, on why the definitions or theorems are introduced in a section and where they will be applied in the thesis.

2.1 Formal Languages

Since regular grammars are used to represent and manipulate the structure of document examples and regular expressions are used to abstract a set of concrete strings, it is necessary to introduce some definitions about formal languages and discuss some of their characteristics.

2.1.1 Strings and Languages

A finite nonempty set Σ of arbitrary *symbols* (such as the ASCII character set) is called a *finite alphabet*.

A *string* over Σ is a *finite* sequence of symbols from Σ . All strings over Σ form an *infinite* set, denoted by Σ^* . The symbol ε stands for the *empty string* which contains no symbols and is

considered to be in Σ^* for every Σ . The *length* of a string s , denoted by $|s|$, is the number of symbols in s .

If u and v are strings over Σ , uv is the *concatenation* of them. Two strings u and v are *equal* if u and v have the same length and contain the same symbols in the same order.

The string u is a *prefix* of the string v if and only if there exists a string w ($w \neq \varepsilon$) such that $uw = v$, e.g., “ban” is a prefix of “banana”. Respectively, the string u is a *suffix* of the string v if and only if there exists a string w ($w \neq \varepsilon$) such that $wu = v$, e.g., “nana” is a suffix of “banana”.

The concept of the strings is used widely in Chapter 4 and 6.

Any finite or infinite subset of Σ^* is called a *language* L .

A *positive example of language* L is a string accepted by L ; conversely, a *negative example of* L is a string not accepted by L . The *MarkItUp!* learning system, currently, only adopts positive examples during the learning process.

2.1.2 Regular Expressions and Finite Automata

A *regular expression* [27] over a finite alphabet Σ is defined recursively as follows:

- ε is a regular expression;
- For each $a \in \Sigma$, a is a regular expression;
- If r and s are regular expressions denoting the languages $L(r)$ and $L(s)$, respectively, then $(r)|(s)$, $(r)(s)$, $(r)^*$ and (r) are regular expressions that denote the sets $L(r) \cup L(s)$, $L(r)L(s)$, $L^*(r)$ and $L(r)$, respectively.

In the above notations, the parentheses (r) and (s) may be substituted by regular expressions if desired.

A language denoted by a regular expression r is called as a *regular set*, written as $L(r)$. For instance, $L(a|b) = \{a, b\}$.

If two regular expressions r and s denote the same language, r and s are called *equivalent* and denoted as $r \equiv s$, for example, $(a|b) \equiv (b|a)$.

A regular expression can be compiled into a *recognizer* which is a program. It takes a string x as input and answers “yes” if x is a sentence of the language and “no” otherwise. The

recognizer represents a generalized transition diagram called a *finite automaton* (also called a *Deterministic Finite Automaton*, *DFA* for short). A DFA is formally denoted by a 5-tuple $(Q, \Sigma, \sigma, q_0, F)$, where

- Q is a finite nonempty set of *states*;
- Σ is a finite alphabet of *input* symbols;
- σ is a *transition function* mapping $Q \times \Sigma \rightarrow Q$;
- $q_0 \in Q$ is a *start* state;
- $F \subseteq Q$ is a set of *final* states.

A DFA allows only a single transition from a state on a specific input symbol. When a finite automaton allows zero, one or more transitions from a state on the same input symbol, the finite automaton is called a *Nondeterministic Finite Automaton* (*NFA* for short). Formally a NFA is denoted by a 5-tuple $(Q, \Sigma, \sigma, q_0, F)$, where Q, Σ, q_0 , and F (states, inputs, start state, and final states) have the same meaning as for a DFA, but σ is a map from $Q \times \Sigma \rightarrow \wp(Q)$.

The transition on the empty input ε is called an ε -*transition*, denoted as $\sigma(q, \varepsilon)$.

To implement the manipulations on regular sets, the following theorems provide a theoretical basis.

Theorem 2.1 ([27, Theorem 2.3]) Let r be a regular expression. Then there exists a NFA with ε -transitions that accepts $L(r)$. ■

In [3], the authors give an algorithm to construct a DFA from an NFA ([3, Algorithm 3.2]).

Theorem 2.2 ([27, Theorem 3.8]) There is an algorithm to determine if two finite automata are equivalent. ■

The following result is a direct corollary of the above theorems.

Corollary 2.3 There is an algorithm to determine if two regular expressions are equivalent.

Based on these results, a regular expression can be represented as a DFA. Thus, whether two regular expressions are equal can be determined by comparing their corresponding DFAs (Chapter 6).

2.1.3 Binary Relation of Regular Expressions

To abstract from concrete strings in documents and grammar rules, the more-specific-than binary relation of regular sets plays an important role. It is a basis of organizing regular expressions and using rewrite rules. Mitchell [37] and Vanlehn & Ball [49] give two kinds of definitions of this binary relation in their applications.

Let \mathfrak{R} be a set of regular expressions. For a given regular expression $r \in \mathfrak{R}$, let $L(r)$ be a set of strings matched by r . The following definitions are introduced.

Definition 2.1 (*Relation $<$*) Given $r, s \in \mathfrak{R}$, if $L(r) \subset L(s)$, then we say r is *more specific than* s , denoted by $r < s$, shorthand as more-specific-than. ■

Definition 2.2 (*Relation \leq*) Given $r, s \in \mathfrak{R}$, if $L(r) \subseteq L(s)$, then we say r is *more specific than or equal to* s , denoted by $r \leq s$, shorthand as more-specific-than-or-equal-to. ■

Definition 2.3 (*Relation \equiv*) Given $r, s \in \mathfrak{R}$, if $L(r) = L(s)$, then we say r is *equivalent* to s , denoted by $r \equiv s$. ■

Clearly, “ \equiv ” identifies an equivalence relation over \mathfrak{R} .

Definition 2.4 (*Comparable and Incomparable*) Given $r, s \in \mathfrak{R}$, if at least one of the relations $r \leq s$ and $s \leq r$ holds, then we say that r and s are *comparable*, denoted as $s \asymp r$; otherwise, they are *incomparable*, denoted as $s \not\asymp r$. ■

The following result is derived from the above definitions.

Theorem 2.4 The relation $<$ (and \leq) over \mathfrak{R} is transitive.

Proof: Let r, s , and $t \in \mathfrak{R}$ such that $r < s < t$. That is, $L(r) \subset L(s) \subset L(t)$, so that $L(r) \subset L(t)$. By Definition 2.1, we have $r < t$. Similarly, if $r \leq s \leq t$, we can infer $r \leq t$. ■

These definitions and theorems in the above sections are widely used in chapters 3, 4 and 6.

2.2 Graphs

This section introduces some related concepts with respect to graphs. These concepts will be used in Chapters 4 and 5 to organize a set of regular expressions and to discuss sequences of elements with arbitrary ordering.

2.2.1 Directed Graphs and Undirected Graphs

A *directed graph*, G , consists of a finite set V and an *irreflexive* binary relation on V [19]. The members in V are called *nodes* (or *vertices*). The binary relation may be represented either as a collection E of *ordered* pairs or as a function from V to its *power set*,

$$\text{Adj}: V \rightarrow \wp(V),$$

where $\text{Adj}(v)$ is called the *adjacency set* of node v . The ordered pair $(v, w) \in E$ is called an *edge*.

A sequence of nodes (x_0, x_1, \dots, x_n) , $n \geq 1$, is a *path of length n* from node x_0 to node x_n if there is an edge which leaves node x_{i-1} and enters node x_i for $1 \leq i \leq n$.

For a node x , the number of edges entering x is called the *in-degree* of x , the *out-degree* of x is the number of edges leaving x .

Two graphs $G = (V, E)$ and $G' = (V', E')$ are called *isomorphic*, denoted $G \cong G'$, if there is a bijection $f: V \rightarrow V'$ satisfying for all $x, y \in V$,

$$(x, y) \in E \Leftrightarrow (f(x), f(y)) \in E'$$

Let $G = (V, E)$ be a graph with node set V and edge set E . The graph $G^{-1} = (V, E^{-1})$ is said to be the *reversal* of G , if

$$E^{-1} = \{(x, y) \mid (y, x) \in E\},$$

A *symmetric closure* of G is the graph $\overline{G} = (V, \overline{E})$, where

$$\overline{E} = E \cup E^{-1}$$

A graph $G = (V, E)$ is called *undirected* if its adjacency relation is symmetric, i.e., if

$$E = E^{-1},$$

or equivalently,

$$E = \overline{E}$$

2.2.2 Directed Acyclic Graphs

A *directed acyclic graph* (or *DAG* for short) is a directed graph that has no cycles. Figure 2.1 shows an example of a DAG.

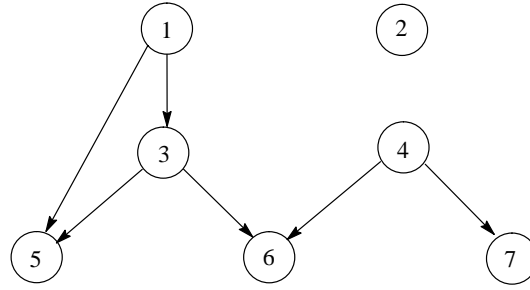


Fig. 2.1 Example of a DAG

A node having in-degree 0 will be called a *base* node. One having out-degree 0 is called a *leaf*. In Figure 2.1, nodes 1, 2, and 4 are base nodes and nodes 5, 6 and 7 are leaves.

If (x, y) is an edge in a DAG, x is called a *direct ancestor* of y , and y is called a *direct descendant* of x . For example, in Figure 2.1, node 4 is a direct ancestor of node 7; node 7 is a direct descendant of node 4.

If there is a path from node x to node y , then x is said to be an *ancestor* of y and y is said to be a *descendant* of x . In Figure 2.1, node 6 is a descendant of node 1; node 1 is an ancestor of node 6.

DAGs are used in Section 4.2.4 to construct concept bases and to infer linear ordering lists of the nodes in the concept bases.

2.2.3 Permutation Graphs

Let π be a permutation of numbers $1, 2, \dots, n$ and denoted as the sequence $[\pi_1, \pi_2, \dots, \pi_n]$. For example, the permutation $\pi = [2, 3, 4, 1]$ has $\pi_1 = 2, \pi_2 = 3, \pi_3 = 4, \pi_4 = 1$. The notation π_i^{-1} denotes $(\pi^{-1})_i$ which indicates the position in the sequence where the number i can be found, that is, $(\pi^{-1})_{\pi_i} = i$; for the above example, $\pi^{-1} = [4, 1, 2, 3]$, where $\pi_1^{-1} = 4, \pi_2^{-1} = 1, \pi_3^{-1} = 2, \pi_4^{-1} = 3$.

Given a permutation π of numbers $1, 2, \dots, n$, a *permutation graph* (or *PG* for short) for π is an undirected graph $G[\pi] = (V, E)$ where

$$V = \{1, 2, \dots, n\}, \text{ and } E = \{(i, j) \mid i, j \in V \text{ and } (i - j)(\pi_i^{-1} - \pi_j^{-1}) < 0\}.$$

Informally, each edge in a PG indicates an inversion between two nodes. An undirected graph G is called a *permutation graph* if there exists a permutation π such that G is a graph isomorphic to $G[\pi]$.

Theorem 2.5 For a permutation π of numbers $\{1, 2, \dots, n\}$ there is a sole PG corresponding to it, i.e., a permutation π of numbers $\{1, 2, \dots, n\}$ and a permutation graph $G[\pi]$ expressing a one-to-one mapping.

Proof: The definition of the permutation graph and the construction of it ([19], Golumbic, pp. 157).

The application of the PG is described in Chapter 5.

2.3 Document Structures

Since the problem domain of this thesis refers to documents structure recognition (Section 1.1), starting from this section some concepts are introduced which provide background knowledge related to the problem domain and overall goals.

Document structures are meant to describe the various parts of a document and the connections between them. Generally, two distinct structures are associated with a document: *the layout structure* and *the logical structure* which are independent of each other and are determined by different processes – formatting process (layout structure) and editing process (logical structure).

However in the problem domain of this thesis, the source documents have unusual structures – implicit structures, they are called *syntactic structures* of the documents. In the following sections, the syntactic structure of the documents is introduced first, and then the usual structures of documents are discussed.

2.3.1 Syntactic Structures

Syntactic structures of documents describe a kind of implicit structures, such as the field names in bibliography documents; the delimiters, such as the points in an expression of date; invariant strings at the beginning of e-mail documents (e.g., the string “From”, “Subject”, etc.); and section numbering etc.. With the help of such structures, the human reader can easily recognize the contents of documents and understand what is meaning of the contents.

Note that not every document is associated with a syntactic structure.

2.3.2 Layout Structure

The geometric or layout structure is the result of dividing and subdividing the content of a document into increasingly smaller parts, on the basis of the *presentation* [47].

The document layout structure is usually determined by a formatting process, for instance, a book has 100 pages, on page 3 there are 28 lines. The formatting process may be controlled by attributes called *geometric directives* associated with the logical structure. For example, the geometric directive requires that a chapter starts on a new page, or the title of a section and the first two lines of its first paragraph are presented on the same page. Geometric directives may be collected into layout styles each of which may be referred to by one or more logical objects.

2.3.3 Logical Structure

The logical structure is the result of dividing and subdividing the content of a document into increasingly smaller parts, on the basis of the *human perceptible meaning* of the document content [47].

The document logical structure is determined by the author and embedded in the document during the editing process, e.g. title of the document, author(s), summary, chapter, etc. It specifies a kind of logical relationship among the logical objects.

The relationships among logical objects in the logical structure are typically in the form of *sequences* and *hierarchical nests*. The logical structure breaks a document, for example, into chapters, sections, and paragraphs, defines headings, and determines links and references among various objects. The structure in question is abstract and totally independent of the way the document is presented.

The document logical structure is often represented by a tree structure [16, 14]. This model is particularly useful since it allows both sequences and hierarchical nests of objects to be expressed.

In order to parameterize recognition algorithms and, in particular, to interpret the layout structure to build up the logical structure, or to transform one logical structure into another logical structure, a formal description is necessary. A formalism based on grammars seems appropriate to describe a generic logical structure [29, 33]. The nonterminals of this grammar represent the various hierarchical objects in the logical structure, while terminals correspond to document elements. Grammar rules allow optional objects to be described, as well as sequences, alternatives and iterations.

2.3.4 Relations Between Layout and Logical Structure

The layout (or geometric) structure and the logical structure provide alternative views on the same document. For instance, a block can be regarded as consisting of chapters containing

figures and paragraphs, or alternatively, as consisting of pages that contain text blocks and/or graphic blocks. There is an obvious relationship between these two structures because the task of an editor is to make the logical structure defined by the author reveal itself in the presentation of the document.

There is no one-to-one mapping between logical structure and geometric structure since the same logical document can be presented in different ways, in other words, a logical structure corresponds to a variety of geometric structures, while a geometric structure can be abstracted into different logical structures. However, certain links can be established between the two structures.

A transformation of a geometric structure into a logical structure can usually be found, that is, a logical structure can be regarded as an abstraction of a geometric structure. But the reverse transformation does not always exist because some typesetting elements are needed that have no corresponding description in the logical structure. For instance, layout notions such as a page or a line do not have logical equivalents, nor do page numbers and hyphens in divided words. These must be considered as artificial elements introduced during typesetting.

Within this framework, the goal of document structure recognition can be regarded as determining a logical abstraction from a geometric structure.

2.4 Markup

Document processing systems typically require to incorporate additional information into the document being processed. When a document is to be printed, a formatter has to process the document. The input to such a formatter consists of the text of the document interspersed with formatting commands. These formatting commands or added information in the natural text of the document being processed and structured are called *markups*. With the advent of text-processing systems, new types of markup and new types of processing came. Until now there are three major types of markup [13] to work with the unstructured documents: *Presentation markup*, *procedural markup*, and *descriptive markup*.

Presentation markup expresses the most basic organization of a document, such as horizontal and vertical spacing etc. The goal of presentation markup is to make the document suitable for reading.

Procedural markup consists of formatting commands, such as “insert a blank line” and “start a new page”, in a document. The problem of procedural markup is that the markups are

mapped to actions of a specific device, that means, certain markups correspond to a special formatter. When the document is manipulated by different formatters or is used in some other applications, the markup in the document must be changed.

Descriptive markup is the highest level of markup. It overcomes the problem of procedural markup, e.g. it is independent from device and software. In addition, descriptive markup can guarantee a one-to-one mapping between logical elements and markup. This markup is adopted in the standard document description language – SGML (see following sections) and becomes a part of SGML-documents.

An SGML-document is a string of characters which consists of the text of the document interspersed with markup tags to identify the start and end of each logical item. In other words, it consists of two different types of data: one type forms the *content* of a document, the other type constitutes the *markup* of a document which explains the content's structure. The rigorous structure description of an SGML-document is machine readable but also easily understood by humans.

Recently, to specify the formatting and transformation of SGML-documents, the International Standards Organization (ISO) defined the Document Style Semantics and Specification Language (DSSSL). It is also a kind of markup but associates processing with SGML-documents rather than unstructured documents.

2.5 Document Description Language – SGML

SGML (*Standard Generalized Markup Language*) was standardized by ISO in an effort to standardize electronic manuscript encoding techniques [5]. It specifies how descriptive markup can be incorporated into a document which can help to organize a well-defined logical structure of documents.

SGML is a generic markup language. It can be used to describe any document structure and the description is independent from hardware and software.

The main idea of SGML is to add *tags* to the document that identify the different structural components independent of the layout information. For example, a tag for a chapter could be <chapter> (this thesis uses the font like <chapter> to describe examples) instead of a chapter-head description (e.g. 16pt Times, Bold, Centered). The tagged documents (*SGML-documents*) are independent of devices (computer systems or other text entry/processing devices), character sets, types of processing, and file organizations.

SGML is not a text formatting system. It is designed as a standard for text interchange format. For example, an SGML-document is easily converted to text formatters like T_EX [32], *troff* [41], or some other similar type of formatter. It can be used to produce a typeset document on paper [50] also.

The following subsections detail the main features of SGML: SGML markup, SGML DTD (*Document Type Definition*), and SGML parser.

2.5.1 SGML Markup

Marking up in SGML means to insert tags into the documents. Every element that requires markup is enclosed between `<tag name>` and `</tag name>`. `<tag name>` is called a *start-tag* of the “tag name” and `</tag name>` called an *end-tag* of the “tag name”. The contents among start-tag and end-tag is called *tagged*. For instance, a chapter would therefore be marked up as follows:

```
<chapter>
MarkItUp! is a system to recognize...
...
... one major issue for further developments.
</chapter>
```

Total tagging is cumbersome for a user, thus the SGML syntax allows the omission of *end-tag* where they are redundant. For example, to define a list, the end marker can be left out for each element, only the start marker is identified.

```
<list>
<item> this is the first item. No end marker.
<item> this is second item.
    Only the end of list marker is required.
</list>
```

These examples show that an SGML-document does not contain formatting instructions for word processors or printers. To obtain a formatted output from an SGML source document, such as the above examples, the markup has to be translated into specific formatting commands for the text formatter to enable formatting and printing.

2.5.2 SGML DTD

Every SGML-document refers to a program DTD which specifies the document class, defines the logical structure of a document in terms of the elements that comprise it (title, author, ad-

dress etc.) and the rules for marking up the document instance. A document markup is formalized by associating it with a DTD, which includes a specification describing the order in which document elements can occur.

The DTD contains a set of ELEMENT definitions giving the name (*generic identifier*) of the element and a *content model* that defines which sub-elements and character strings can occur in the content.

The content model consists of elements or terminals with the *group connectors* – *and*(&), *or*(/) and *seq*(,), where the group elements are evaluated from left to right, and the *occurrence indicators* – *rep*(*), *opt*(?) and *plus*(+) to describe the relationship among the elements or the characteristic of the elements in a structural description. The occurrence indicators in a DTD indicate how often the preceding element or group of elements may occur. The indicators are directly corresponding to the operators defined in elements. If no occurrence indicators are used, the preceding element must occur exactly once. The indicator * allows an element occurring zero or more times. The indicator + requires the element occurring at least once, and the indicator ? denotes an optional element.

As an example, consider a scientific paper described by a DTD grammar:

```
<!ELEMENT paper - - (title, abstract?, te-para+)>
<!ELEMENT title - - CDATA>
<!ELEMENT abstract - - (ab-para+, ind-term*)>
<!ELEMENT ab-para - - CDATA>
<!ELEMENT ind-term - - CDATA>
<!ELEMENT te-para - - CDATA>
```

where te-para, ab-para, and ind-term express text-paragraph, abstract-paragraph, and index-term, respectively.

The DTD means that the first logical element in a paper conforming to this grammar must be its title. The paper can optionally have an element abstract, that is, the following element of title may be an element abstract if there is, or may be a text paragraph if there is no element of type abstract in a document.

2.5.3 SGML Parser

An SGML-document is read and interpreted by an SGML *parser* which analyses and checks whether the markup in the document conforms to the rules defined in the associated DTD, and

inserts tags whose presence is implied. The end product is a fully SGML-document conforming to the DTD, or an error message. An SGML parser is a program or a suite of programs.

After an SGML parser parsed the logical structure of an SGML-document, it will report any error messages it finds. But it does not correct the error messages and does not mark up a non-SGML document. Once an SGML-document has been verified by a parser, it can be processed in many different ways, for instance, it can be inserted into databases.

2.5.4 The Marking Up (Tagging) Process

Independent from whether procedural markup or descriptive markup is used, there always exists the problem of correctly inserting the markup into the text of the document. Since SGML defines only the *syntax* of a standard generalized markup language, it does not support an approach to insert tags into the large number of untagged documents.

The early and normal marking up approach is that the user inserts markup into the text by hand. For example, for powerful formatters such as T_EX or *t_ro_ff* the user usually inserts the formatting commands provided by T_EX or *t_ro_ff* into a document.

For SGML markup, there are several ways [25] of working:

- In a normal editor, tagging is done by hand
- In certain editors, frequently occurring tags in the text are bound to program keys on the keyboard, or *templates* for a given DTD are supported, i.e. a skeleton SGML file with the major tags already in the file
- The tags are added by a program, such as the tools IMSYS, FastTAG, and DREAM.

An Approach to Document-Structure Recognition

This chapter gives an overview of the learning system *MarkItUp!*. At first the DREAM parser generator is introduced which is used by *MarkItUp!* to actually mark up source documents. In particular, the DREAM DSD (*Document Structure Description*) is discussed which is used to describe the necessary structuring knowledge. After that, Section 3.2 gives an overview of *MarkItUp!* which learns DSDs from examples. Section 3.3 provides an example to demonstrate the learning process of *MarkItUp!*. Details of the learning approaches are discussed in chapter 4 and the system implementation is given in chapter 6.

3.1 DREAM

DREAM [20, 21] is a parser generator (see Figure 3.1) specifically designed for extracting the logical structure of documents, based on formatting and content information. For this purpose it uses DSDs (*Document Structure Descriptions*) which consist of rules to relate the layout information found in unstructured documents with a desired logical structure.

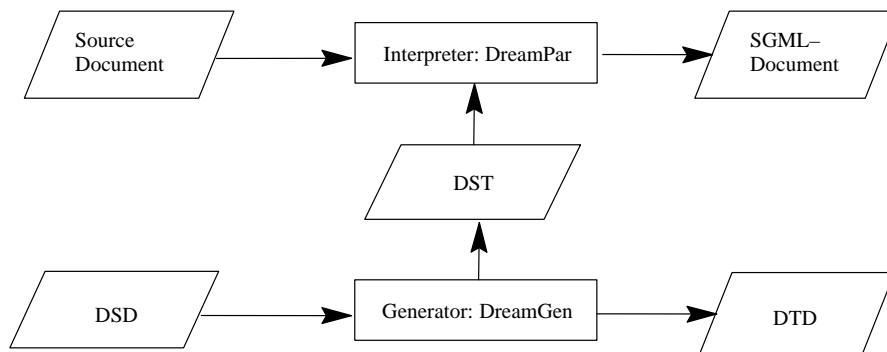


Fig. 3.1 The system architecture of DREAM

The input of the DREAM system are a *source document* and a DSD. The source documents are untagged documents which come from electronic sources or which are the result of documents layout structure analysis. The output of the system are an SGML-document and its DTD if the user requires the DTD.

The two main components of DREAM are a generator *DreamGen* and a parser *DreamPar*. *DreamGen* is used to compile DSDs and generate corresponding parser tables DSTs (*Document Structure Tables*). *DreamPar* uses the DSTs in order to introduce markups into the source documents, that express their logical structure.

For various source documents DREAM requires different DSDs to define their logical structure. And with different DSDs man can get diverse SGML-documents. Therefore, to get the desired SGML-documents by DREAM, it is required to support correct DSDs for source documents.

A DSD consists of two parts, the *structure description* and the *recognition style*. The structure description is very similar to SGML DTDs. The recognition description is a characteristic part of DSD which is composed of regular expressions and functions. Section 3.1.1 and 3.1.2 will describe these two parts in more detail.

3.1.1 Structure Description of a DSD

The structural part of a DSD describes the document structure in terms of hierarchically structured elements.

Each DSD starts with a statement “<!DOCTYPE *type-name* [” which gives a document type name and ends with a statement “[>”, where the part with italic font means a variable part in the DSD.

The square brackets in the statements enclose the entire definition part of a DSD which consists of a set of *element definitions*. Each element definition has the form:

“<!ELEMENT *element-name* – – (*structure description* / *recognition style*)>”

Where the italic font has the same meaning as above; the element-names are used as tag names in the markup process. If an element-name *E* appears in a structure description, it leads to another element definition, that is, there is an element definition to describe the element-name *E* in the DSD.

The first element definition in any DSD is called the *document structure root* (or *root* for short). Except the document structure root, all element definitions may appear in arbitrary orderings. For instance, a piece of a sample document could look like this:

Document-Sample 3.1

```
...
!␣Suad␣Alagic\n
"␣Object-Oriented Database Programming␣"\n
...
```

where the symbol `␣` and the symbol `\n` denote a *blank* character and a *return* character (an empty string at the end of a line) respectively (in the following examples the symbols express the same meaning). These symbols express the syntactic structure of this document. In Document-Sample 3.1 the symbols `!␣` indicate the document author, the author is followed by the document title indicated by the symbols `"␣`. Before the element author and after the element title there are other elements which are omitted here (denoted by `...` in the example). A complete example will be given in Document-Sample 3.2 in Section 3.1.3. The document hierarchy structure is easily written down in terms of a DSD:

DSD-Sample 3.1

```
<!DOCTYPE bibdoc [

    <!ELEMENT bibentry - - (author, title)>
    <!ELEMENT title    - - recognition style>
    <!ELEMENT author  - - (fname, "␣", lname)>
    <!ELEMENT fname   - - recognition style>
    <!ELEMENT lname   - - recognition style>
    ...
]>
```

where the document type name is `bibdoc`. The document structure root is `bibentry`. The elements `bibentry` and `author` are described by the structure descriptions: `author`, `title` and `fname`, `"␣"`, `lname`, respectively. The structure description `author`, `title` has two meanings: (1) the element `bibentry` contains two elements: the element `author` and the element `title`; (2) the element `author` is followed by the element `title`. A similar explanation can be applied for the element `author`. The elements `title`, `fname`, and `lname` are described by *recognition style* which will be replaced by DSDs' expressions and functions in Section 3.1.3.

DSD-Sample 3.1 shows the structure of the document definition that explains those parts of DSD's which are similar to SGML DTD's. The following section will discuss the extension of DSD's with respect to DTD – recognition styles.

3.1.2 Recognition Styles of DSDs

Recognition styles in DSDs provide detailed information by regular expressions and functions to actually analyze and mark up the document. To describe recognition styles, it is necessary to define the allowed form of regular expressions and functions in DSDs.

3.1.2.1 Regular Expressions in DSDs

Regular expressions in DSDs identify delimiting text portions and element contents. Delimiters, such as the symbols `!□` and `"□` in Document-Sample 3.1, are filtered out and the remainder of the texts are mapped into corresponding elements in the output document.

DREAM supports a form of regular expression notations defined in [20]. But this section only gives the partial notations which will be used in the following examples.

- (a) The regular expression `'A'` matches exactly one character `'A'`.
- (b) The regular expression `'.'` matches any character.
- (c) A set of characters enclosed by square brackets `[` and `]` matches any single character in that set. For example, the regular expression `[0123456789]` matches any single digit. A range of ASCII characters may be specified by giving the first and last characters, separated by a hyphen `-`. For example, the above regular expression `[0123456789]` can be shortened as `[0-9]`.
- (d) A regular expression matching a single character may be followed by one of several repetition operators:
 - ? The preceding regular expression is optional and matched at most once;
 - + The preceding regular expression will be matched one or more times;
 - * The preceding regular expression will be matched zero or more times;
 - # It has a similar meaning as the operator `*`. But it forces minimal parsing (see below).
- (e) Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions. For example, two regular expressions `[A-Z]` and

`[a-z]` can be concatenated as `[A-Z][a-z]` which matches the strings starting with a capital letter and followed by an arbitrary number of small letters.

- (f) Two regular expressions may be combined by the alternation operator `|`; the resulting regular expression matches any string matching either subexpression. For example, two regular expressions `[A-Z]` and `[a-z]` joined as `[A-Z] | [a-z]` match a capital letter or a small letter.
- (g) Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression may be enclosed in parentheses (and) to override these precedence rules.
- (h) The caret `^` matches the empty string at the beginning of a line and the `$` matches the empty string at the end of a line.
- (i) In the above regular expressions the special characters such as `?`, `+`, `*`, `|`, `-`, and `\`, etc., lose their special meaning when used in backslashed versions `\?`, `\+`, `*`, `\|`, `\-`, and `\\`.

The operator `#` is a new and an important operator in the DREAM DSD. It parses elements only if none of the subsequent element definitions is matched. With the regular expression `.#`, the DREAM parser tries to parse subsequent regular expressions before accepting the next character as belonging to `.#`. The regular expression `.#` means that the parser can accept *arbitrary strings*.

3.1.2.2 Functions in DSDs

DREAM offers three functions: `copy()`, `cut()` and `paste()` which can be described by a general form: `function-name(<regex>)`. The names and functions of `copy()` and `cut()` in DSDs correspond to the current editing operations in text processing programs. The content included by `copy()` is mapped into the output document, whereas the content in `cut()` is filtered out. The extra function `paste()` makes it possible to add extra sequences into the output document. Regular expressions in DSDs are used only as an argument of these functions.

These regular expressions and functions can replace the *recognition style* of the element `title` in DSD-Sample 3.1 as follows:

```
<!ELEMENT title -- (cut("^\""), copy([a-zA-Z\\-]+), cut("\"\"$"))>
```

The string beginning `cut("^\"")` and ending `cut("\"\"$")` is called a *recognition style* of the element `title`. The meaning of the recognition style is that the strings matched by the

regular expressions `"\ "□` with the empty string at the line beginning and `"□\"` with the empty string at the line end are filtered out. The string matched by the regular expression `[a-zA-Z□]+` (arbitrary letters and blanks) is mapped onto the element `title` in the output document.

3.1.3 A Complete Example of a DSD

A complete description of Document-Sample 3.1 and the corresponding DSD are shown as follows.

Document-Sample 3.2

```
\□bk\n
!□Suad□Alagic\n
"□Object-Oriented Database Programming□"\n
/□Springer□*□1989\n
>□DBDObject\n
```

where the symbols `\□` at the beginning of the example indicate the document code, the code followed by the document author starting with the symbols `!□`, the document title starting with the symbols `"□`, the document source starting with `/□`, and the document category starting with the symbols `>□`. The DSD of the example is written down as:

DSD-Sample 3.2

```
<!DOCTYPE bibdoc[
<!ELEMENT bibentry - - (code, author, title, source, category)>
<!ELEMENT code - - (cut("^\\□"), copy([a-z]+), cut($))>
<!ELEMENT title - - (cut("^\"□"), copy([a-zA-Z□\\-]+),
cut("□\""$))>
<!ELEMENT author - - (cut("^!□"), fname, cut("□"), lname, cut($))>
<!ELEMENT fname - - (copy([A-Za-z\\-.]+))>
<!ELEMENT lname - - (copy([A-Za-z]+))>
<!ELEMENT source - - (cut("^/□"), publication, cut("□*□"), date,
cut($))>
<!ELEMENT publication - - (copy([a-zA-Z]+)) >
<!ELEMENT date - - (copy([0-9]+)) >
<!ELEMENT category - - (cut("^\\>□"), copy([A-Za-z]+), cut($))>
]>
```

With this DSD the DREAM parser identifies the document `author` by the following sequence: the symbols `!␣` at the beginning of the line, the element of the first name `fname`, the symbol `␣`, the element of the last name `lname` at the end of the line. The symbols are filtered out, the elements are identified further on the basis of their element definitions somewhere in the DSD. Thus, the element `author` is described as follows: filtering out the symbols `!␣` at the beginning of the line, mapping arbitrary letters, hyphens, or periods (denoted by the regular expression `[A-Za-z\-.]+`) into the element `fname` in the output document, filtering out the symbol `␣`, mapping arbitrary letters (denoted by the regular expression `[A-Za-z]+`) into the element `lname` in the output document, and then filtering out the empty string at the end of line. A similar explanation holds for the other elements.

DREAM has already been successfully applied to diverse sources as *Usenet*, *Articles*, *Publishing Abstracts*, and downloads from online databases like *Compuscience* and *Conference*. The resulting documents can be further processed with any SGML-based tool.

3.2 System Overview of *MarkItUp!*

The goal of the *MarkItUp!* system is to learn DREAM DSDs from examples instead of the user writing the DSDs. The system is designed on the basis of the scheme proposed by B. Knobe and K. Knobe [31], a kind of refinement method for grammatical inferences. Figure 3.2 shows the overall structure of the *MarkItUp!* system.

The input document of the *MarkItUp!* system is a document collection (see Section 1.1) which consists of (nested) sequences of subdocuments with similar format. The document collections as source documents can be provided either by electronic sources, such as electronic mail, on-line public databases etc., or as results of document layout structure analysis, like ASCII information from OCR.

The output document of the system is a *structured document* (SGML-document).

The tagging part of the system is the *DREAM parser* [20, 21].

Three components – *structure editor*, *scanner*, and learning which includes *content abstraction* and *structure unification & abstraction* – form the kernel of the system. A *list of concepts* contains partial ordered abstract strings (they are called *concepts* and represented by *regular expressions* in Section 4.2) which describe the general characteristic of strings appearing in a class of documents and are used by the learning component. Diverse sets of abstract strings correspond to different concept sequences, that is, the list of concepts is changeable.

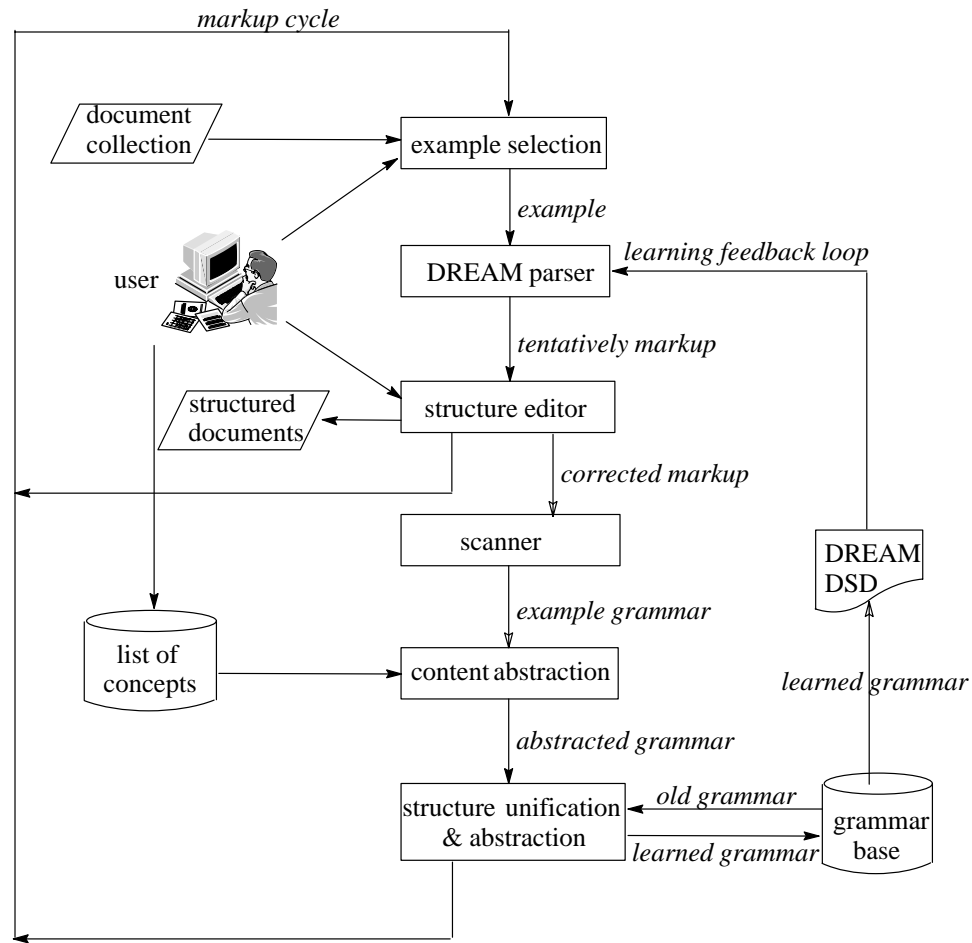


Fig. 3.2 System overview of *MarkItUp!*

When the user starts the *MarkItUp!* system, the system enters a learning cycle: the *markup cycle*. In the learning cycle, the user can:

- control the system when it is called and when it will be stopped;
- select examples from a document collection through a computer terminal;
- input or modify the abstract strings of the document's collection;
- manually mark up the initial example in the structure editor;
- judge the tentatively marked up examples on whether they are correct and correct them if they are not satisfactory.

To accomplish the other functions in the cycle, the user calls the system to generate the list of concepts on the basis of the given abstract strings of the documents' collection and to start the *learning feedback loop* in which the system can

- scan the structured document;
- abstract from the concrete strings;
- learn a new logical structure of documents on the basis of the old grammar;
- translate the learned grammar into a DREAM DSD;
- finally call the DREAM parser to mark up new examples.

The following subsections describe the *MarkItUp!* system under four aspects: starting the system, structure editor, scanner and learning strategies.

3.2.1 Starting the *MarkItUp!* System

The system is started when the *user* activates it. The user selects an example (*example selection*), which is sent to the parser. Since initially there is no grammar for the document, the parser cannot further structure the example. In this case, the user has to manually mark it up by means of a simple yet comfortable *structure editor*. In subsequent cycles, when a grammar is available, the parser tries to mark up the example with as much structure as possible. If the user is not satisfied with the marked up structure, the user can change the markups or add new markups with the help of the structure editor.

3.2.2 Structure Editor

The structure editor is a window to allow the user accessing documents, executing the *MarkItUp!* system commands, marking up documents, correcting the marked-up documents, and displaying the results using a graphic representation, etc.

For giving a complete explanation of the functions of the structure editor, let us consider a process to manually mark up an example in the structure editor at the initial state of the started system. The initial state of structure editor looks like the description in Figure 3.3.

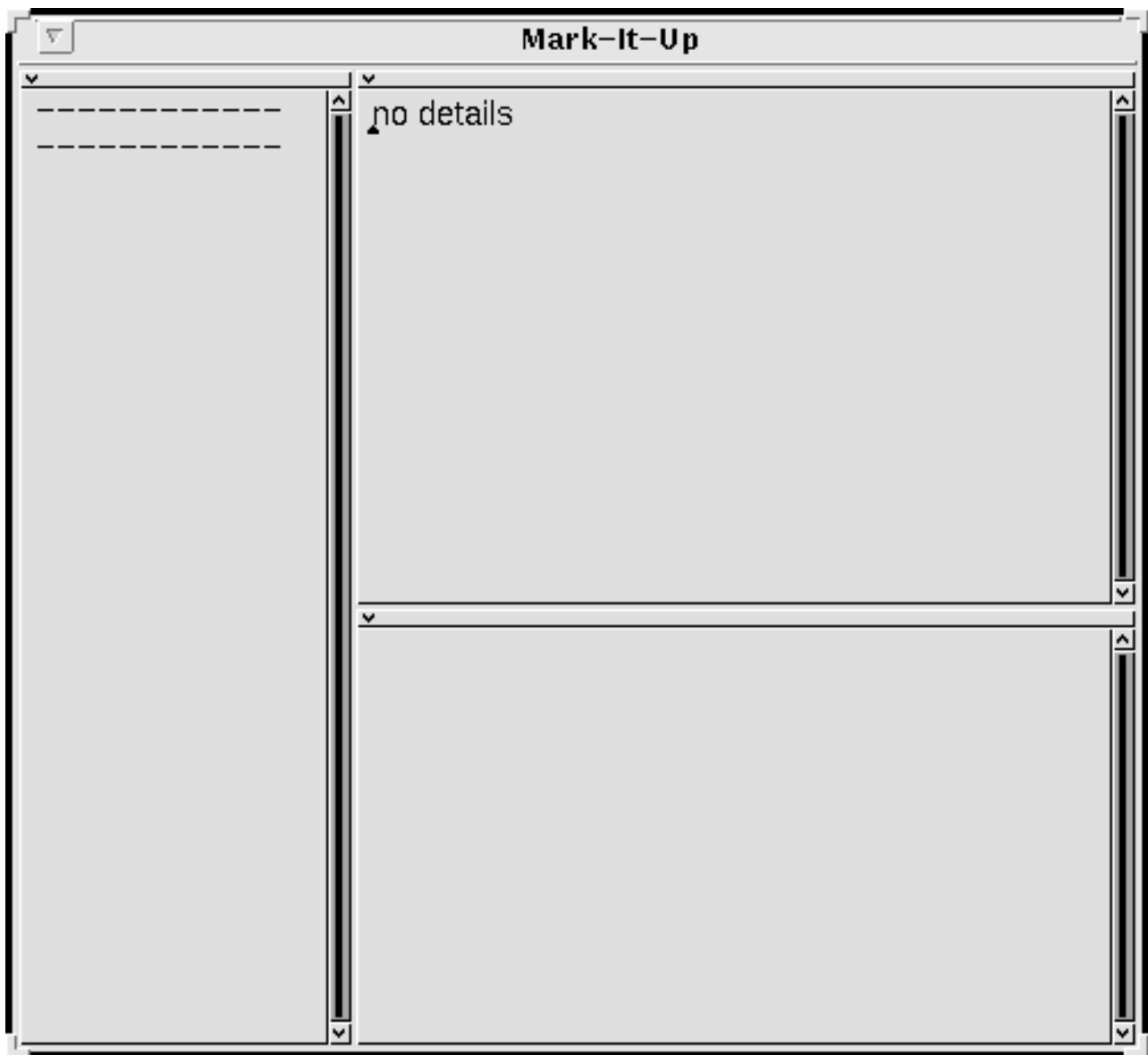


Fig. 3.3 The initial state of the structure editor

In this case the user has to first access the example (e.g. Document-Sample 3.2 in Section 3.1.3) from a file or type it from the keyboard into the structure editor directly. When the example is loaded, the structure editor shows the example as described in Figure 3.4.

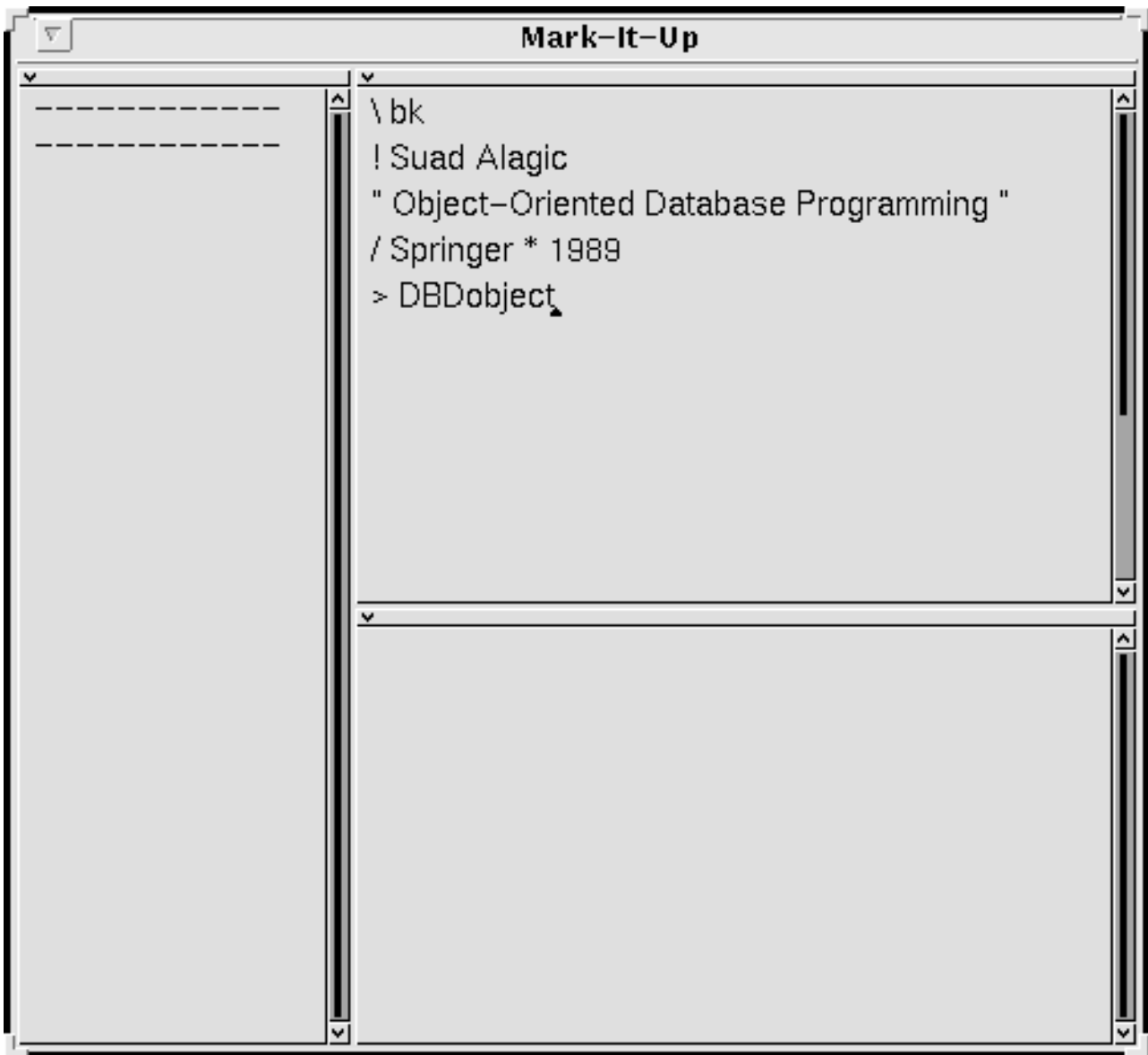


Fig. 3.4 An example loaded into the structure editor

In the structure editor man can manually mark up a string with the following three steps:

- (1) highlight a tagging string by a mouse;
- (2) call the tagging function from the editor menu;
- (3) type a string as the highlighted string's tag name when the function requires it.

Figure 3.5 shows the steps (1) and (3): the highlighted string "Object-Oriented Database Programming" and a dialog view, in which the tag name "title" of the highlighted string is typed.

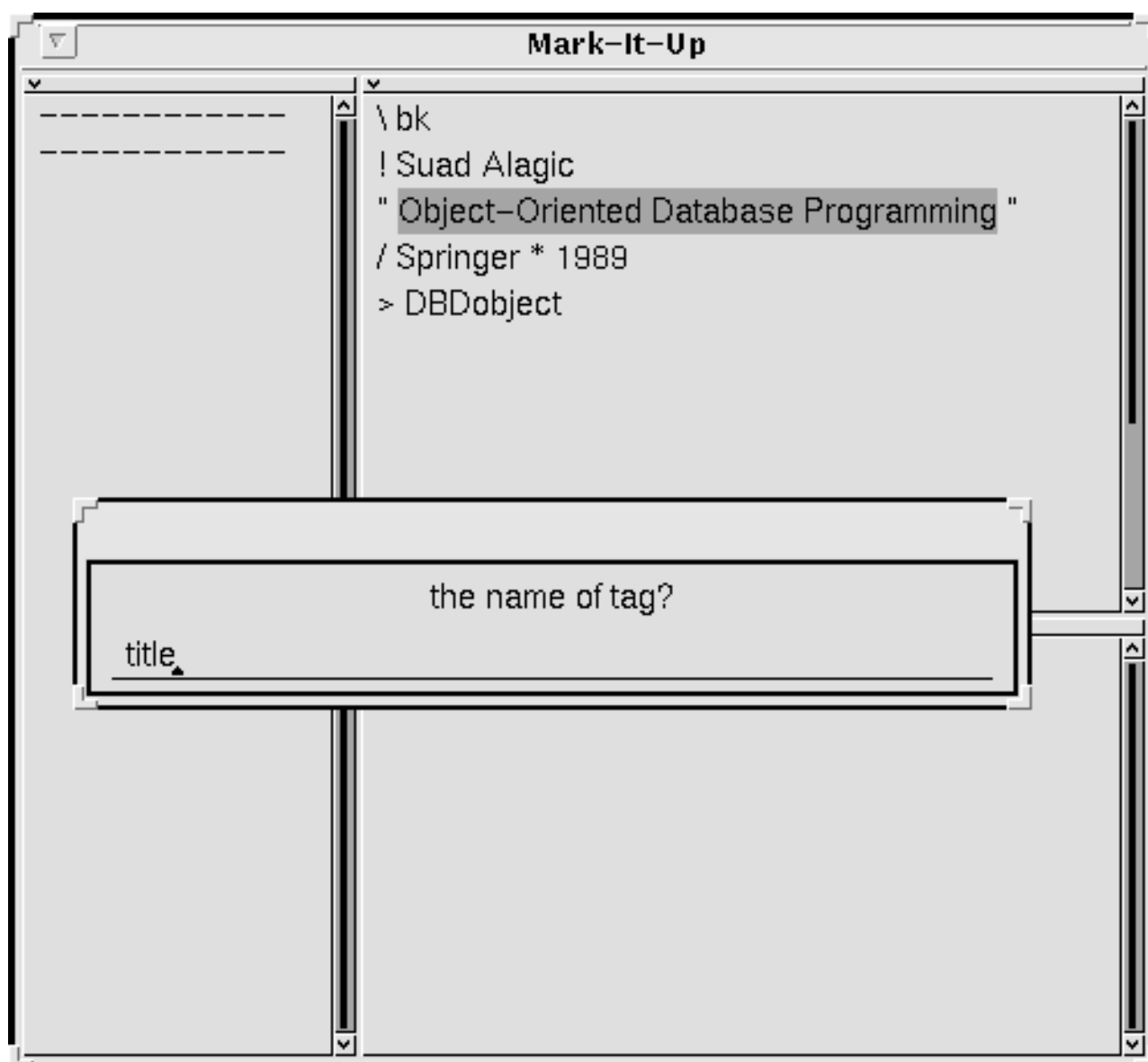


Fig. 3.5 Manually marking up a string in the structure editor

The dialog view is displayed after the second step is done. After the function is executed, the highlighted string is enclosed by the tag name with a bold font shown in the structure editor. The result of tagging a string is shown in Figure 3.6.

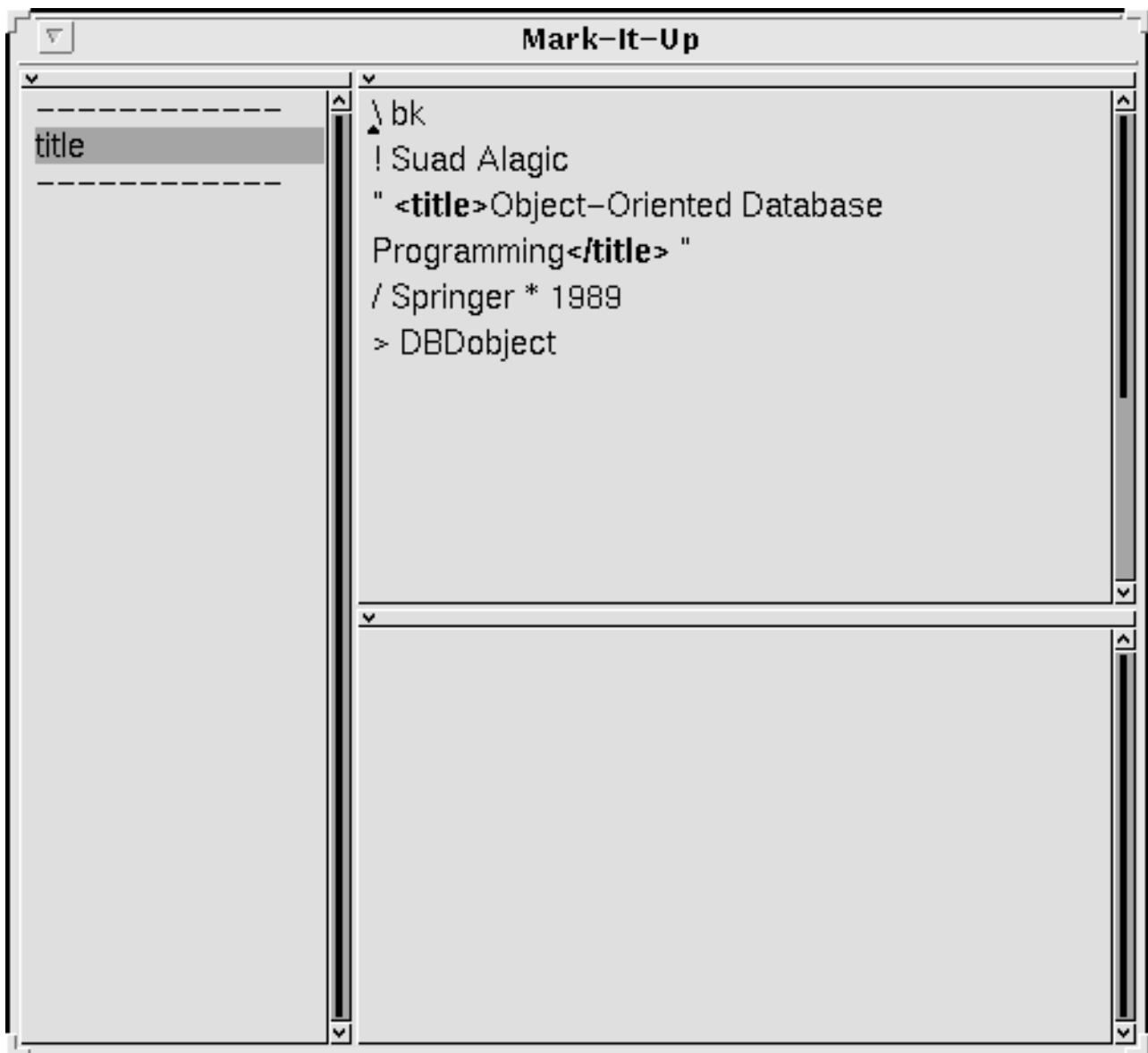


Fig. 3.6 The result of manually marking up a string in the structure editor

By repeating the three steps, the user can mark up the whole example in the structure editor. The marked-up result of Document-Sample 3.2 in the structure editor is shown in Figure 3.7:

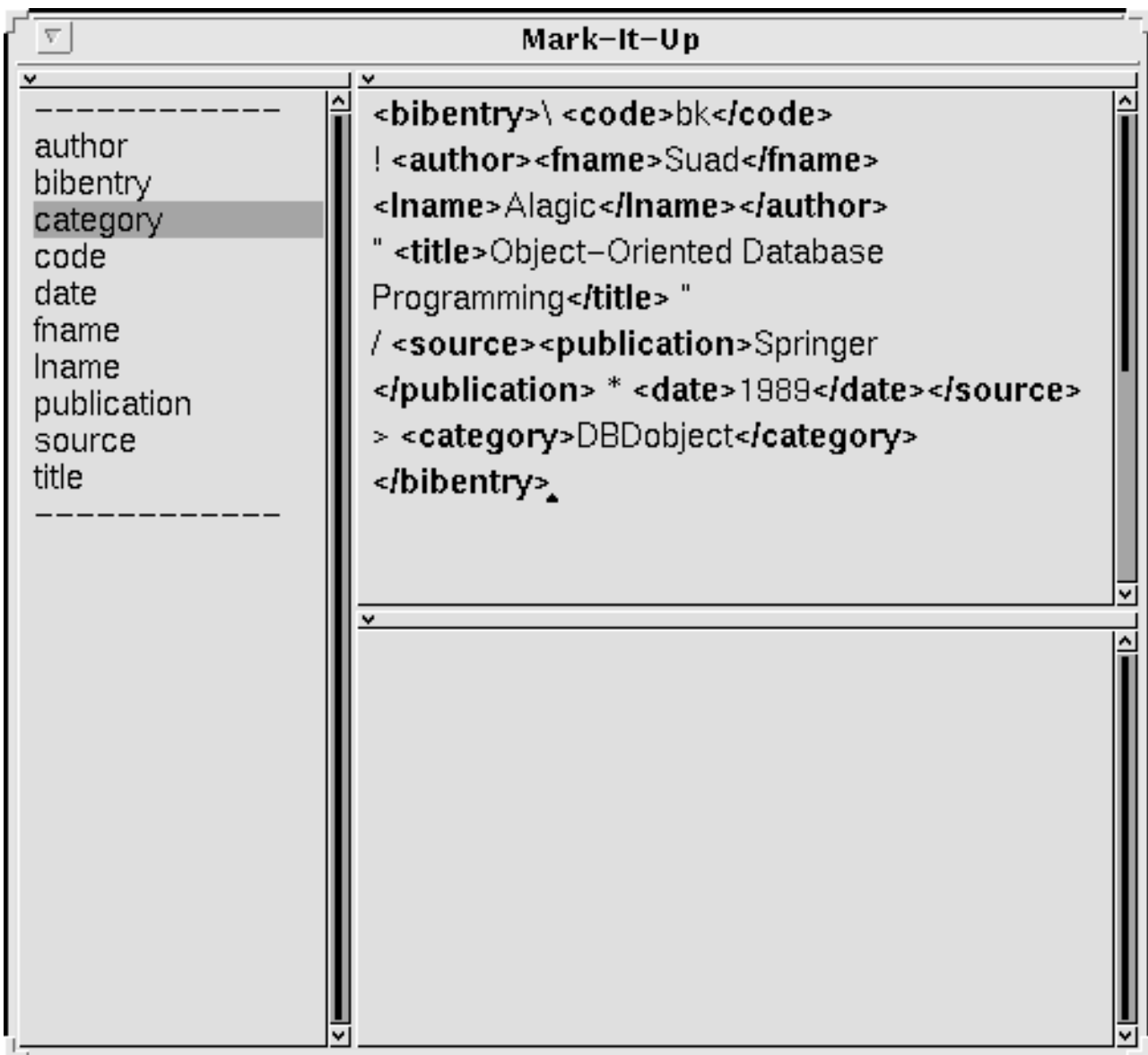


Fig. 3.7 The manually marking up result of the example in Figure 3.4

where tag names are included by angled brackets `<` and `>` and displayed in bold font. The bold font is not necessary in SGML-documents. It is used here to aid the user.

Either start-tag `<tag name>` or end-tag `</tag name>` expresses a *tag*. The contents separated by tags are *strings*. There are two kinds of strings: *cut-string* and *copy-string*.

Cut-strings are filtered out explicitly. They are identified by the following rule: if a string is not directly surrounded by a *pair of tags* which consists of a start-tag and an end-tag with the *same* tag name, the string is a cut-string. For instance, in Figure 3.7, the string `"\□"` in the

first line is a cut-string because it is directly surrounded by the tags `<bibentry>` and `<code>` which are not a pair of tags.

Copy-strings are formed by the rest in the document and are mapped with the surrounding tag in the output document. For instance, the string "bk" in the first line is a copy-string and is shown in the output document as `<code>bk</code>`. In the same way, man can judge the other cut- and copy-strings in the example.

The marked-up example is then passed to a *scanner*.

3.2.3 Scanner

The scanner scans a marked-up example and extracts the format and the structure information from the example. The extracted information is represented by a grammar (a hypothesis grammar) which can be easily translated into a DREAM DSD. Regardless of their representation, a grammar and its DREAM DSD are the same.

Each nonterminal of the grammar corresponds to a tag. Its definition in the form of a rule is generated on the basis of the example structure.

Each concrete string in the grammar is a terminal. Since there are two kinds of concrete strings: cut-strings and copy-strings which play different roles in the learning process, it is necessary to distinguish them in a grammar rule, obviously. If the right-hand side (RHS) of a rule contains other nonterminals, the terminals (if they exist) in the rule are cut-strings; if the RHS of a rule contains only one terminal without nonterminals, the terminal is a copy-string.

3.2.4 Learning

With the initial grammar DREAM is obviously able to parse and to mark up exactly the original example. In order to mark up subsequent examples, the grammar has to be abstracted such that DREAM can parse different contents and slightly different structures. For this purpose the version space technique is applied to the current domain knowledge (a set of predefined abstract strings) for abstracting the terminals (*content abstraction*). Original strings from the document are matched by the abstract strings with a heuristic search procedure. The abstract strings will cover more strings than the original one.

In order to reflect structural deviations, such as missing or multiply occurring elements, a generalization subroutine is called which unifies a new grammar with the grammar acquired

from previous examples (initially empty) and generalizes them (*structure unification & abstraction*) with rewrite rules. The new grammar is stored and can be used to parse a new example, whereby a new markup cycle could be started.

The acquired structure of a concrete document can be regarded as a prototype of a class of document structures. Although one cannot determine all features of a class after analyzing only a few examples, the structures derived from them give important clues for the description of the class.

3.3 Demonstration of *MarkItUp!*

To demonstrate the *MarkItUp!* approach, suppose that the user wants to structure a portion of a document. The portion contains two examples. One example is Document-Sample 3.2 in Section 3.1.3, the other is as follows:

Document-Sample 3.3

```
\□pr\n
!□Robert□Abarbanel\n
@□Intellicorp\n
"□Connections, Perspective and Reformation□"\n
/□ACM SIGMOD 87□*□May.1987\n
>□DBDkb\n
```

where the symbols \□ at the beginning of the example indicate the document code, the code followed by the document author starting with the symbols !□, the document location starting with the symbols @□, the document title starting with the symbols "□, the document source starting with the symbols /□, and the document category starting with the symbols >□.

The user selects the first example, and marks it up manually according to SGML syntax in the structure editor (see Section 3.2.2, Figure 3.7).

The *MarkItUp!* system accepts the marked-up example and generates a grammar to describe the structure of the example. Each rule of the grammar has the form: *nonterminal* → *right-hand side of the nonterminal* (for the syntax of the grammar rule see Section 4.3.2). The initial grammar looks like the following:

Grammar-Sample 3.1

```
bibentry -> "^\\□" code "\\n^!□" author "\\n^\"□" title "□\"\\n^/□"
source "\\n^>□" category "\\n"
```



```

code -> "bk"

author -> fname "□" lname

fname -> "Suad"

lname -> "Alagic"

title -> "Object-Oriented Database Programming"

source -> publication "□*□" date

publication -> "Springer"

date -> "1989"

category -> "DBDObject"

```

where the terminals are enclosed in the quotations `"`. The nonterminals in the RHS of a rule will be further defined somewhere in the grammar.

There are two kinds of rules in the grammar: (1) the RHS of the rule contains cut-strings (terminals) and element names (nonterminals), such as the rule `bibentry` and `author`, this rule type represents a kind of structure; (2) the RHS of the rule contains only a copy-string (terminal), such as the rules `code` and `category`. Since the two kinds of rules play totally different roles in the thesis, it is necessary to formally and separately define them.

Definition 3.1 (*Structure-rule*) If the right-hand side of a grammar rule consists of nonterminal(s) or nonterminal(s) and terminal(s), the rule is called a *structure-rule*. ■

Definition 3.2 (*String-rule*) If the right-hand side of a grammar rule consists of terminal(s), the rule is called a *string-rule*. ■

The applications of the structure-rules and the string-rules will be further discussed in the following paragraphs and latter chapters.

With the initial grammar – Grammar-Sample 3.1, DREAM can exactly parse the same example but cannot properly parse another example. In order to mark up other examples, the initial grammar is abstracted further with a set of abstract strings (the details of the string abstraction see Section 4.2). The concepts used here are a subset of the example concepts in Figure 4.2, that is, the concepts applied here do not include small letter and capital letter:

Grammar-Sample 3.2

```

bibentry -> "^\\□" code "\\n^!□" author "\\n^\"□" title "□\"\\n^/□"
source "\\n^>□" category "\\n"

```

```

code -> "bk" | [a-zA-Z]+
author -> fname " " lname
fname -> "Suad" | [a-zA-Z]+
lname -> "Alagic" | [a-zA-Z]+
title -> "Object-Oriented Database Programming"
      | [a-zA-Z]+"\"([a-zA-Z]+" ")+[a-zA-Z]+
source -> publication " "* " date
publication -> "Springer" | [a-zA-Z]+
date -> "1989" | [0-9]+
category -> "DBDobject" | [a-zA-Z]+

```

where the changed parts are denoted by the bold fonts, note that in the following examples, the part with bold font in the examples always identifies some kind of difference between old and new examples (except tags); the strings consist of one of the following contents: (1) a concrete string such as `bk`; or (2) an abstract string such as `[a-zA-Z]+`. Each content matches a kind of string(s).

Comparing Grammar-Samples 3.1 and 3.2, man may find that the structure-rules in Grammar-Sample 3.1 are not changed after being abstracted by abstract strings, but the string-rules are changed, that is, besides their original strings they have an alternative on the RHS in Grammar-Sample 3.2. For example, on the RHS of the rule `code` there are two alternatives: an original string `"bk"` and an abstract string `[a-zA-Z]+`. The abstract string `[a-zA-Z]+` means that the RHS of the rule `code` can be arbitrary letters. The similar explanation is applied to the other string-rules. The original strings are kept here for two reasons: (1) showing the original strings to the user; and (2) ensuring the grammar can exactly mark up the old example. When more than one different examples have been learned, there are no concrete strings in the old string-rules (see Grammar-Sample 3.3 at below).

Now a new example (Document-Sample 3.3) is sent to the parser that uses the existing grammar – Grammar-Sample 3.2 to tag the new example, the result will be:

DSD-Sample 3.3

```

<!DOCTYPE bibdoc>
<bibentry><code>pr</code>
<author><fname>Robert</fname>

```

```

<lname>Abarbanel</lname>
</author>
<title><anything>@␣Intellicorp</anything>
</title>
<source><anything>Connections, Perspective and Reforma-
tion</anything>
<publication>ACM</publication>
<date><anything> SIGMOD 87</anything>
</date>
</source>
<category><anything>May.1987</anything>
DBDkb</category>
</bibentry>

```

DREAM cannot tag the element starting with the symbols @␣ since there is no such element in Document-Sample 3.2 and cannot correctly tag the elements `title`, `publication`, and `date` since they have different contents which are not covered by the string abstractions in Grammar-Sample 3.2. It marks the unknown elements up using a special tag name *anything* which tells the user that the tagged is an unrecognizable element in the example. The italic fonts used in the example and the following DSD-Sample 3.4 show a learning or a learned portion in the examples.

For the recognizable elements, DREAM automatically throws away cut-strings and marks up copy-strings appropriately. For the unrecognizable strings or incorrect elements, the user uses the structure editor for correcting the markups of the example.

DSD-Sample 3.4

```

<!DOCTYPE bibdoc>
<bibentry><code>pr</code>
<author><fname>Robert</fname>␣<lname>Abarbanel</lname></author>
@␣<location>Intellicorp</location>
<title>Connections, Perspective and Reformation</title>
<source><publication>ACM SIGMOD 87</publication>
<date>May.1987</date></source>
<category>DBDkb</category>
</bibentry>

```

The corrected example is not a correct SGML-document because it contains cut-strings @␣ to provide some learning information for a new learning cycle. When the system learns

the corrected example, the cut-strings can be automatically identified and added into a new grammar, that is, a complete description of the new element *location* can be captured from the example. However, a simple SGML-document cannot give such information to the learning cycle. Therefore, at the beginning, the cut-string cannot be filtered out. After the corrected example has been learned, the user deletes the cut-string to aim at a SGML-document.

The corrected example is sent into a new learning cycle. There are two alternative strategies to learn the corrected example: one is an exhaustive learning strategy – learning from an entire example; the other is a partial learning strategy – learning from those elements in the example which are deviations from the existing elements in the grammar. For the details of these strategies see Section 6.6.2.

The new grammar of DSD-Sample 3.4 is abstracted and combined with the Grammar-Sample 3.2 to generate a unified grammar – Grammar-Sample 3.3:

Grammar-Sample 3.3

```

bibentry ->  "^\\□"  code  "\\n^!□"  author  ("\\n^@□"  location)?
            "\\n^"□"  title  "□\\"n^/□"  source  "\\n^>□"  category  "\\n"

code -> [a-zA-Z]+

author -> fname "□" lname

fname -> [a-zA-Z]+

lname -> [a-zA-Z]+

location -> "Intellicorp" | [a-zA-Z]+

title -> [a-zA-Z]+ "\\-" ([a-zA-Z]+ "□")+ [a-zA-Z]+ |
        [a-zA-Z]+ [., ]+ ([a-zA-Z]+ "□")+ [a-zA-Z]+

source -> publication "□*□" date

publication -> [a-zA-Z]+ | ([a-zA-Z]+ "□")+ [0-9]+

date -> [0-9]+ | [a-zA-Z]+ [., ] [0-9]+

category -> [a-zA-Z]+

```

Grammar-Sample 3.3 is more general than Grammar-Sample 3.2. The differences between Grammar-Samples 3.2 and 3.3 are: (1) there is a new rule **location** in Grammar-Sample 3.3 and the element **location** is an optional element, denoted by `("\\n^@□" location)?` in the rule **bibentry**, because the element **location** occur only in Document-Sample 3.3 but not in Document-Sample 3.2; (2) except the new rule **location**, there are no other concrete

strings in Grammar-Sample 3.3, since the old rules have learned two different examples; (3) the right-hand side of the rules `title`, `publication`, and `date` have new alternative abstractions since their old abstractions cannot recognize the new strings in Document-Sample 3.3.

After finishing the above processes, a new markup cycle could start when the user selects another new example.

3.4 Summary

The main characteristic of the *MarkItUp!* learning cycle is an incremental learning which combines manual markup and automated markup methods. The motivation of the learning approach is to make the task easier for the user who wants to structure documents. The *user* needs only to provide an idea of how s/he expects a formatted document to be structured and the *system* synthesizes a recognition program from this information. The details of the system learning and synthesizing methods are discussed in Chapter 4.

Learning

This chapter formalizes the learning approaches and explains their basic properties. It details the problems associated with abstracting the concrete strings in documents and with learning deviating structures in documents.

4.1 Learning Problems and Learning Levels

Learning in *MarkItUp!* is isolating the differences and extracting the common features from source documents.

The most obvious difference between individual documents is their contents. To be able to cover similar contents (strings) in the subdocuments, an approach is needed to match different contents in documents. In order to also accept and structure similar documents which deviate from the learned documents structure, an approach is needed to learn the new logical structures.

In *MarkItUp!* a grammar is used to describe the structure of documents. The *terminals* of the grammar specify a *content level* that expresses the concrete strings in the document; and the *nonterminals* of the grammar specify a *structure level* that describes the documents' logical structure. Thus, learning in *MarkItUp!* is carried out at the two levels. That is, the terminals in the grammar are abstracted at the content level and the nonterminals in the grammar are generalized at the structure level.

Figure 4.1 presents a high-level description of the learnings at the two levels in *MarkItUp!*. When the grammar contains concrete strings, content abstraction will be carried out. The concrete strings are replaced by a set of *string patterns*. When the system cannot correctly mark up a new example with the existing grammar, the user has to provide proper information to structure the example, then the existing grammar is merged with the grammar generated from the updated example. After the merging, a new grammar is inferred.

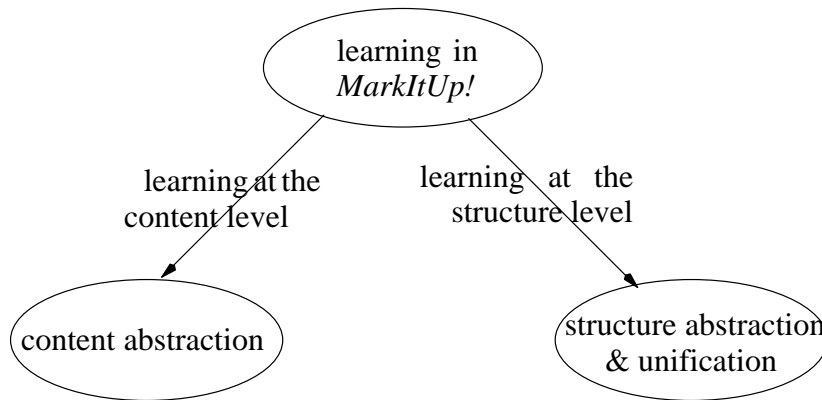


Fig. 4.1 *MarkItUp!* learning

4.2 Learning at Content Level

4.2.1 Goals, Problems and Overall Approach

Learning at the content level means abstracting sequences of terminals to form string patterns. The aim of abstracting a string is to arrive at rules which also accept similar strings; that is, the strings which are matched by the same string patterns. To reach such a goal, the version-space method [35, 26] (see Section 7.2.2) is adopted. With aid of such method the concrete strings are generalized on the basis of partial ordering domain knowledge components, which are called *concepts* (see Definition 4.1 in Section 4.2.2).

The learning problem at the content level can be summarized as follows.

Input:

A set of example strings.

Output:

A list of abstractions within the provided concepts that are matched with the presented example strings.

Techniques:

- (1) A representation of a concept.
- (2) A set of concepts to abstract example strings.
- (3) An ordering strategy for concepts.

- (4) Algorithms to learn strings on the basis of the concepts.

The string concepts are required to identify the common characteristics of concrete strings, such as all strings of the digits, all strings of small letters, etc. For this reason, the concepts should satisfy the following two requirements:

- (1) Expressiveness: Being able to denote a set of strings showing some characteristics, for example, the strings starting with a capital letter.
- (2) Tuneability: Being able to capture the main characteristics of the document strings; that is, the concepts should be able to distinguish between strings of different kinds. For instance, an e-mail address is always required to contain the symbol @, whereas a normal post address has not such a requirement. Therefore, for e-mail documents the symbol @ must be one of the concept in its domain knowledge, but for the normal post documents there may be no such concept contained in its domain knowledge.

Expressiveness refers to the problem of how to represent a concept (Section 4.2.2). Tuneability refers to the problem of what kind of concepts are suitable for individual documents (Section 4.2.3). Besides representing and defining concepts, the problems are how to organize the concepts in the *MarkItUp!* system (Section 4.2.4) and how to use the concepts to abstract strings (Section 4.2.5).

4.2.2 Concepts and Binary Relation

Since this thesis is not interested in analyzing the semantics of the document but only the syntactic structure, syntactic concepts like `<digit>` or `<letter>` (Figure 4.2) suffice for these purposes. These syntactic concepts can be easily described by regular expressions. Regular expressions can express what strings can appear in documents and search for the strings in the documents. Especially, regular expressions have an important property based on the more-specific-than binary relation (Definition 2.1). It means that two regular expressions r and s fulfill the binary relation $r \leq s$, if and only if r matches a subset of all the strings which s matches.

Note that the more-specific-than relation is defined in terms of the denotations of expressions in the representation language, and not the expressions themselves. To practically compute a more-specific-than relation by a computer program, it must be possible to determine whether r is more specific than or equal to s by examining the expressions of r and s , rather

than computing the (possible infinite) sets of examples which they match. For regular expressions, the relation is practically computed by means of DFAs/NFAs, that is, *automata theory* [27] is the theoretical foundation for applying regular expressions in *MarkItUp!*.

The more-specific-than relation however has the property that given any two regular expressions man cannot always find the one that is more specific than the other although they may have some common strings that can be matched. For instance, two regular expressions `[Ff]rom` (it can match strings “From” and “from”) and `From*` (it can match the strings starting with “Fro” and followed by arbitrarily many characters ‘m’) can be both applied to one specific string “From” without the requirement that one of them applies to every string the other applies to.

However, the relation has the *transitivity* property over a regular expressions set \mathfrak{R} (Section 2.1.3, Theorem 2.4). This property is important. With such a property the relation provides a powerful basis for determining concepts, ordering concepts and deciding string matching strategies in the domain knowledge of *MarkItUp!*.

On the basis of the above discussions, it is able to give a definition for concepts in this thesis.

Definition 4.1 (*Concept*) A *concept* is a domain knowledge component that denotes a set of strings. It is defined and supplied by the user and represented by a regular expression. ■

4.2.3 Determining Concepts

The requirement of *tuneability* discussed in Section 4.2.1 is a condition which must be considered when the concepts for a class of documents are defined.

At first sight, a natural and an attractive idea is that from example strings one directly induces a concept matching the example strings. In other words, the idea is to infer a *smallest* finite automaton of a concept which is compatible with a given finite sample consisting of a finite set of strings marked as “accepted” and another finite set of strings marked as *rejected*. But it can be shown that this idea is an NP-hard problem [18, 6].

However, it is possible to find *reasonable* concepts, that match the given example strings, from a set of *given* concepts defined by the user. The restriction is to seek a *specific* concept from the reasonable concepts. The specific concept means that it matches not only the example strings, but also a small amount of other strings which are not elements in the set of the

example strings. The reason of the restriction is that if the user defines a general concept, for example the concept $[a-zA-Z0-9]^+$, it may match a large amount of strings but hardly distinguish some strings, such as “1994” and “abc”. Therefore, the concept is not of any help for the string recognition. For this reason, determining *specific* concepts is very important although it is not easy.

The following sections will discuss how to find the reasonable concepts from a set of concepts defined by the user and how to select a specific concept from the reasonable concepts.

4.2.4 Ordering Concepts

Since there exist two kinds of concepts: comparable and incomparable, it is not easy to directly order a set of given concepts. For this reason, the concepts are organized in a *concept base* in the form of a *directed acyclic graph* (a DAG). A DAG gives a picture of what kind of relationships (comparable or incomparable) exists among the concepts. On the basis of the DAG, using the *Topological Sorting* algorithm [2], it is easy to get a linear ordered list of the given concepts. With the help of this list a specific concept can be derived.

4.2.4.1 Concept Base

An efficient organization of the concept base lies in observing how the more-specific-than relation is defined on the concept base. Suppose that the input concepts are c_1, c_2, \dots, c_n . Then on the basis of the given concepts and the more-specific-than relation among them, a concept base is built in the form DAG G as follows.

Let a concept base be a DAG $G, G = (C, E)$, where

$$C = (c_1, c_2, \dots, c_n), c_i \neq c_j (i \neq j), 1 \leq i, j \leq n,$$

$$E = \{(c_i, c_j) \mid c_i, c_j \in C, c_j < c_i \text{ and } c_i \text{ is a direct ancestor of } c_j\}.$$

This construction shows that if two concepts r and $s, r < s$, then there is an edge from s to $r, s \rightarrow r$ in the concept base. In other words, comparable concepts are connected with edges in the concept base.

Figure 4.2 gives an example of the concept base. To explain the meaning of the node (concept), in this concept base each node has two parts: the name of the node and the syntactical expression of the node. Actually, the node's name does not appear in the concept base. The

name of each node is enclosed by angled brackets `<` and `>`, and the syntactical expression of each node is enclosed by parentheses `(` and `)`. For instance, the node `<any character>` (`'.''`) expresses that the concept has a name `any character` in the concept base and its syntactical expression is a regular expression `'.'`.

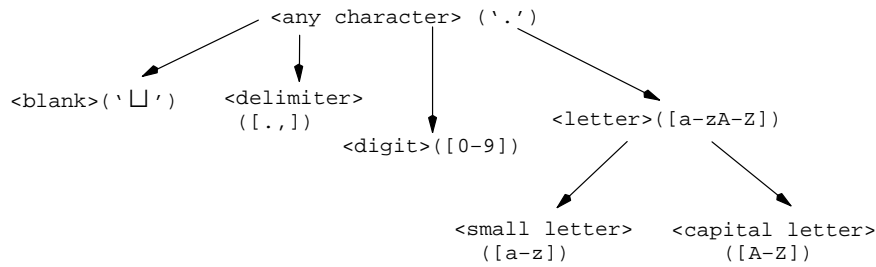


Fig. 4.2 An example of the concept base

From Figure 4.2, we can see that `<blank>`, `<delimiter>`, `<digit>`, `<small letter>`, and `<capital letter>` (as well as `<blank>`, `<delimiter>`, `<digit>`, and `<letter>`) are incomparable concepts. However, `<small letter>` and `<letter>` (as well as `<capital letter>` and `<letter>`) are comparable concepts. Furthermore, the concept `<small letter>` is more-specific-than the concept `<letter>` (as well as the concept `<capital letter>` is more-specific-than the concept `<letter>`). The concept `<any character>` is comparable with every other concept in the figure and each concept is more-specific-than `<any character>`; that is, the concept `<any character>` is an ancestor of every other concept in the concept base.

For various documents their concepts can be different. But the method to create the concept base is the same.

4.2.4.2 Linear Ordering List of Concepts

On the basis of the concept base, the concepts can be ordered by the topological sort algorithm. A topological sort of a DAG is any ordering m_1, m_2, \dots, m_k of the nodes of the graph such that edges go from nodes earlier in the ordering to later nodes; that is, if $m_i \rightarrow m_j$ is an edge from m_i to m_j , then m_i appears before m_j in the ordering. Since incomparable concepts cannot be ordered as comparable concepts, the ordering of the concepts is not unique. However, in every ordering, if $m_i \rightarrow m_j$, then the sequence always holds: m_i appears before m_j .

Aho, Hopcroft, and Ullman (reference [2], page 222) give an algorithm to do the topological sort with a *depth-first* search procedure.

After the topological sort of the concept base, we arrive at a linear ordering list of the concepts which is in reverse topological order, but meets our needs. For instance, the ordered list C of the concept base shown in Figure 4.2 is:

$C = \langle \text{blank} \rangle, \langle \text{delimiter} \rangle, \langle \text{digit} \rangle, \langle \text{small letter} \rangle, \langle \text{capital letter} \rangle, \langle \text{letter} \rangle, \langle \text{any character} \rangle$

Since the topological sort algorithm does not order incomparable concepts, such as the concepts $\langle \text{blank} \rangle$, $\langle \text{delimiter} \rangle$, $\langle \text{digit} \rangle$, $\langle \text{small letter} \rangle$, and $\langle \text{capital letter} \rangle$. The positions of the incomparable concepts in the ordered list are changeable. In other words, it is able to get various ordered lists for the incomparable concepts. However, because the concept $\langle \text{small letter} \rangle$ is more-specific-than the concept $\langle \text{letter} \rangle$ and the concept $\langle \text{capital letter} \rangle$ is more-specific-than the concept $\langle \text{letter} \rangle$, the concepts $\langle \text{small letter} \rangle$ and $\langle \text{capital letter} \rangle$ must appear before the concept $\langle \text{letter} \rangle$ in any ordered list of the above concepts as well as in the above ordered list C .

Using the ordered list, string abstracting can now be carried out.

4.2.5 Learning from Strings

A *string* is a list of characters. Two kinds of strings have been introduced in Section 3.2.2: copy-strings and cut-strings. Because they play different roles in the learning process, the two kinds of strings are abstracted by different strategies.

Copy-strings record the contents in the document which will be mapped into the output document. However, for different documents copy-strings are very different. Therefore, they need to be abstracted so that the abstracted copy-strings are able to mark up documents with the same structure, but different contents.

Cut-strings will be filtered out during the process of mark up. They usually serve as indicators for the beginning or the end of an element. Abstracting these delimiters too much would result in an ambiguous grammar. Therefore, cut-strings are abstracted much less than copy-strings.

The following two subsections will discuss the problems of learning from copy-strings and cut-strings, respectively.

4.2.5.1 Learning from Copy-Strings

Copy-strings represent concrete strings which will be transported from one document to another. Therefore, a copy-string is assumed that it consists of a variable number of characters which play the same roles in the document, that is, there is no special character which has a special meaning in the document and is required to be identified especially. In other words, the following two hypotheses are reasonable:

- (1) the characters in a copy-string are at the same level;
- (2) the number of characters in the copy-strings does not influence the abstracted result.

These two hypotheses imply that a string pattern can be inferred which matches the copy-string. Under these two hypotheses, abstracting a copy-string includes two steps:

- (1) to abstract each character in the copy-string on the basis of the ordered list of concepts;
- (2) to infer a string pattern based on the abstractions of the characters in the string.

The following algorithm 4.1 will discuss how to abstract a copy-string S . Suppose that S consists of a set of characters s_l , $1 \leq l \leq m$, that is, $S = \{s_1, s_2, \dots, s_m\}$. On the basis of an ordered list of the concepts $C = c_1, c_2, \dots, c_n$, the algorithm tries to abstract each character in the string S and stores the abstracted result at first in a *sequence* which consists of abstract concepts and/or concrete characters (unmatched characters). The algorithm starts with the first character in the string S . If one character is accepted by a concept c_i , $1 \leq i \leq n$, c_i is added into the sequence; If there is no concept matching with the character, the algorithm directly adds the character in the sequence as a default most specific concept for the string. When all characters in the string S have been parsed, the sequence is further abstracted to form a *string pattern*. For instance, subsequent occurrences of a concept (e.g. $[a-z][a-z][a-z]$) and subsequent occurrences of concepts (e.g. $[A-Z][a-z]^+ \sqcup [A-Z][a-z]^+ \sqcup$) in the sequence are transformed into an arbitrarily long sequence ($[a-z]^+$) and an arbitrarily long sequence ($([A-Z][a-z]^+ \sqcup)^+$) in the string pattern, respectively. The aim of the algorithm is to find a *specific* string pattern for the string S .

Algorithm 4.1 Abstracting a copy-string based on an ordered list of the concepts

Input. A copy-string $S = s_1, s_2, \dots, s_m$ and an ordered list of the concepts $C = c_1, c_2, \dots, c_n$.

Output. A string pattern p which accepts the string S .

Method. Parse S from left to right. For each character in S , scan the list C from left to right, choose the *first* element c_i (if any exists) in C which accepts the character. Then c_i is a specific

concept of the character. If c_i does not exist, the algorithm keeps the character in the string pattern p as the default most specific concept. When the whole string is parsed, the parsed result is abstracted and the algorithm outputs the abstracted string pattern p . The formal description of the algorithm is given as follows. Note that the italic fonts in the body of the algorithms express *variables*, *parameters*, or *functions*; while the bold fonts express *key words*.

```

1  begin
2     $p := \text{null};$ 
3     $b := \text{false};$ 
        //b is a global variable. If there exist unmatched characters in the string, it is true.
        //It is tested in Algorithm 4.2.
4    for ( $l = 1; l \leq \text{the length of } S; l++$ ) do           // parse string from left to right.
5    begin
6        for ( $i = 1; i \leq n; i++$ ) do           //i is used to count the length of the list C.
7            if  $c_i$  accepts  $s_l$  then
8                begin
9                    add  $c_i$  into  $p$ ;
10                   break;
11                end;
12            if ( $i = n + 1$ ) then  $b := \text{true};$  //i = n + 1 means that there is no concept which accepts  $s_l$ .
13        end;
14        scan  $p$  from left to right to abstract the concept sequence;
15        transform subsequent occurrences of a concept (e.g.  $[a-z][a-z][a-z]$ ) into
        an arbitrarily long sequence ( $[a-z]^+$ ) or subsequent occurrences of concepts
        (e.g.  $[A-Z][a-z]^+ \sqcup [A-Z][a-z]^+ \sqcup$ ) into an arbitrarily long sequence
        ( $([A-Z][a-z]^+ \sqcup)^+$ );
16    return (string pattern  $p$ );
17 end

```

Algorithm 4.1 gives an approach to abstract a concrete copy-string on the basis of a set of concepts. Note that each copy-string is a RHS of a string-rule in the grammar. When learning different examples, for the same string-rule (e.g. R), the concrete strings may be different. After the different copy-strings are abstracted by Algorithm 4.1, it is able to get several string patterns of the copy-strings for the rule R . A list is chosen to contain these string patterns and the list of the rule R is denoted as A_R . Of course, each string-rule in the grammar has its own *string pattern* list. These lists are the final output of the string learning.

To get the output, the focus is on the string-rule R in the grammar which has various string patterns of copy-string after learning several examples. The goal is to get the list A_R that maintains the whole learned string patterns of the rule R and has no redundant abstracted

form. The following algorithms discuss how to arrive at the goal. Before that, two definitions are given:

Definition 4.2 (*Abstracted-list A_R*) A list A_R is called an *abstracted-list* for the string-rule R , if the list contains all learned string patterns of the RHS of R . ■

Definition 4.3 (*Reduced A_R*) An abstracted-list A_R is called *reduced*, if for any two string patterns in A_R , none is more-specific-than the other. ■

In Algorithm 4.2 (see below), A_R either is an empty list or a reduced list. When learning a new copy-string E (got from another example) of the rule R , the list A_R is used to test whether the new string can be accepted by A_R . If so, there is no change for the list A_R . A_R will be used for the following examples of copy-strings again. Otherwise, the string E will be abstracted by Algorithm 4.1. and return a value to Algorithm 4.2. On the basis of the return value, the algorithm constructs a new reduced list or an error message to the user.

Algorithm 4.2 Learning from a new copy-string

Input. A new copy-string E of the rule R , an ordered list of the concepts $C = c_1, c_2, \dots, c_n$, and the reduced list $A_R = p_1, p_2, \dots, p_k$.

Output. A “new” reduced list A_R which accepts all strings accepted by the input A_R , as well as the string E ; or an error message to the user.

Method. If the reduced list A_R is not an empty list, scan the list A_R from left to right, to choose the *first* element p_j in A_R such that p_j accepts the string E , $1 \leq j \leq k$. The algorithm outputs the original A_R to the caller. Otherwise, the algorithm calls Algorithm 4.1 to abstract the string E . If the abstracted result contains concrete characters, the algorithm gives an error message to the user and the user is required to redefine the concepts. If the abstracted result is a new string pattern p_j , it will be added into the list A_R , while A_R is required to still be a reduced list. The formal description of the algorithm is given as follows.

```

1  begin
2    if  $A_R$  is not an empty list then
3      for  $j := 1$  to  $k$  do
4        begin
5          take the string pattern  $p_j$  from  $A_R$ ;
6          use the string pattern to match the string  $E$ ;
7          if (the string pattern can match the string) then return (the list  $A_R$ );
8        end;
```



```

9      call Algorithm 4.1 to abstract the string  $E$  on the basis of  $C$ , and get a return value  $p$ ;
10     if  $b$  then return (an error message to the user)
           // there is at least one character which has not been abstracted by the concepts.
11     else
12     begin
13         delete all  $p_t$  in  $A_R$ , such that  $p_t \leq p$ ,  $1 \leq t \leq k$ ;
14         add a new string pattern  $p$  into the list  $A_R$ ;
15         return (the list  $A$ );
16     end;
17 end

```

Algorithms 4.1 and 4.2 are used to abstract concrete copy-strings and to learn new copy-strings from different examples. Note that the order that examples are presented to the algorithms does not affect the result given.

4.2.5.2 Learning from Cut-Strings

For a cut-string, it may consist of key characters and/or no-key characters. The number of key characters influences the recognition of the cut-string. Therefore, the key characters in the cut-string are not allowed to be further abstracted. However, characters in sequences of trailing blanks, tabs, and returns (all shorted as S) are regarded as no-key characters. They are allowed to be abstracted into the form $(S)^+$. If there are different key words for a cut-string, they are simply kept in a list as alternatives of the cut-string.

4.2.5.3 Examples for String Abstraction

Section 3.3 has shown an example of string abstraction (Grammar-Sample 3.2) by applying Algorithms 4.1 and 4.2 with a subset of the example concepts in Figure 4.2. The following grammar shows an abstracted result by using the example concepts in Figure 4.2.

Grammar-Sample 4.1

```

bibentry -> "^\\□" code "\\n^!□" author "\\n^\"□" title "□\"\\n^/□"
source "\\n^>□" category "\\n"

code -> "bk" | [a-z]+

author -> fname "□" lname

fname -> "Suad" | [A-Z][a-z]+

lname -> "Alagic" | [A-Z][a-z]+

title -> "Object-Oriented Database Programming"
      | [A-Z][a-z]+"\\-\"([A-Z][a-z]+"□")+[A-Z][a-z]+

```

```

source -> publication "□*□" date
publication -> "Springer" | [A-Z][a-z]+
date -> "1989" | [0-9]+
category -> "DBDobject" | [A-Z]+[a-z]+

```

Comparing Grammar-Sample 3.2 (see Section 3.3) and Grammar-Sample 4.1, it is not difficult to find that the same string may have different abstracted result when it is abstracted by different concept set. And when there are more comparable concepts in the domain knowledge, the abstracted form of copy-strings will be more complex. The reason is that in this case it is easier for each character to find its own specific concept, but it is not easy to find a common concept for most characters in the string. However, since it is not so important in English to distinguish capital letter and small letter, the abstracted result of Grammar-Sample 3.2 is also acceptable.

Let's see another example for dates. Support there are four forms to express a date:

```

01.03.1994
01. 03. 1994
Jan. 03. 1994
Jan. 3rd. 1994

```

If we use the example concepts in Figure 4.2. and apply Algorithms 4.1 and 4.2 to abstract them separately, we may get the following forms:

```

date -> "01.03.1994" | ([0-9]+[.,])+[0-9]+
date -> "01. 03. 1994" | ([0-9]+[.,]"□")+ [0-9]+
date -> "Jan. 03. 1994" | [A-Z][a-z]+[.,]"□"[0-9]+[.,]"□"[0-9]+
date -> "Jan. 3rd. 1994" |
    [A-Z][a-z]+[.,]"□"[0-9][a-z]+[.,]"□"[0-9]+

```

The later example indicates another fact. That is, different representations of a string lead to different string patterns.

4.3 Learning at Structure Level

4.3.1 Goals, Problems and Overall Approach

The goal of learning at the structure level is to incrementally construct a grammar from a finite number of structured examples, in such a way that similar but not necessarily identical struc-

tures of different examples can be recognized and unified automatically. To arrive at the goal the grammar is generalized by a set of generalization rules. The learning problem at the structure level can be summarized as:

Input:

- (1) A logical structure description of a new example.
- (2) A logical structure description of previous examples (initially empty).

Output:

A *general* logical structure description that is consistent with all the presented sample structures.

Techniques:

- (1) A representation of logical structure description.
- (2) A method to abstract and unify the old and new logical structure description.

Similar to the content level, the first problem is how to describe the logical structure of documents. Then it is necessary to decide the generalizing strategies of the structures learned from examples. These two problems are dependent, that is, different descriptions of document logical structure correspond to different generalizing strategies. They will be discussed in the following subsections. Section 4.3.2 gives a formal description of the document logical structure. Section 4.3.3 discusses the strategies to generalize the structures learned from examples. Section 4.3.4 gives control strategies to use the learning rules and shows some examples using the rules.

4.3.2 Representation of Document Logical Structure

MarkItUp! uses a subset of the SGML grammar [5] – also called *document type definitions* (DTDs) – for representing the logical structure of input documents. The simplified SGML grammar is a regular grammar and consists of several rules (productions) which break down logical elements into more simple elements. The syntactical form of each rule is as follows:

$$\text{element_name} \rightarrow \text{rexpr}(e_1, e_2, \dots, e_k)$$

where,

- `element_name` (nonterminal) is a non-empty string;

- rexpr is a regular expression on elements (parameters) e_1, e_2, \dots, e_k and $e_i \neq \varepsilon$ for all $i, 1 \leq i \leq k$;
- each of e_1, e_2, \dots, e_k is either a regular expression or a new `element_name` that must be defined by a new rule somewhere in the grammar. This ensures that the rules are non-recursive.

The elements e_1, e_2, \dots, e_k use a number of operators for *sequence elements* (`()`), *repetition elements* (`+`), *iteration elements* (`*`), *optional elements* (`?`), and *alternative elements* (`|`). Where the plus sign `+`, the asterisk or star `*`, and the question mark `?` are three *occurrence indicators*.

- `+` *Required and repeatable*: the element occurs one or more times
- `*` *Arbitrarily repeatable*: the element appears zero or more times
- `?` *Optional*: the element appears zero or one times

Parentheses (`()` and `{}|`) are used to group expressions, and items within groups occur according to the *connectors* used as follows:

- `,` *Sequence*: the elements must occur in the specified order
- `|` *Alternative*: exactly one of the connected elements may occur

For saving some parentheses the binding precedence (`? + * | ,`) is assumed.

Currently a restriction beyond the regularity of the simplified SGML grammar is the lack of support of *sequences of elements with arbitrary ordering* (the connector operator is the symbol ampersand `&`, it means that all elements in a group must occur but can be in any order). Although this feature can be useful as a goal for structural abstraction (see below), general purpose rules for unifying and abstracting permutation sequences tend to perform too badly to be incorporated into the highly interactive mark up process. The sequences of arbitrary ordering will be discussed in Chapter 5.

4.3.3 Learning Logical Structure by Rewrite Rules

When incrementally generalizing a grammar from a limited number of structured examples, four situations can occur (see below). To deal with them the corresponding rewrite rules are defined.

- (1) The same logical structure can appear in more than one example or slightly deviating logical structures can share many common elements. Keeping the structures as

alternatives is not necessary and is redundant. To get a simple expression of the logical structures, unification rules are applied.

- (2) The nested elements of a grammar rule, such as

$$A_1 \mid \dots \mid A_m \mid (B_1 \mid \dots \mid B_n) \mid C_1 \mid \dots \mid C_k,$$

should be simplified, otherwise, these will lead to a complex or unclear structure in the subsequent learning process. Simplification rules are used to deal with such cases.

- (3) Some elements in one grammar occur more than once. In this case, man can assume that the elements can occur repeatedly in the documents and adopt an abstract expression to represent it. This is done by abstraction rules. For instance, an element A is repeated two times, it may be abstracted as A^+ . After that the element may occur arbitrarily often as opposed to the actual number of occurrences in the example. Of course, the abstracted expression will lose the precision.
- (4) The last and also the most complex circumstance is that the unification rules cannot be directly applied, and the structures cannot be simplified and abstracted by using the simplification and abstract rules above. For further simplifying these structures, the abstract-merge rules are introduced.

The following sections discuss unification rules (with the name *unify*), simplification rules (with the name *simplify*), and abstraction rules (including two kinds of rules with the names *abstract* and *abstract-merge*, respectively) in detail. Each rewrite learning rule has one of the general patterns: *rule-name(parameter)* or *rule-name(parameter₁; parameter₂)*, where a *parameter* expresses a right-hand side of a grammar rule.

These learning rules lead to two distinguishing steps in the learning process: In section 4.3.3.1, *unifying* new example structures with an already existing grammar or *simplifying* an existing grammar is discussed such that the resulting grammar can recognize *exactly* the structure of the additional example and all the old ones. In section 4.3.3.2 the strict unification rules are extended to a more tolerant *merging* mechanism – abstraction rules, whereby the resulting grammar can anticipate small structural deviations.

4.3.3.1 Unification and Simplification Rules

The purpose of the unification and simplification rules are to simplify the grammar derived from structured examples. The unified or simplified grammar is *exactly* equal to the original

one, that is, the unified or simplified grammar recognizes and describes the same structure of documents as the non-simplified grammar does.

Rules for Unification

The most straightforward way of unifying several grammars for structured examples would be to simply form a top level disjunction. However, such a grammar would soon get highly redundant and, more gravely, would give no clue for further abstraction. The simple disjunction (enumeration) of all different example substructures of each element does not carry us much further. Each element would be defined as an alternative of highly overlapping sequences. The *unification* rules described in the following *merge* new structures with existing element rules such that the *commonalities* are represented only once, and the *structural deviations* are made explicit.

The general pattern of all unification rules is: $unify(old; new)$. The rules express the minimal language that contains the languages defined by *old* and *new*. The parameter *old* is the already acquired definition of some elements, and the parameter *new* is the sequence of elements derived from the new example (possibly empty).

Let \mathfrak{R} be a set of regular expressions. The formal description of the function *unify* over \mathfrak{R} is as follows: $unify(old; new) \equiv (old \mid new)$, where $old, new \in \mathfrak{R}$.

Definition 4.4 (*Trivial unification*) If $unify(old; new) = (old \mid new)$, the unification is called a *trivial unification*. ■

Rule 1 Unification of elements or sequences with an empty element ϵ :

$$unify(\epsilon; A) = unify(A; \epsilon) = A?$$

A is an arbitrary expression except ϵ .

ϵ denotes an empty element, that is, when an expression *A* exists in old (or new) examples, but does not exist in the new (or old) examples, then the non-existing expression *A* in the new (or old) examples is defined as an empty element ϵ . Such empty elements may be introduced by applying rule 4.

Rule 2 Unification of an expression with a more specific expression:

$$unify(B; A) = unify(A; B) = A \quad \text{if } B \leq A$$

A, B are arbitrary expressions.

Rule 3 Unification of optional elements:

$$\text{unify}(B; A?) = \text{unify}(A?; B) = \begin{cases} A^* & \text{if } B = A^+ \text{ or } B = A^* \\ A? & \text{if } B = A \\ (\text{unify}(A; B))? & \text{otherwise} \end{cases}$$

A, B are arbitrary expressions except $A = \varepsilon$.

Rule 4 Unification of sequences with a common prefix or suffix:

$$(a) \text{ unify}((A, B); (A, C)) = A, \text{ unify}(B; C)$$

$$(b) \text{ unify}((B, A); (C, A)) = \text{unify}(B; C), A$$

A, B, C are arbitrary expressions except $A = \varepsilon$.

Rule 5 Unification of alternatives:

$$\text{unify}(A_1 \mid \dots \mid A_n; B) = \begin{cases} A_1 \mid \dots \mid A_n \mid B & \text{if } \text{unify}(A_i; B) \text{ is } \textit{trivial} \text{ for all } i \\ C_1 \mid \dots \mid C_n \mid B & C_i = \begin{cases} \text{unify}(A_i; B) & \text{for all } i, \text{unify}(A_i; B) \text{ is non-trivial} \\ A_i & \text{otherwise} \end{cases} \end{cases}$$

A_i, B are arbitrary expressions except $B = \varepsilon$ and $A_i = \varepsilon$, for all i ($1 \leq i \leq n$).

If for all i there exist only *trivial* unifications between B and A_i , then B is simply added as an additional alternative. Otherwise, B is merged with all those A_i for which there exists a *non-trivial* unification of A_i and B .

Rule 6 Trivial unification:

$$\text{unify}(A; B) = A \mid B$$

A, B are arbitrary expressions.

Definition 4.5 (*Non-overlapping elements*) When two elements A and B are unified, and only the *trivial* unification (Rule 6) can be applied, the elements A and B are called *non-overlapping*. ■

By applying the rules *exhaustively*, in order of their specification, man arrives at grammars in which the elements are non-overlapping. Otherwise, they can be further simplified by one of unification rules (except Rule 6).

Rules for Simplification

The application of Rules 2-6, as will be discussed in Section 4.3.4, or an intellectual modification of the generated grammar may lead to nested expressions. A number of simplification rules are applied using the associativity of sequence and alternative for flattening.

There is only one parameter in the simplification rules. The general pattern of the simplification rules is $simplify(x)$ which gives a simplified representation of element x . Over \mathfrak{R} , a formal description of the function $simplify$ is: $simplify(x) \equiv x, x \in \mathfrak{R}$.

Rule 7 Simplification of operators:

$$(a) \text{ simplify}((A^*)?) = A^*$$

$$(b) \text{ simplify}((A^*)^*) = A^*$$

$$(c) \text{ simplify}((A+?) = A^*$$

$$(d) \text{ simplify}((A^*)+) = A^*$$

$$(e) \text{ simplify}((A+)^*) = A^*$$

$$(f) \text{ simplify}((A?)?) = A?$$

$$(g) \text{ simplify}((A?)^*) = A^*$$

A is an arbitrary expression except $A = \epsilon$.

Rule 8 Simplification of optional elements:

$$\text{simplify}((A_1 \mid \dots \mid A_m)?) = A_1? \mid \dots \mid A_m?$$

Rule 9 Simplification of sequences:

$$\text{simplify}(A_1, \dots, A_m, (B_1, \dots, B_n), C_1, \dots, C_k) = A_1, \dots, A_m, B_1, \dots, B_n, C_1, \dots, C_k$$

A_i, B_j, C_h are arbitrary nonempty elements for all i, j , and h

$$(1 \leq i \leq m, 1 \leq j \leq n, \text{ and } 1 \leq h \leq k).$$

Rule 10 Simplification of alternative elements:

$$\text{simplify}(A_1 \mid \dots \mid A_m \mid (B_1 \mid \dots \mid B_n) \mid C_1 \mid \dots \mid C_k) = A_1 \mid \dots \mid A_m \mid B_1 \mid \dots \mid B_n \mid C_1 \mid \dots \mid C_k$$

A_i, B_j, C_h are arbitrary nonempty elements for all i, j , and h

$$(1 \leq i \leq m, 1 \leq j \leq n, \text{ and } 1 \leq h \leq k).$$

Since the unification (also simplification) rules are disjoint and the input and output of the rules are equivalent, the sequence in which the examples are used does not influence the final result.

4.3.3.2 Abstraction Rules

The aim of the abstraction rules is to induce a new grammar from structured examples so that the resulting grammar rules can recognize more of the document-structures than what the

original grammar rules have been able to recognize. Two kinds of abstractions are distinguished:

- The first one is an extension of the unification rules. It will be applied during merging when only *trivial* unification (Rule 6) is possible. It has two parameters with the general pattern: $abstract_merge(old \mid new)$. The abstract-merge function over \mathfrak{R} can be formally depicted as: $(old \mid new) < abstract_merge(old \mid new)$, where $old, new \in \mathfrak{R}$.
- The second one is an extension of the simplification rules. It is applied *before* an example grammar unifies with another example grammar or *after* the example has been merged into the old grammar to further simplify the inferred grammar. It has only one parameter with the general pattern $abstract(x)$. The formal description of the abstraction function over \mathfrak{R} is:

$$x < abstract(x), x \in \mathfrak{R}.$$

Rules for Abstraction

For merging new examples with the existing grammar in a more tolerant way than in the cases where only the trivial unification exists, five abstraction rules are introduced: Whereas the unification Rules 4(a) and 4(b) merge only sequences with a common prefix or suffix, the abstraction Rules 11(a)-(b) merge sequences with a comparable prefix or suffix (where $A' < A$), the abstraction Rules 12(a)-(c) merge sequences with a number of common subsequences interleaved with distinct subsequences.

Rule 11 Abstraction merge of comparable prefix or suffix:

$$(a) \quad abstract_merge(A, B; A', C) = \max(A', A), unify(B; C) \quad \text{if } A' \asymp A$$

$$(b) \quad abstract_merge(B, A; C, A') = unify(B; C), \max(A', A) \quad \text{if } A' \asymp A$$

where A, A', B, C are arbitrary expressions, except $A = \epsilon$ and $A' = \epsilon$;

$$\max(A', A) = \begin{cases} A' & \text{if } A < A' \\ A & \text{if } A' < A \end{cases}$$

This rule is an extension of Rule 4. It considers the case $A' \asymp A$. For example, for the two elements ac^* and bc^+ , a suitable unification rule cannot be found to simplify them (except Rule 6). But they can be abstracted by Rule 11. The abstraction result is $(a \mid b)c^*$ which is not equal to the original expression $ac^* \mid bc^+$, since $(a \mid b)c^*$ includes the element b , but $ac^* \mid bc^+$ does not.

When there exists no rule to be used in merging new examples with existing grammar rules (except Rule 6), Rule 12 adopts the methods of defining new optional elements or of dividing sequences into two parts to get non-trivial unification subsequences.

Rule 12 Abstraction merge of sequences:

- (a) $abstract_merge(A, B_1, \dots, B_m; C_1, \dots, C_n) = A?, unify(B_1, \dots, B_m; C_1, \dots, C_n)$
when $unify(B_1, \dots, B_m; C_1, \dots, C_n)$ is non-trivial.

Likewise, there are two rules (b) and (c).

- (b) $abstract_merge(B_1, \dots, B_m, A; C_1, \dots, C_n) = unify(B_1, \dots, B_m; C_1, \dots, C_n), A?$
when $unify(B_1, \dots, B_m; C_1, \dots, C_n)$ is non-trivial.
- (c) $abstract_merge(A_1, \dots, A_h, B_1, \dots, B_m; D_1, \dots, D_k, C_1, \dots, C_n)$
 $= unify(A_1, \dots, A_h; D_1, \dots, D_k), unify(B_1, \dots, B_m; C_1, \dots, C_n)$
when at least one of $unify(A_1, \dots, A_h; D_1, \dots, D_k)$ and $unify(B_1, \dots, B_m; C_1, \dots, C_n)$ is non-trivial.

To further simplify the grammar in the second abstraction case, Rule 13 performs a grouping on finite sequences of consecutive equal subsequences in a similar way as for abstraction at the content level.

Rule 13 Abstraction of repeated elements

$$abstract(A, (B_1, \dots, B_m), (B_1, \dots, B_m), \dots, (B_1, \dots, B_m), C) = A, (B_1, \dots, B_m)^+, C$$

A, B_i, C are any expressions, except $B_i = \epsilon$, for all i ($1 \leq i \leq m$).

Although the abstraction rules are disjoint, the input and output of the rules are not equivalent. Therefore, the final result is dependant on the sequence of examples.

4.3.4 Applying the Learning Rules

The four types of learning rules have been discussed in Section 4.3.3. They will be used for dealing with the different cases in the learning process. Now the problem is *when* and *how* they are applied during learning cycles. The following sections give some control strategies for applying the learning rules and show some examples to explain how the rules are executed.

4.3.4.1 Control Strategies for Applying the Learning Rules

One reason for applying the learning rules is to make the form of the learned grammar simple and exact. Of course, the two aspects of simplification and exactness are mostly contradictory

in the learning problem. In this thesis, simplification is considered as the major aim. For this reason, when two grammar rules are unified, the simplification rules and the abstraction rules will be applied at first. After that the unification rules, and then the abstract-merge rules are chosen. By this requirement, an ordering is described as follows:

Strategy: The control strategy for applying the learning rules

- 1 apply the *simplify* rules and/or the *abstract* rule to the grammar rules;
- 2 unify the old and new grammar rules with the *unify* rules; **If** only the *trivial* unification is possible, **go to** 3, otherwise unify them and then **stop**;
- 3 abstract the old and new grammar rules with *abstract-merge* rules (call **Rule 6_{ext}**).

For applying the abstraction rules, an extension rule of Rule 6 is added:

Rule 6_{ext}: The control strategy for applying *abstract-merge* rules

When two rules are unified and only the *trivial* unification (Rule 6) is possible, one of the abstract-merge rules is tentatively applied as follows:

- 1 **if** Rule 11 can be applied **then** abstract the rules with Rule 11 **else**
- 2 **begin**
- 3 find incomparable subsequences *C* such that
the remaining subsequences of the input rules are non-trivial;
- 4 **if** *C* is a subsequence at the beginning of one of the input rules **then** apply Rule 12(a)
- 5 **else if** *C* is a subsequence at the end of one of the input rules **then** apply Rule 12(b)
- 6 **else** apply Rule 12(c);
- 7 apply the unification rules to the abstracted rules further and then **stop**;
- 8 **end**

4.3.4.2 Some Examples for the Learning Rules

The following examples illustrate the usage of the above learning rules. Note that the element name in SGML is limited to 8 characters. To more easily read the following examples, any number of characters describing an element are allowed in the following examples.

Example 4.1 Abstraction of repetition elements

Let a paper element of SGML DTD be:

```
SGML-DTD 4.1 paper -> title, author, address, author, address,
                    author, address, abstract, text-param*
```

When the abstraction Rule 13 is applied to the RHS of *paper*, the following result is produced:

```
abstract(title, author, address, author, address, author,
         address, abstract, text-para*)
 $\stackrel{13}{\Rightarrow}$  title, (author, address)+, abstract, text-para*
```

Example 4.2 Unification of sequences with the maximum common prefix or suffix

Let two elements of SGML DTD paper be:

SGML-DTD 4.2 paper -> title, author, address, abstract, text-para

SGML-DTD 4.3 paper -> title, author, address, abstract

Unify the RHSs of the two elements as follows:

```
unify(title, author, address, abstract, text-para;
      title, author, address, abstract)
 $\stackrel{4a}{\Rightarrow}$  title, author, address, abstract, unify(text-para;  $\epsilon$ )
 $\stackrel{1}{\Rightarrow}$  title, author, address, abstract, text-para?
```

The unified RHS of the element paper is:

paper -> title, author, address, abstract, text-para?

Example 4.3 Unification of alternative elements

Let an element date be defined as:

SGML-DTD 4.4 date -> day|month|year

To discuss the two cases of the unification Rule 5, another two elements of date unify with SGML-DTD 4.4:

SGML-DTD 4.5 date -> time

SGML-DTD 4.6 date -> month

(a) unifying SGML-DTD 4.4 and SGML-DTD 4.5

```
unify(day|month|year; time)
 $\stackrel{5}{\Rightarrow}$  day|month|year|time
```

Now a new SGML-DTD is generated:

SGML-DTD 4.7 `date -> day|month|year|time`

(b) unifying SGML-DTD 4.4 and SGML-DTD 4.6

`unify(day|month|year; month)`

$\stackrel{5\&2}{\Rightarrow}$ `day|month|year`

The result of the unification is the following SGML-DTD:

SGML-DTD 4.8 `date -> day|month|year`

Of course, SGML-DTDs 4.7 and 4.8 can be further unified using other unification rules, but in this example the focus is on the unification Rule 5.

Example 4.4 Unification of optional elements

From the result of Example 4.2, the element of SGML DTD `paper` is:

SGML-DTD 4.9 `paper -> title, author, address, abstract,
text-para?`

Let a new element of SGML DTD `paper` be:

SGML-DTD 4.10 `paper -> title, (author, address)+, abstract,
text-para*`

When the RHS of SGML-DTD 4.9 is unified with the RHSs of SGML-DTD 4.10, the following result is obtained:

`unify(title, author, address, abstract, text-para?;`

`title, (author, address)+, abstract, text-para*)`

$\stackrel{4a\&2}{\Rightarrow}$ `title, (author, address)+, abstract,`

`unify(text-para?; text-para*)`

$\stackrel{3}{\Rightarrow}$ `title, (author, address)+, abstract, text-para*`

Example 4.5 Abstraction merge of sequences

Let elements of SGML DTD `biography` be:

SGML-DTD 4.11 `biography -> ID, head, body, reference`

SGML-DTD 4.12 biography -> head, body

SGML-DTD 4.13 biography -> name, biogdata, body, signature

They are used to show the application of abstraction Rule 12:

(a) unifying SGML-DTD 4.11 and SGML-DTD 4.12:

$unify(ID, head, body, reference; head, body)$

$\stackrel{6_{ext}}{\Rightarrow} abstract-merge(ID, head, body, reference; head, body)$

$\stackrel{12a}{\Rightarrow} ID?, unify(head, body, reference; head, body)$

$\stackrel{4a}{\Rightarrow} ID?, head, body, unify(reference; \epsilon)$

$\stackrel{1}{\Rightarrow} ID?, head, body, reference?$

(b) unifying SGML-DTD 4.11 and SGML-DTD 4.13:

$unify(ID, head, body, reference; name, biogdata, body,$
signature)

$\stackrel{6_{ext}}{\Rightarrow} abstract-merge(ID, head, body, reference; name, biogdata,$
body, signature)

$\stackrel{12c}{\Rightarrow} unify(ID, head; name, biogdata), unify(body, reference;$
body, signature)

$\stackrel{4a\&6}{\Rightarrow} (ID, head \mid name, biogdata), body,$
 $unify(reference; signature)$

$\stackrel{6}{\Rightarrow} (ID, head \mid name, biogdata), body, (reference \mid signature)$

4.4 Summary

The learning approach discussed in this chapter covers the following characteristics: (1) Incremental learning so that the grammar can be efficiently modified using additional examples; (2) Embedding version-space methods in the grammatical inference learning cycle, with the aid of version-space methods man can generalize concepts on the basis of a set of training data and a language; (3) Isolated learning of each production; (4) Judging the positive and negative examples by the user. The learning result is a grammar which can be used to mark up an electronic document into an SGML-document with the DREAM parser.

Sequence of Arbitrary Ordering

The kind of structure-rule whose RHS consists of *ordered* elements has been discussed, that is, for this kind of rule the positions of the elements in its RHS are fixed. After learning several examples, some structure-rules may have alternative RHSs in which the positions of elements are exchanged. This chapter will discuss these RHSs that consist of the sequences whose elements have arbitrary ordering.

5.1 Problem and Goal

For a document collection (see Section 1.2) the logical elements may occur in different ordering in the subdocuments. When such subdocuments are met, man can derive a set of RHSs for a structure-rule in which the logical elements have different orderings, that is, the positions of the elements in the RHSs are not fixed. The set of RHSs of such a structure-rule is called *alternative RHSs* or *alternative sequences* of the structure-rule (or short *alternative sequences*).

A simple and typical example is the name of an author that consists of first name(s) and last name(s). The first name will be written before the last name or after the last name. When man uses a grammar rule to describe such cases, man gets a rule `author` whose RHS has two alternatives: `first-name` followed by `last-name` (denoted as `first-name, last-name`) or `last-name` followed by `first-name` (denoted as `last-name, first-name`).

When a RHS of a structure-rule consists of n elements and the positions of all elements are exchangeable, man may derive $n!$ permutation sequences of the elements, that is, the number of the alternative sequences of the elements can be at most $n!$. Although the number of alternative sequences, normally, is smaller than $n!$, enumerating all alternatives is too clumsy. Further more, it may lead to complex or unclear descriptions in the subsequent learning process or during the translation from an abstracted grammar to a DSD. For these reasons, an expression is needed to represent such alternative sequences. Fortunately, SGML provides a connector ampersand `&`, with which such alternative sequences can be described by a *general expression*. For instance, the alternative sequences of the above example can be represented by the general expression: `first-name & last-name`.

It is now easy to connect all elements in the alternative sequences by the connector ampersand & which definitely includes all actually existing alternative sequences. However, these completely unordered sequences are normally too ambiguous.

Thus the goal of this chapter is to infer a general expression to represent the alternative sequences derived from the learned examples and introduce a mechanism which identifies the minimal subsequences that can be unordered but keeps the rest of it ordered in the general expression. The way from a set of concrete alternative sequences to a general expression is therefore an abstraction learning process. The learning process will be discussed in the following sections.

The remainder of this chapter is organized as follows. Section 5.2 introduces some basic concepts and notations which will be used in the following sections. Section 5.3 discusses how to infer a general expression from the alternative sequences.

5.2 Basic Concepts and Notations

Since the focus is on the RHSs of the structure-rules, the following definitions and discussions serve for these RHSs.

Definition 5.1 (*Unordered elements and Ordered elements*) The elements e_1, e_2, \dots, e_n of a RHS of a structure-rule are *unordered elements*, if their positions can be changed in the alternative sequences of the rule. Otherwise, they are *ordered elements*. ■

Suppose there are unordered elements e_1, e_2, \dots, e_n . If the position of the element e_i ($1 \leq i \leq n$) in a sequence is denoted by a number j ($1 \leq j \leq n$), the list of the numbers, denoted by $\{1, 2, \dots, n\}$, expresses a sequence of the elements. Each permutation sequence of the elements corresponds to a list of the numbers, denoted by $\pi = [\text{the sequence of numbers}]$. For simplicity, when man discusses permutation sequences, the numbers can be used to describe the sequences instead of the concrete elements of the sequences. For instance, $\pi = [3, 4, 2, 1]$ expresses the permutation sequence e_3, e_4, e_2, e_1 . Among the permutation sequences, there exists a special sequence, which is called a *standard sequence*.

Definition 5.2 (*Standard sequence and normal sequences*) If a permutation sequence consists of elements e_1, e_2, \dots, e_n and the position of element e_i in the sequence is the number i for all i ($1 \leq i \leq n$), the sequence of the elements is called a *standard sequence*, denoted by $\pi = [1, 2, \dots, n]$. The other permutation sequences of the elements are called *normal sequences*. ■

For the alternative sequences derived from the learned examples, the sequence that appeared in the first example is appointed as the standard sequence.

In the π of the standard sequence, the numbers are ordered according to the ordering defined on natural numbers. If two numbers in a π occur out of this ordering, that is, the larger number is to the left of the smaller one in a sequence, they form an *inversion*. For example, in the permutation sequence $\pi = [2, 1, 3, 4]$, numbers 2 and 1 form an inversion.

Definition 5.3 (*General expression*) A *general expression* (GE) of numbers $\{1, 2, \dots, n\}$, denoted by a symbol η , is an abstracted expression of the alternative sequences in which the numbers are written from 1 to n and connected by parentheses ‘(’ and ‘)’, commas ‘,’ and ampersands ‘&’, where the ampersand has a higher associated precedence than the comma. ■

Note that a general expression always implies a standard sequence.

If $\pi = [4, 2, 3, 1]$, a GE of the π is inferred as $\eta := 1 \ \& \ (2, 3) \ \& \ 4$ which means that number 2 must appear before number 3 in any sequence and they can exchange the position with number 1; number 4 can exchange with number 1 to number 3, but it is not adjacent to number 3 (except in the standard sequence of the numbers). From this GE, man can derive the following normal sequences: $\pi_1 = [4, 2, 3, 1]$, $\pi_2 = [4, 1, 2, 3]$, $\pi_3 = [2, 3, 1, 4]$, $\pi_4 = [2, 3, 4, 1]$, and $\pi_5 = [1, 4, 2, 3]$, where π_2 to π_5 are extra normal sequences which do not occur before the GE is inferred. It means that a GE may include other permutation sequences which have not occurred before the GE is inferred.

5.3 Inferring a General Expression

It is not easy to infer a general expression directly from a set of alternative sequences. However, a PG (see Section 2.2.3) represents a good way to describe the positions of unordered elements in a sequence. Therefore, with the help of their PGs a general expression can be inferred from alternative sequences. The major steps to reach such the goal are:

- (1) Creating permutation graphs (PGs) of permutation sequences (Section 5.3.1);
- (2) Getting the union graph of the PGs (Section 5.3.2);
- (3) Inferring a general expression on the basis of the union graph (Section 5.3.3).

5.3.1 Constructing a Permutation Graph

The definition of permutation graphs (PGs) and the relationship between a permutation graph $G[\pi]$ and a permutation π of numbers $\{1, 2, \dots, n\}$ have been described in Section 2.2.3. On

the basis of these knowledge, a permutation graph $G[\pi]$ can be constructed from a permutation π of numbers $\{1, 2, \dots, n\}$ in the following way:

- (1) The nodes of $G[\pi]$ are numbered from 1 to n ;
- (2) Two nodes are joined by an edge if they form an inversion.

For instance, $\pi := [3, 4, 2, 1]$ contains five inversions, since in the standard sequence both number 3 and number 4 should follow number 1 and number 2, and number 2 should follow number 1. Therefore, the $G[\pi]$ in Figure 5.1 shows: both number 3 and number 4 connected to number 1 and number 2 (four edges), and number 2 connected to number 1 (one edge).

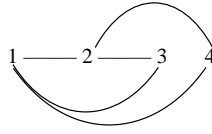


Fig. 5.1 The permutation graph of $\pi = [3, 4, 2, 1]$

Where, a continuous line “—” is used to express an inversion (normal edge) in a permutation graph.

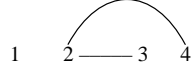
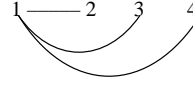
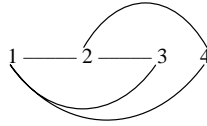
The permutation graph of the standard sequence consists of isolated nodes, that is, there is no edge among the nodes in the PG. The PG is called a *standard* PG. In the following sections, when a graph is called a permutation graph, it means that it is not a standard permutation graph.

If the numbers in a sequence are ordered differently from their ordering in the standard sequence, then this is a normal sequence. In other words, a normal sequence is always defined relative to a standard sequence. Since there is a one-to-one mapping between the permutation graph and the normal sequence, if a permutation graph is not a standard permutation graph, the graph also implies a standard permutation graph. Therefore, for a non-standard permutation graph there always exists two sequences: a standard sequence and a normal sequence.

5.3.2 Union Graph of PGs

A union graph of PGs is called an *abstraction graph* (AG) and defined as follows:

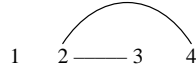
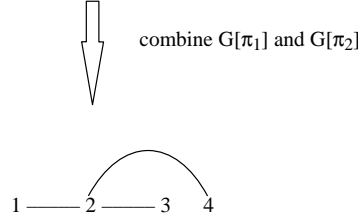
Definition 5.4 (*Abstraction graph*) If $PG_i = (V, E_i)$, $1 \leq i \leq n$, an *abstraction graph* (AG) of permutation graphs is a union graph of them so that $AG = (V, \bigcup_{i=1}^n E_i)$. ■

**Fig. 5.2a** $G[\pi_1]$, $\pi_1 = [1, 3, 4, 2]$ **Fig. 5.2b** $G[\pi_2]$, $\pi_2 = [2, 3, 4, 1]$ combine $G[\pi_1]$ and $G[\pi_2]$ **Fig. 5.2c** A new AG**Fig. 5.2** A union graph is a PG

Note that an AG may either be a permutation graph (called a *PG-AG*) or a non-permutation graph (called a *non-PG-AG*), because of two possible cases occurring when PGs are combined:

- (1) The union graph is still a PG, that is, there exists a permutation π of numbers $\{1, 2, \dots, n\}$ for the union graph (see Theorem 2.5 in Section 2.2.3);
For example, in Figure 5.2, $\pi_1 = [1, 3, 4, 2]$, $\pi_2 = [2, 3, 4, 1]$. Their corresponding permutation graphs $G[\pi_1]$ and $G[\pi_2]$ are shown in Figure 5.2a and Figure 5.2b, respectively. Combining $G[\pi_1]$ and $G[\pi_2]$ produces a new AG (Figure 5.2c). The new AG is a PG, the corresponding permutation sequence π of AG is $\pi = [3, 4, 2, 1]$. Therefore, this AG is a PG-AG.
- (2) The union graph is not a PG, that is, for the union graph there does not exist a corresponding permutation π of numbers $\{1, 2, \dots, n\}$.
For example in Figure 5.3, $\pi_1 = [1, 3, 4, 2]$, $\pi_2 = [2, 1, 3, 4]$. Their corresponding permutation graphs $G[\pi_1]$ and $G[\pi_2]$ are shown in Figure 5.3a and Figure 5.3b, their union graph (Figure 5.3c) is a new AG but it is not a PG, that is, there does not exist a π of numbers $\{1, 2, 3, 4\}$ whose permutation graph is the result of the combination. This AG thus is a non-PG-AG.

Similarly combining a PG and an AG (with the same nodes), the consequence may be a PG-AG or a non-PG-AG. Every AG, no matter whether it is either a PG-AG or a non-PG-AG, has the property described below.

**Fig. 5.3a** $G[\pi_1]$, $\pi_1 = [1, 3, 4, 2]$ **Fig. 5.3b** $G[\pi_2]$, $\pi_2 = [2, 1, 3, 4]$ **Fig. 5.3c** A new AG**Fig. 5.3** A union graph is not a PG

In a PG (or an AG) there are two kinds of nodes: (1) *isolated* node which has no edge connected with other nodes in the PG; (2) *connected* node which has at least one edge connected with another node in the PG. The connected nodes construct a *connected graph*, that is, there exist paths from one node to another node in the graph. A PG may consist of several connected sub-graphs and isolated nodes. The following lemma and corollary describe the property of the nodes in the connected sub-graphs.

Recall that each node in a permutation graph is always identified by a corresponding integer. Thus, for convenience, a set of nodes is called an *interval* if the set of integers corresponding the nodes forms an interval (of integers).

Lemma 5.1 Given a permutation graph PG , the set of nodes in each connected sub-graph of PG forms an interval.

Proof: Let $PG' = (V', E')$ be a connected sub-graph of PG . At first we show if $(i, j) \in E'$, $i < j$, then $k \in V'$ for any $i \leq k \leq j$.

Observe that $(i, j) \in E'$ implies $\pi_i^{-1} > \pi_j^{-1}$. If $\pi_i^{-1} > \pi_k^{-1}$, then $(i, k) \in E'$; if $\pi_i^{-1} \leq \pi_k^{-1}$ then $\pi_k^{-1} > \pi_j^{-1}$ and hence $(k, j) \in E'$. Thus, $k \in V'$.

Now let m and n be respectively the minimal and maximal nodes in V' . It follows that there is a path from m to n . In this path, each edge, say (i, j) , implies that $k \in V'$ for any $i \leq k \leq j$. Consequently, $k \in V'$ for any $m \leq k \leq n$. This completes the proof. ■

Corollary 5.2 Given an abstraction graph AG , the set of nodes in each connected sub-graph of AG forms an *interval*.

Proof: Since each connected sub-graph of AG is a union of several connected sub-graphs of permutation graphs, the result immediately follows from Lemma 5.1. ■

5.3.3 Deriving a General Expression from an AG

An AG may be a PG-AG or a non-PG-AG. Whatever it is, Lemma 5.1 and Corollary 5.2 ensure that the nodes in the sub-graph of the AG form an interval.

For a PG-AG, if the nodes i and j form an inversion, where $i < j$, based on the definition of PG (see Section 2.2.3) the property can be inferred: j with each node in the interval $[i+1, j-1]$ forms an inversion also. On the basis of this property and Lemma 5.1 a GE of a PG-AG can be derived.

For a non-PG-AG this property doesn't hold. Since every AG originally comes from PGs, to derive a GE of a non-PG-AG, the AG can be abstracted by using this PG's property, that is, if the above condition holds: $i < j$ and i and j form an inversion, it is assumed that j connects with each node in $[i+1, j-1]$ although some edges may not exist in the AG.

On the basis of the above discussion, the following algorithm is used to derive a GE (see Definition 5.3) of the AG.

Since edges in an AG express that the positions of nodes connected by them are exchangeable, isolated nodes denote that the positions of the nodes in a sequence are non-exchangeable, the nodes in the AG are first classified into isolated nodes and connected nodes. Each set of connected nodes consists of a connected sub-graph. From Corollary 5.2, man knows that the nodes in the connected graph form an interval. Therefore, the nodes in the connected graph can be sequentially scanned and can be connected with commas, ampersands and parentheses. Finally all sub-graphs are connected by commas according to increasing nodes' numbers.

Algorithm 5.1 Deriving a general expression from an AG

Input. An AG.

Output. A general expression of the AG.

Method. Suppose the nodes are ordered from number 1 to number n , that is, the node number stands for the position of the node in the standard sequence. Here when two nodes are ex-

changeable, it means their positions are exchangeable. On the basis of such an assumption, man can sort and group the nodes by the following steps:

- (1) Partition the AG to several sub-graphs (GH_j) which are either connected graphs or isolated nodes, that is, each sub-graph j (GH_j) is either a connected graph or an isolated node. The sub-general expression of GH_j is denoted by $\eta_{gh(j)}$. The $\eta_{gh(j)}$ of an isolated node is the number of the node and the $\eta_{gh(j)}$ of a connected graph is inferred by the following step;
- (2) For each connected graph ($V_j = [j, k]$, $1 \leq j < k \leq n$), scan the nodes from j to k and connect them using commas, ampersands and parentheses to get a sub-general expression $\eta_{gh(j)}$. Since ampersand has a higher associated precedence than comma, when a set of ordered elements will be connected with an unordered element, they are put in parentheses as an entire unit to connect with other unordered elements. For this reason the ordered elements are tried to figure out first (see the first part of Function subGE, from line 5 to line 13) and are kept in a pair of parentheses when they are connected with another unordered elements (see Function parGE).
- (3) If there exist m ($m > 1$) sub-graphs, use the commas to connect each sub-graph according to increasing nodes' number (Lemma 5.1 and Corollary 5.2 ensure that it can be done in such ordering), that is, $\eta := \eta, \eta_{gh(j)}$ ($1 \leq j \leq m$).

Routine Deriving_GE

Suppose there are m sub-graphs.

```

1  begin
2       $V := \{\text{the nodes of AG}\};$ 
3       $E := \{\text{the edges of AG}\};$ 
           //E is a global variable such that the following functions can call it
4      if AG is not a connected graph then partition the AG to several sub-graphs ( $GH_j$ );
5      for each connected sub-graph  $GH_j$  ( $V_j = \{j, j+1, \dots, k-1, k\}$ ) do  $\eta_{gh(j)} := \text{subGE}(j, k);$ 
6       $\eta := \eta_{gh(1)};$ 
7      for  $j := 2$  to  $m$  do  $\eta := \eta, \eta_{gh(j)};$ 
8      return( $\eta$ );
           //η is a general expression of the AG
9  end
```

Function subGE(startNode, endNode)

```

1  begin1 //to make readers easily match the corresponding begin and end in this function,
           // the number at the end of each begin and end identifies pairs of begin and end.
2       $\eta_{gh} := \text{startNode};$             $i := \text{startNode};$            //ηgh is used to express a GE of a sub-graph
3      while  $i < \text{endNode}$  do
4          if  $i$  connected with  $i+1$  then
5              begin2
6                  scan the nodes from left to right;
```

```

7      find the biggest node  $l \in [i+1, endNode]$  such that the nodes in  $[i+1, l]$ 
      are ordered elements and each of them is connected with  $i$ ;
8      if  $\exists$  a node  $\in [i+1, l]$  which is connected with at least one node (except  $i$ ) in  $\eta_{gh}$ 
9          then  $\eta_{gh} := parGE(\eta_{gh}, i+1, l)$ ;
10     if  $l = i+1$  then  $\eta_{gh} := \eta_{gh} \& i+1$  // to avoid unnecessary parentheses around  $l$ .
11         else  $\eta_{gh} := \eta_{gh} \& (i+1, i+2, \dots, l-1, l)$ ;
12      $i := l$ ;
13 end2
14 else
15     begin3 //  $i$  doesn't connect with  $i+1$ 
16         if  $i+1$  doesn't connected with any node in  $\eta_{gh}$ 
17             then  $\eta_{gh} := \eta_{gh}, i+1$ 
18                 //  $i+1$  as an ordered element connected with the nodes in  $[startNode, i]$ 
19             else //  $i+1$  as an unordered element connected with one of the nodes in  $[startNode, i]$ 
20                 begin4
21                      $\eta_{gh} := parGE(\eta_{gh}, i+1, l)$ ;
22                      $\eta_{gh} := \eta_{gh} \& i+1$ ;
23                 end4;
24              $i := i+1$ ;
25         end3;
26     return( $\eta_{gh}$ );
27 end1

```

Function $parGE(\eta_{gh}, startNode, endNode)$

```

1  begin
2      scan  $\eta_{gh}$  from left to right;
3      find the first node  $h$  in  $\eta_{gh}$  which is connected with one of the nodes in  $[startNode, end-$ 
Node];
      // suppose  $startNode$  is connected with  $h$ , man can infer that  $startNode$  is connected
      // with the nodes in  $[h+1, startNode-1]$  also.
4  if the  $h$  is not in a pair of parentheses ()
5  then //  $h$  is not yet classified into an entire unit
      add the left parenthesis ( to the  $\eta_{gh}$  before the node  $h$  and the right parenthesis ) to
      the end of  $\eta_{gh}$  // to create an entire unit
6  else //  $h$  has been included in an entire unit
      add the left parenthesis ( to the  $\eta_{gh}$  in front of the existed the leftest parenthesis which
      includes  $h$  and the right parenthesis) to the end of  $\eta_{gh}$ ;
      // to form a new entire unit that contains the old one
7  return( $\eta_{gh}$ )
8  end

```

The algorithm is also applicable for permutation graphs. With the above algorithm, man can infer a general expression on the basis of a PG or an AG. Now let us consider some examples to apply the above algorithm.

Example 5.1 A PG in Figure 5.4

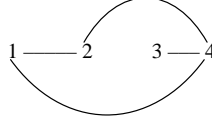


Fig. 5.4 A PG

- (1) no sub-graph
- (2) $\eta_{gh} = (1\&2, 3)\&4$
- (3) $\eta := \eta_{gh}$

The general expression denotes the permutation sequences: $\pi_1 = [1, 2, 3, 4]$, $\pi_2 = [2, 1, 3, 4]$, $\pi_3 = [4, 1, 2, 3]$, $\pi_4 = [4, 2, 1, 3]$. Where π_2 and π_3 are new normal sequences which do not appear in the existing sequences represented by Figure 5.4.

Example 5.2 A PG in Figure 5.3a

- (1) two sub-graphs, $\text{GH}_1 = \{1\}$, $\eta_{gh(1)} = 1$; $\text{GH}_2 = \{2, 3, 4\}$
- (2) $\eta_{gh(2)} = 2\&(3, 4)$
- (3) $\eta := 1, 2\&(3, 4)$

The general expression denotes the permutation sequences: $\pi_1 = [1, 2, 3, 4]$, $\pi_2 = [1, 3, 4, 2]$ which are equal to the alternative sequences represented by Figure 5.3a.

Example 5.3 A PG in Figure 5.3b

- (1) three sub-graphs, $\text{GH}_1 = \{1, 2\}$; $\text{GH}_2 = \{3\}$, $\eta_{gh(2)} = 3$; $\text{GH}_3 = \{4\}$, $\eta_{gh(3)} = 4$
- (2) $\eta_{gh(1)} = (1\&2)$
- (3) $\eta := (1\&2), 3, 4$

The general expression denotes the permutation sequences: $\pi_1 = [1, 2, 3, 4]$, $\pi_2 = [2, 1, 3, 4]$ which are equal to the alternative sequences represented by Figure 5.3b.

Example 5.4 An AG in Figure 5.3c

- (1) no sub-graph
- (2) $\eta_{gh} = 1\&2\&(3, 4)$
- (3) $\eta := \eta_{gh}$

The general expression denotes the permutation sequences: $\pi_1 = [1, 2, 3, 4]$, $\pi_2 = [2, 1, 3, 4]$, $\pi_3 = [1, 3, 4, 2]$, $\pi_4 = [2, 3, 4, 1]$, $\pi_5 = [3, 4, 1, 2]$, $\pi_6 = [3, 4, 2, 1]$. Where π_4 to π_6 are new

normal sequences which do not appear in the existing sequences represented by Figures 5.3a and 5.3b, in other words, the abstracted expression can denote more permutation sequences than exist in the original AG.

The four examples show that the general expression inferred on the basis of a PG, on the one hand, may denote more sequences than the existing sequences represented by the PG (Example 5.1), on the other hand, it can denote exactly the same sequences represented by the PG (Example 5.2 and Example 5.3). However, for an AG (Example 5.4), the general expression always implies more sequences than the existing sequences represented by the AG.

5.4 Summary

This chapter describes a learning approach to infer a general expression of sequences of elements with arbitrary ordering. The overall idea of the learning approach is to use permutation graphs describing the alternative sequences, where the edges of the permutation graphs describe the deviation among the sequences. After combining the permutation graphs, man can get a union graph that contains all edges in the permutation graphs. On the basis of the union graph man can infer a general expression of the alternative sequences.

Implementation

The first section of the chapter gives an overview of the system architecture of *MarkItUp!*. According to the architecture, the subsequent sections describe the major implemented algorithms of each component in the system. Section 6.6 discusses several examples in detail to illustrate the different strategies for incrementally learning a grammar. Section 6.7 evaluates the effectiveness of *MarkItUp!* on a bibliographic document source. Finally, Section 6.8 summarizes implementation and evaluation.

6.1 System Architecture of *MarkItUp!*

Figure 6.1 gives the system architecture of *MarkItUp!*. It shows the main components of *MarkItUp!*: the *user interface* component which accepts the users' operations and feeds back the system results; the *scanner* which scans the marked-up examples; the *learning* component which abstracts contents at the content level and structures at the structure level; the *DREAM grammar generator* which translates the abstracted grammar into DREAM DSD (document-structure description); and the *DREAM parser* which parses examples with the DREAM DSD.

The *list of concepts* is used to contents abstraction. The *grammar* is applied for both contents abstract and structures unification and abstraction. The *document collection* contains many subdocuments with similar structures.

All components except the parser are implemented in Smalltalk, the DREAM parser is implemented in C++.

6.2 User Interface

MarkItUp! supports two views to keep contact with the users: The *structure editor* and the *concept editor*. The structure editor is available for text-editing and structure-editing. It can support untagged or tagged documents, and the tagging process. The concept editor accepts a set of concepts, creates and displays a corresponding concept base as a DAG. Both editors support several functions (see Sections 6.2.1 and 6.2.2) via pull-down menus.

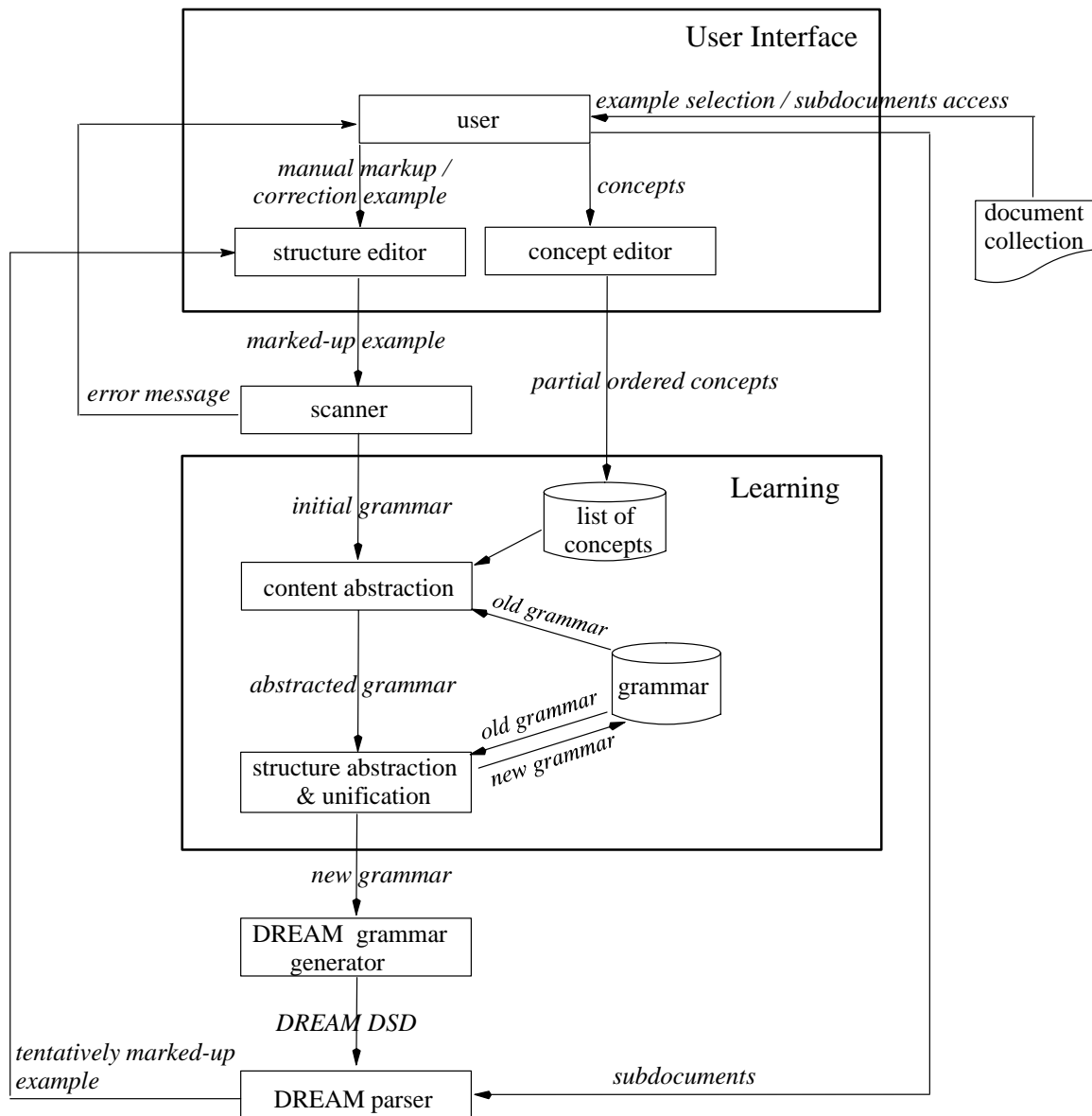


Fig. 6.1 System architecture of *MarkItUp!*

6.2.1 Structure Editor

Section 3.2.2 has briefly introduced the structure editor. This section will describe the individual components of the structure editor and their functions of the components.

The structure editor supports a graphical interface consisting of three views for distinct functions used to capture a grammar description from the user. The views are: *Structure View*, *Editor View*, and *Graphic View*. They are identified in Figure 6.2 and will be discussed in their respective subsections, in no particular order. Figure 6.3 shows the structure editor in use.

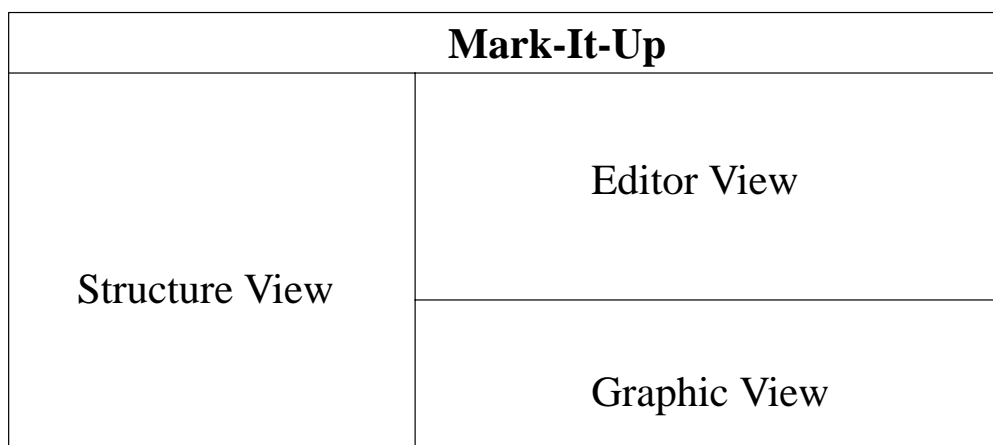


Fig. 6.2 Organization of the structure editor

Structure View

The Structure View is shown on the left side of the structure editor. It displays a list of *tag* names which appear in the Editor View as markups. The tag list can be changed when the user manually tags an example in the Editor View or uses the menu in the Structure View. More details of the former case are described in the Editor View in the part of *special-editing-operations*. Here the latter case is discussed which uses the menu in the Structure View to modify the tag list. When the cursor is moved into the Structure View, and the right button of the mouse is clicked, the user is offered the following options via the menu in the Structure View:

- Option *add*: It allows the user to add a new tag name into the Structure View. After the new tag is accepted by the system, the system automatically sorts the tags names according to alphabetic ordering.
- Option *delete*: It permits the user to delete a tag name from the Structure View.
- Option *rename*: It allows the user to change a tag name in the Structure View. But the user should note that the corresponding markup's name in the Editor View cannot be changed automatically. Therefore, it has to be changed manually. Otherwise, an inconsistency will raise an error message when the changed tag name is highlighted in the Structure View or when the content of the old tag name in the Editor View is manipulated.
- Option *save*: It permits the user to save the tags names to a file that can be loaded at a later time. During the execution of this option, the system will ask the user for the output file name.

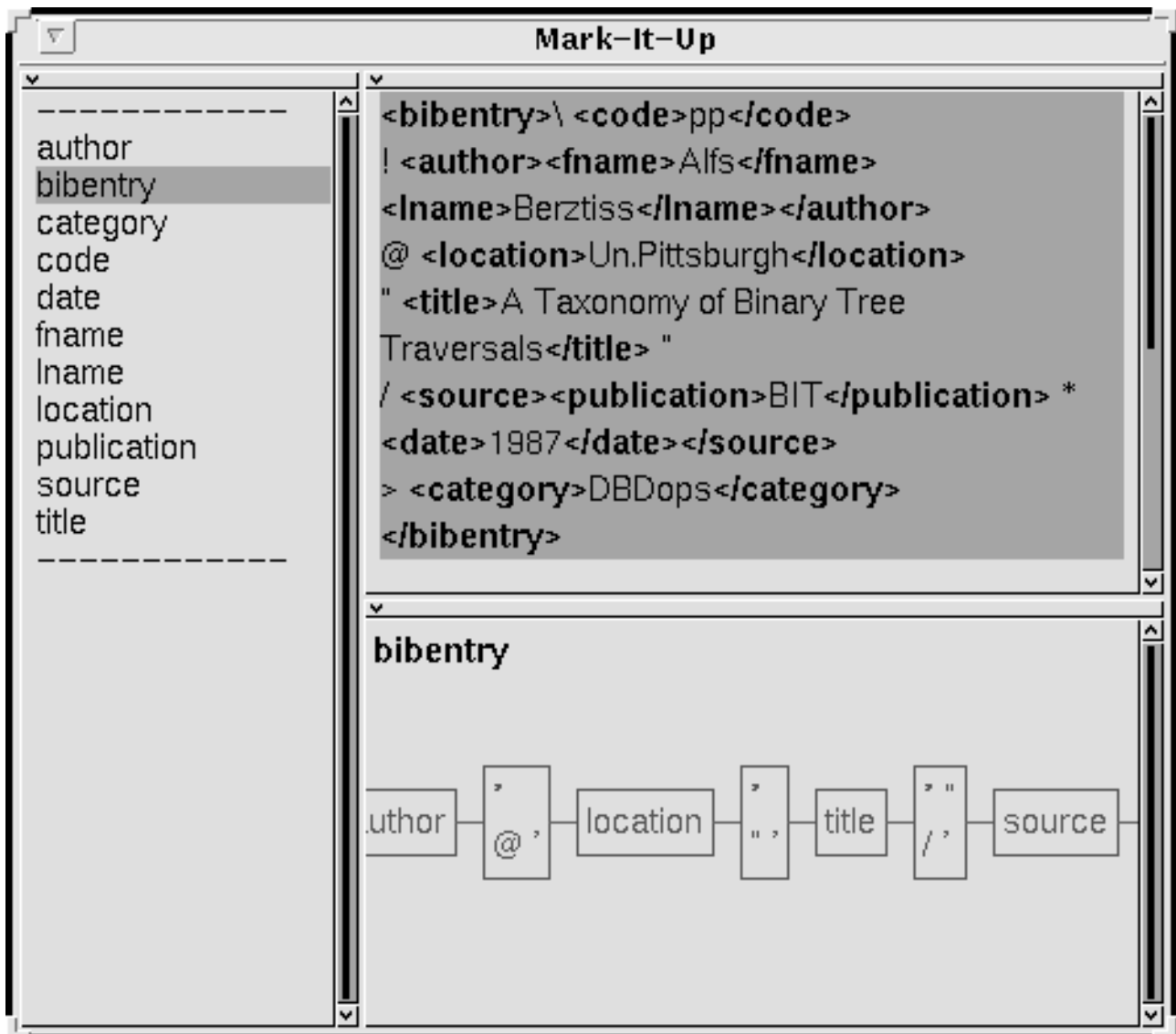


Fig. 6.3 An example of structure editing

- Option *load*: It allows the user to load the previously created tags names into the Structure View. Similarly to the option *save*, there is a dialog between the system and the user for an input file name.

When the user wants to modify one of the tags in the Structure View, the cursor must be positioned in the Structure View at first. Then the user selects the desired function from the menu with the right button of the mouse. For example, if the user wants to delete the tag name *author* in Figure 6.3, the user will do the following two steps:

- Highlight *author* in the Structure View with the mouse;

- Select the option *delete* in the menu.

After the two steps have been done, the system will execute the corresponding function and *delete* the tag name. When the function is performed, the name `author` will disappear from the tag list.

Besides modifying the tags, one can also see the RHS of a tag in the Graphic View, when the tag is highlighted with the cursor in the Structure View (the detailed explanation and an example of the such a function will be depicted in the description of the Graphic View below).

Editor View

The Editor View is a text view for inputting and editing documents. The cursor can be positioned anywhere within the Editor View by using the mouse. Once a text or part of it has been highlighted with the cursor, options such as *cut/copy* and *paste* can be performed on the highlighted text. These options can be called from an editing-menu. Besides them, the editing-menu represents other options that can be grouped as *compile-operations*, *special-editing-operations*, *normal-editing-operations*, and *I/O-operations*. The four groups contain a set of important options, therefore, they need to be further explained in the following.

The compile-operations include *four* options that can be performed for compiling a particular marked-up example. These options operate on three different marked-up examples:

- Option *Compiling the first marked-up example*.
When the system starts, there is no marked-up example and no grammar. Usually the user marks up an example and then uses the compiling option to utilize the marked-up example in the markup cycle (Figure 3.2) without grammar unification.
- Option *Compiling non-marked-up examples*.
After the first example is compiled, the system is able to mark up another example based on the existing grammar. This option is selected when the system gets a new example and there is an old grammar in the system.
- Option *Compiling the user updated element*.
When a tentatively marked-up example does not satisfy the user, it has to be corrected. When the option is selected, the corrected results (new structures or new strings) will be learned by the system.
- Option *Compiling the user updated example*.
The option is similar with the above option. The difference between them is that

this option is used to implement the exhaustive learning strategy, whereas, the above option is used to implement the partial learning strategy. For details on the two learning strategies see Section 6.6.2.

The special-editing-operation contains *one* option *markUpAString*. When a string is highlighted in the Editor View and this option is selected, the system will:

- insert start- and end-tags into the highlighted string.

This changes the focus of attention to the definitions of tags. When tagging a string with such a option, two cases will occur: the given tag name does not exist or it exists already.

If a tag does not exist, a new tag name is created. The new tag is added into the Structure View and the marked string is surrounded by the new start- and end-tags in the Editor View. For example, if there is an untagged string "pp" in the Editor View and the user wants to tag it with the name `code` which does not exist in the Structure View, the result of executing *markUpAString* is (1) the new tag name `code` is created in the Structure View and (2) the string "pp" is marked up by `<code>pp</code>`.

If the tag already exists, such as the tag `code` in Figure 6.3, after the above option is executed, the system will ask the user whether to insert the tag into the marked string to validate the regularity of the grammar. If there is no conflict, the start- and end-tags are inserted into the marked string. Otherwise, the user will define a new tag name for the marked string or cancel the call.

When a string occurs between two start- (or end-) tags or between different start- and end-tags, it is recognized as a cut-string.

The normal-editing-operations include *seven* Smalltalk system options (*again*, *undo*, *cut*, *copy*, *paste*, *accept* and *cancel*) in the menu, where the option *accept* means that, when a block of data is accepted by Smalltalk, the data has been stored in the Smalltalk system environment.


The last group in the editing-menu is I/O-operations. It includes *two* options that work on document files. One is to load a document from a file (*getTextFromFile*). Another one is to save the marked-up document as a file (*saveTextInFile*). The file can be read or stored into a directory which the user has identified.

Therefore, the input to the Editor View can come from two sources: either an entire file as discussed above, or by keystrokes from the keyboard when a user is entering text. If one part


of the text is loaded from a file, the user can expand the text with the keyboard. When the input of the expanded text is finished the user can select the *accept* option to store it in the Smalltalk environment. Normally, the user doesn't know where the data is stored. It is managed by the Smalltalk system.

Graphic View

The area of the Graphic View is located on the right-bottom of the structure editor (Figure 6.2) and is used to display the structure of a grammar rule. When the user highlights a tag in the Structure View, the corresponding rule is displayed on the Graphic View. For instance in Figure 6.3, the tag name `bibentry` is highlighted in the Structure View, the RHS of it shows in the Graphic View, where the bold string `bibentry` expresses a nonterminal of the grammar.

The graph describes the RHS of the tag `bibentry` which contains cut-strings, such as  and nonterminals, such as, `author`, `location`, etc., in the grammar. If the structure of a highlighted tag does not exist, the area of the Graphic View is empty.

The Graphic View has its own menu with which one can get more information about the rule. The menu contains *two* options that can be performed on a selected terminal or nonterminal string. A string is selected (highlighted) by using the mouse. Once a string is selected, one of the following options in the menu on the Graphic View can be chosen and the corresponding function will be executed.

- Option *select*: The corresponding function of the option is a movement command. It changes the focus of attention to the definition of the selected string. If the selected string is a terminal, there is no highlighted tag in the Structure View, and the content in the Graphic View does not change. If the selected string is a nonterminal, the highlighted tag in the Structure View is changed into the selected string, and the corresponding RHS of the selected string is shown on the Graphic View instead of the old one. For instance, in the Graphic View of Figure 6.3, if strings  is selected, no tag name is highlighted in the Structure View, while the content in the Graphic View is not changed; whereas, if the string `author` is selected, in the Structure View the tag name `author` is highlighted and in the Graphic View, the RHS of `author` is displayed.
- Option *inspect*: It shows the class type of the selected string. Every string in the *MarkItUp!* grammar belongs to one of class types which are defined by the system.

6.2.2 Concept Editor

In the concept editor the screen is divided into two areas by a vertical line: *Concept Set View* is on the left of the vertical line and *Concept Base View* is on the right of the vertical line. Figure 6.4 shows views and options in the menus of the concept editor.

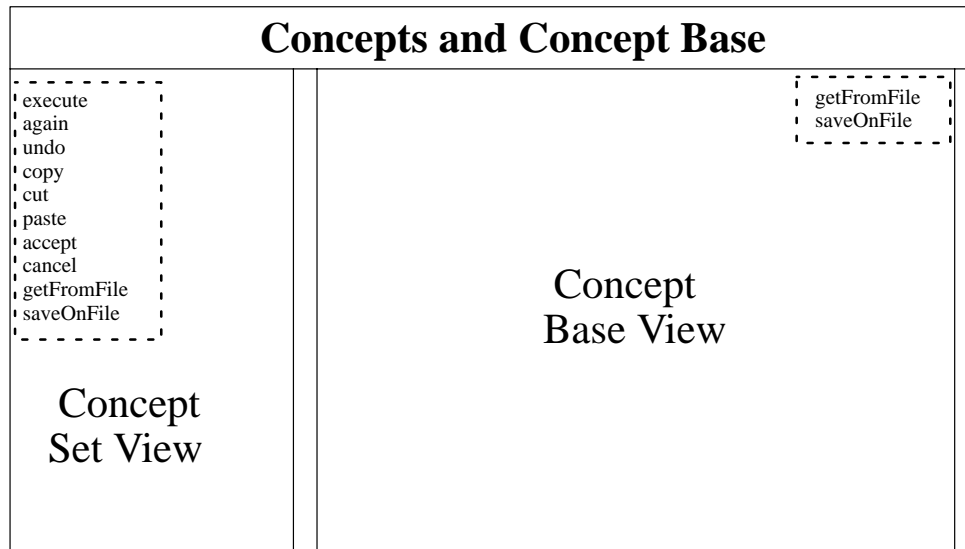


Fig. 6.4 Views and menus of concept editor

The Concept Set View is responsible for accepting a set of concepts from an external file or direct type-in from the keyboard. Once the concepts are entered, the user can select an *execute* option in the menu on the Concept Set View to construct a corresponding concept base. The graphic display of the concept base is shown in the Concept Base View immediately. The concepts or concept base can be stored as a file using the *saveOnFile* option in the menu.

Figure 6.5 shows the prototype of the concept editor in use. There are six concepts: $[a-zA-Z]^+$, $[a-z]^+$, $[A-Z]^+$, $[b-z]^+$, $[f-n]^+$, and $[a-p]^+$ in the Concept Set View. The Concept Base View shows the corresponding concept base of the six concepts (in Smalltalk a string is enclosed by single right quotations `'`). The symbol star `*` means any string which is not described in the input concepts and added automatically by the system as a default value. Among the concepts in the Concept Base View, there exists a kind of *descendant* relation which is a very important factor to directly identify whether concepts are comparable or incomparable according to the DAG in the Concept Base View. For each concept there is one or more symbol points `.` in front of it. The number of points identifies the descendant relation described by the following definition in visible as indentations.

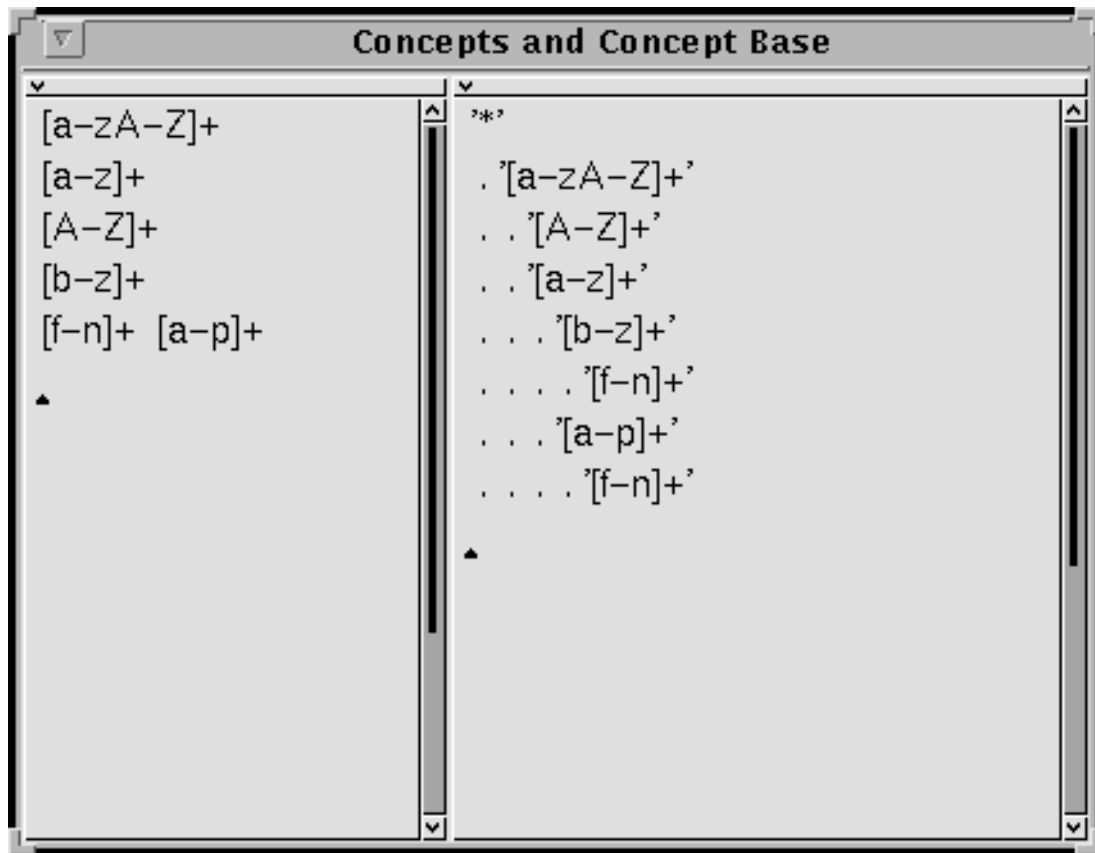


Fig. 6.5 Prototype of a concept editor

Definition 6.0 (*Descendant concept*) A concept A is a *descendant* of another concept B in the Concept Base View, if the number of points in front of A is larger than the number of points in front of B and from A to B there is no other concept C , where the number of points in front of B and C are equal. ■

For instance, in Figure 6.5, the concept $[b-z]^+$ is a descendant of the concept $[a-z]^+$, but it is not a descendant of the concept $[A-Z]^+$ since from $[b-z]^+$ to $[A-Z]^+$ there is another concept $[a-z]^+$, and $[a-z]^+$ and $[A-Z]^+$ have the same number of points in front of them.

Using the number of the symbol points and the descendant relation one can decide about the comparability of the concepts in the Concept Base View by the following rules. Suppose concept A , concept B , and concept C be different (not equivalent) concepts in a Concept Base View, then the rules are:

- If the concept A and the concept B have the same number of points in front of them in the Concept Base View or they have different numbers of points in front of them but there is no descendant relation between them, they are *incomparable*. For

instance, in Figure 6.5, the concept $[a-z]^+$ and $[A-Z]^+$ are incomparable since they have the same number of points in front of them; the concept $[A-Z]^+$ and $[b-z]^+$ are also incomparable since there is no descendant relation between $[A-Z]^+$ and $[b-z]^+$.

- If A is a descendant of B , then A and B are *comparable* and $A < B$, for instance the concept $[f-n]^+$ and $[a-zA-Z]^+$ in Figure 6.5.
- If the concept A_1, A_2, \dots, A_n are incomparable and $C < A_1, C < A_2, \dots, C < A_n$, the concept C will appear n times in the Concept Base View. For example, in Figure 6.5 the concept $[b-z]^+$ and $[a-p]^+$ are incomparable, but they are comparable with the concept $[f-n]^+$, therefore, the concept $[f-n]^+$ appears *twice*. Note that this rule applies only for displaying the concept C in the Concept Base View. In the system DAG, the concept C (e.g. $[f-n]^+$) shows as in Figure 6.6 and appears only once.

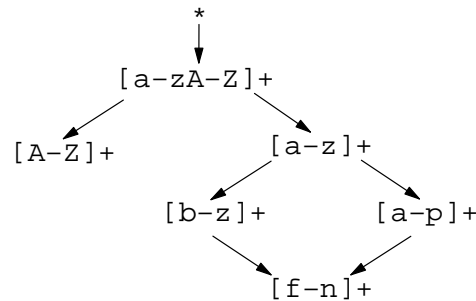


Fig. 6.6 A graphic description of the concept base in Fig. 6.5

Until now the discussion and description concentrate on how the concept base is shown at the user interface. Algorithm 6.1 will discuss how the concept base is created in the system.

Algorithm 6.1 Creating a concept base

Input: A set of concepts which the user defines

Output: A concept base as a DAG (described in Section 2.2.2) in which the concepts are ordered according to the binary relation (described in Section 2.1.3), that is, $\forall(c_i, c_j) \in DAG, c_j < c_i$

Method:

Routine *create_Concept-base*

```

1  begin
2       $N := \{\text{input concepts}\};$                                 //a set of concepts defined by the user.
3       $dag := \emptyset;$      $sub-dag-set := \emptyset;$ 
4      While  $N \neq \emptyset$  do
5          begin
6              select a node from  $N$ ;
7               $insert\_node(dag, node);$ 
8               $N := N - \{node\};$ 
9          end;
10 end

Function  $insert\_node(dag, node)$ 
1 begin
2      $norelate := \text{true};$ 
3     if ( $dag = \emptyset$ ) then // there is no DAG. The case occurs at the beginning of the algorithm.
4         let any string '*' be a base node of  $dag$ 
5     else for all direct descendants of the base in the current  $dag$  do
                                                //find a suitable position for the node in  $dag$ .
6         begin
7             case  $compare(descendant, node)$  of:
8                 '>':           //  $node < descendant$ , that is, node is a descendant of descendant.
9                     if (descendant is a DAG) then  $insert\_node(descendant, node)$ 
10                    else add node into  $dag$  as a direct descendant of descendant;
11                     $norelate := \text{false};$ 
12                    break;
13                 '<':           //  $descendant < node$ , that is, node is an ancestor of descendant.
14                    add descendant into  $sub-dag-set$ ;    //node may have more than one descendants.
15                     $norelate := \text{false};$ 
16                    break;
17                 end;
18                 if  $norelate$  then                                // node and descendants are incomparable.
19                     add node into  $dag$  as a direct descendant of the current base;
20                 else if  $sub-dag-set \neq \emptyset$  then
21                     begin                                //node is an ancestor of the node(s) in  $sub-dag-set$ .
22                         create a sub-dag in which node is its base node;
23                         insert the whole elements in  $sub-dag-set$  as direct descendants of the sub-dag's base;
24                         add the sub-dag into  $dag$  as a direct descendant of the  $dag$ 's base;
25                          $sub-dag-set := \emptyset;$ 
26                     end;
27                 return ( $dag$ );
28 end    // It is of no significance if two equivalent concepts are in the concept base, therefore,
          // if two concepts are equivalent, only one concept is kept in the  $dag$ .

Function  $compare(a, b)$ 
    //  $a, b$  are regular expressions.

```

```

1  begin
2      if  $a \equiv b$  then return ('=')           //  $a$  is equivalent to  $b$ .
3      else if  $a < b$  then return ('<')       //  $a$  is more-specific-than  $b$ .
4          else if  $b < a$  then return ('>')   //  $b$  is more-specific-than  $a$ .
5          else return ('n');
6  end

```

With this algorithm, the concepts example in Figure 4.2 is depicted in the concept editor as Figure 6.7, where the string `[\]+` is a system representation of an arbitrary number of blanks.

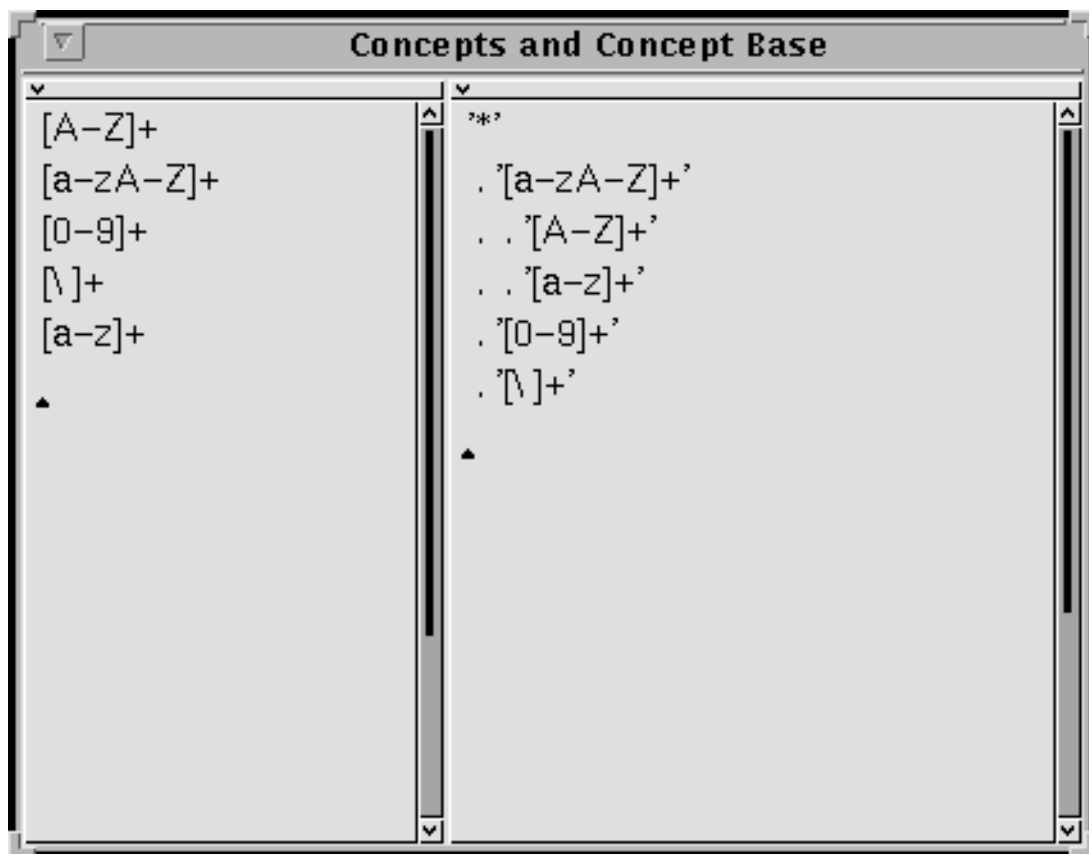


Fig. 6.7 A prototype of concept editor for the concepts in Fig. 4.2

For different type classes, the user can easily create the corresponding concept base. For simplicity, *MarkItUp!* does not choose a strategy that allows to add new concepts dynamically without changing the whole existing concept base. For example, if one wants to add a new concept `[Fr]om` in the concept base shown in Figure 6.7, the system will rebuild the concept base and generate a new concept base rather than searching the concept base and finding a suitable place for the new concept.

6.3 From Marked-up Example to a Grammar

MarkItUp! translates a marked-up example into a grammar by using a *scanner*. The scanner attempts to scan the marked-up example and to generate a corresponding grammar describing the document structure. If the scanned portion is incorrect, that is, the tags are not matched, then the user is asked to supply additional information. For instance, in a marked-up example there is only a start-tag `<author>` but no end-tag `</author>`, in such case, the user has to correct the marked-up example. Otherwise, the scanner identifies start-end tags to create a marked-up tree since a tree is an appropriate data structure for storing both the structure and the text of the document. The document structure can be viewed as the document (*root node*) which is subdivided into components (*interior nodes*), and which can be further subdivided until the indivisible components (*leaf nodes*) are reached. Such a representation is convenient for verifying that a document obeys the syntax rules of the language.

When the scanner processes a marked-up document, at first, a tree called MUPTree is created in the *MarkItUp!* system. At the end, the tree is converged into a set of grammar rules which have the syntactical form described in Section 4.3.2, that is, the RHSs of the grammar rules are regular sets. For each grammar there is a root rule which is called a *start rule* in the implementation system.

An example transformation from a marked-up example to a MUGrammar is shown by Figure 3.7 in Section 3.2.2 and Grammar-Sample 3.1 in Sections 3.3.

6.4 Implementation of Learning Component

The learning component consists of two parts: content abstraction and structure abstraction & unification. The content abstraction learns abstracted concepts from the concept sequence. The structure abstraction & unification solves the problem of generalization grammars by making the grammars more general using more positive examples.

In Section 3.3, the two types of grammar rules have been distinguished on the basis of the RHS of the rules: structure-rule (Definition 3.1) and string-rule (Definition 3.2). When discussing the content abstraction, the focus is on the RHS of the string-rules, whereas, when using learning rules, the focus is only on the RHS of the structure-rules. Therefore, when a grammar is used as input into the learning component, the string-rules and structure-rules are separated. The string-rules are abstracted by algorithms 4.1 and 4.2 in Section 4.2.5.1; the structure-rules are unified and abstracted by the rules in Section 4.3.3.

The implementations of the rules have two features: (1) the parameters of them are formed by the expressions in the RHS of a grammar rule; (2) the implementations are *isolated* for each rule, that is, if the left hand side of rule *A* is not equal to the left hand side of rule *B* in a grammar, the procedure to unify and abstract the RHSs of the rule *A* does not influence the learning process of the rule *B*.

When the RHS of a structure-rule consists of nonterminals and terminals (cut-strings), the nonterminals and terminals can be separated into several groups. The different grouping strategies for the nonterminals and terminals will lead to different results. To get a correct consequence, what remains to be done for the structure-rules is to determine the grouping of cut-strings and nonterminals. This detail strategy is discussed in the following subsection.

6.4.1 Grouping Cut-Strings with Nonterminals

Cut-strings play a role as delimiters in the structure-rules. Usually, certain cut-strings accompany certain nonterminals, in other words, if one nonterminal does not occur in the rule, the cut-string(s) caused by it will also not occur. The relation among nonterminal and cut-strings are called a *cause-effect relation*. However, the problem is that the relation is not obvious in a structure-rule. For example, two bibentry rules look like:

Rule-Sample 6.1

```
bibentry -> "^\\□" code "\\n^!□" author "\\n^@□" location "\\n^\"□"
           title "□\"\\n^/□" source "\\n^>□" category "\\n"
```

Rule-Sample 6.2

```
bibentry -> "^\\□" code "\\n^!□" author "\\n^\"□" title "□\"\\n^/□"
           source "\\n^>□" category "\\n"
```

where cut-strings and nonterminals are sequentially organized. From the rule, it is not easy to see the cause-effect relation among them. However, if the relations in structure-rules are not correctly identified, when unifying and/or abstracting the rules, a wrong result will be created. For instance, if the RHSs of Rule-Samples 6.1 and 6.2 are unified without being based on the relations, one of the results will be:

Rule-Sample 6.3

```
bibentry -> "^\\□" code "\\n^!□" author "\\n^@□" | "\\n^\"□" location?
           "\\n^\"□"? title "□\"\\n^/□" source "\\n^>□" category "\\n"
```


This is not a correct rule of `bibentry` since the element `location` cannot start with the string “\n^\”□” in documents. Therefore, the proper identification of which cut-strings are caused by which nonterminals becomes an important problem.

One simple solution is to compare additional different examples to detect the relations. For instance, in Rule-Sample 6.2 the nonterminal `location` is missing, at the same time, the cut-string “\n^\”□” has not occurred in the rule `bibentry`. Therefore, one knows that the cut-string “\n^\”□” is caused by the nonterminal `location`. However, this is not a practical way since one cannot determine the cause-effect relations in a structure-rule until one finds proper examples.

The other solution is using an algorithm to group cut-strings and nonterminals in a structure-rule (Algorithm 6.2). The basic strategy of the algorithm is that the first element of the rule determines the combining form of cut-strings and nonterminals. If the first element of the rule is a cut-string, the form is one “*cut-string* followed by *nonterminal*” (denoted as *cut-string* + *nonterminal*) or “*cut-string* followed by *nonterminal* followed by *cut-string*” (denoted as *cut-string* + *nonterminal* + *cut-string*); otherwise, the form is “*nonterminal* followed by *cut-string*” (denoted as *nonterminal* + *cut-string*) or a single “*nonterminal*”. Each form expresses a syntactic cause-effect relation and is called a *cut-copy group*.

Algorithm 6.2 Grouping cut-strings with nonterminals in a structure-rule

Input: A structure-rule R containing cut-strings and a global variable $aGroup$

Output: A set of groups to record the combination of cut-strings and nonterminals

Method:

```

Function grouping( $R$ ,  $aGroup$ )
1  begin
2    if the first element on the RHS is a cut-string then  $idx_2 := 0$ 
3    else  $idx_2 := 1$ ;
4    while  $idx_2 < \text{the size of the rule}$  do
5      begin
6         $pos := idx_2$ ;
7         $idx_1 := \text{the position of the first cut-string searching from } pos$ ;
8         $idx_2 := \text{the position of the first cut-string searching from } idx_1$ ;
9        if  $idx_2 = 0$  then  $idx_2 := \text{the size of the rule}$ ;
10       if  $idx_2 \neq \text{the size of the rule}$  then  $idx_2 := idx_2 - 1$ ;
11       get a group which contains the elements from the positions  $idx_1$  to  $idx_2$ ;

```

```

12      add the group to the tail of aGroup with the same name of the nonterminal in the
group;
13      end;
14      return (aGroup);
15  end

```

Applying the above algorithm, the groups of Rule-Samples 6.1 and 6.2 are:

Cut-Group-Sample 6.1

```

("^\\□" code); ("\\n^!□" author); ("\\n^@□" location);
("\\n^"□" title); ("□"\\n^/□" source); ("\\n^>□" category "\\n");

```

Cut-Group-Sample 6.2

```

("^\\□" code); ("\\n^!□" author);
("\\n^"□" title); ("□"\\n^/□" source); ("\\n^>□" category "\\n");

```

On the basis of Cut-Group-Samples 6.1 and 6.2, when the RHSs of Rule-Samples 6.1 and 6.2 are unified, a single correct result shown as Rule-Sample 6.4 is easily inferred:

Rule-Sample 6.4

```

bibentry -> "^\\□" code "\\n^!□" author ("\\n^@□" location)?
           "\\n^"□" title "□"\\n^/□" source "\\n^>□" category "\\n"

```

where cut-string "\\n^@□" is regarded as a part of the nonterminal location.

This strategy is simple and general. But for some special case, the grouping is not exactly correct. For instance, consider the subsequence title "□"\\n^/□" source of Rule-Sample 6.1, if using the above strategy, the string "□"\\n^/□" is grouped with source. However, the substring "□"\\n^/" is caused by title rather than source. Normally, the grouping will not lead to an error. Only for the case that title or source is missing in another example, the grouping will result at an incorrect solution. Of course, this is not serious. To overcome the worst case, an alternative strategy to group cut-strings and nonterminals is considered. Each line of documents is supposed to have complete semantics such that one can regard the character *return* as a specific delimiter. Under the hypothesis, it is assumed that a group is ended by the return character. According to this strategy, the groups of Rule-Sample 6.1 are:

Cut-Group-Sample 6.3

```
( "^\\□" code "\\n"); (^!□" author "\\n");
(^@□" location "\\n"); (^"□" title "□\\n");
(^/□" source "\\n"); (^>□" category "\\n");
```

The semantic meaning of the groups are clearer than in Cut-Group-Sample 6.1. However this strategy lacks generality as it assumes special semantics. The system therefore still adopts the first strategy to group cut-strings and nonterminals.

6.5 DREAM Grammar Generator

The process of the document-structure recognition and tagging is realized by the DREAM parser. Therefore, the inferred grammar from the learning component has to be translated into a DREAM DSD. If the DSD is correct, the DREAM parser will tag the document correctly. Otherwise, DREAM cannot get a correct result. The task of translating the grammar rules into DREAM DSD is done by the DREAM grammar generator.

From the abstracted grammar, a DREAM DSD is generated in two steps: First, the SGML structure is built (without cut-strings); Second, the cut-copy groups at the content-level are used to form the expressions at that level.

To parse subsequent examples with slightly deviating structures and contents, every ELEMENT name in the DSD has been associated with a fallback rule `anything`, which parses all those parts which cannot be parsed by one of the available ELEMENT definitions and is stopped when the next element can be correctly recognized. Such portions, which are surrounded by `<anything>` and `</anything>`, then can be easily identified and further disambiguated by the user.

Figure 6.8 shows a complete DREAM DSD generated from the abstracted grammar of the manually marked-up example in Figure 6.3. The concepts for the examples are: `[a-zA-Z]`, `"."` and `"□"`. The original document of the marked-up example is:

Document-Sample 6.1 The original document of the marked-up example in Figure 6.3

```
\\□pp
!□Alfs□Berztiss\\n
@□Un.Pittsburgh\\n
"□A Taxonomy of Binary Tree Traversals□"\\n
/□BIT□*□1987\\n
>□DBDops\\n
```

```

<!DOCTYPE bibdoc [
<!ELEMENT bibentry - - code,
                        author,
                        location,
                        title,
                        source,
                        category >

<!ELEMENT code - - (anything#, cut("^\\□"), copy([a-zA-Z]+))?>

<!ELEMENT author - - (anything#, cut("$^"!□"), fname, lname)?>

<!ELEMENT fname - - (anything#, copy([a-zA-Z]+))?>

<!ELEMENT lname - - (anything#, cut("□"), copy([a-zA-Z]+))?>

<!ELEMENT location - - (anything#, cut("$^"@□"),
                        copy([a-zA-Z]+ "." [a-zA-Z]+))?>

<!ELEMENT title - - (anything#, cut("$^\"□"),
                        copy(( [a-zA-Z]+ "□" ) + [a-zA-Z]+))?>

<!ELEMENT source - - (anything#, cut("□\\\" \"$^\"/□"),
                        publication, date)? >

<!ELEMENT publication - - (anything#, copy([a-zA-Z]+))? >

<!ELEMENT date - - (anything#, cut("□*□"), copy([0-9]+))? >

<!ELEMENT category - - (anything#, cut("$^\">□"),
                        copy([a-zA-Z]+), cut("$^"))?>

<!ELEMENT anything - - copy(.#)>

]>

```

Fig. 6.8 DREAM DSD of the example in Figure 6.3

Based on the DSD annotated with recognition patterns, the DREAM parser checks for syntax errors in the DSD. If there is no error message, DREAM produces a tagged document which conforms to a document type definition (DTD) in SGML and can be further processed with any SGML-based tools. For instance, when Document-Sample 6.1 is tagged by the DREAM parser with the DSD (shown in Figure 6.8), the tagged document is shown in Figure 6.9.

```

<!DOCTYPE bibdoc>
<bibentry><code>pp</code>
<author><fname>Alfs</fname>
<lname>Berztiss</lname>
</author>
<location>Un.Pittsburgh</location>
<title>A Taxonomy of Binary Tree Traversals</title>
<source><publication>BIT</publication>
<date>1987</date>
</source>
<category>DBDops</category>
</bibentry>

```

Fig. 6.9 The tagged document of Document-Sample 6.1 is created by the DREAM DSD in Figure 6.8

It is an SGML-document generated by a DSD not a learning example, therefore, it does not look like its manually marked-up example (in Figure 6.3) which maintains cut-strings in the document.

6.6 Learning Strategies

The implementation of each component in the system architecture (Figure 6.1) has been sequentially discussed. This section will give several examples to show how the system is used and what kind of learning strategy can be chosen.

6.6.1 Fallback Rule

When the DSD in Figure 6.8 is applied to a new example (Document-Sample 6.2), the effect of the fallback rule anything is shown in Figure 6.10.

Document-Sample 6.2 A new example

```

\␣pr␣n
!␣P.␣Buneman␣n
!␣M.␣Atkinson␣n
"␣Inheritance and Persistence in Database Programming Lan-
guages␣"␣n
/␣ACM SIGMOD␣*␣1986␣n
>␣DBDquery␣n

```

In the new example there are two authors (lines that start with the string " !␣"), the first name `fname` with the abbreviation character point ' . ' which is a new character not oc-

curing in the earlier example, the element `location` (line that starts with the string `"@□"`) is missed, and the format of element `publication` is new.

```
<!DOCTYPE bibdoc>
<bibentry><code>pr</code>
<author><fname>P</fname>
<lname><anything>.</anything>
Buneman</lname>
</author>
<location></location>
<title><anything>
!□M.□Atkinson</anything>
Inheritance and Persistence in Database Programming Lan-
guages</title>
<source><publication>ACM</publication>
<date><anything>□SIGMOD</anything>
1986</date>
</source>
<category>DBDquery</category>
</bibentry>
```

Fig. 6.10 A tentatively marked-up example generated by the DREAM DSD
in Figure 6.8

Therefore, in Figure 6.10 the character point and the second author name are marked up with `anything`, the `location` matches nothing, the second part of the `publication` is marked up with `anything`. However, the `code` (the string `"pr"`), the partial `fname` of the first author (the character `'P'`), the `lname` of the first author (the string `"Buneman"`), the `title` (the string `"Inheritance and Persistence in Database Programming Languages"`), the `source` (the structure `publication`, `date`), the partial `publication` (the string `"ACM"`), the `date` (the string `"1986"`), and the `category` (the string `"DBDquery"`) are properly marked up.

The user corrects the result by means of the *structure editor* where the corrected result is then shown. The corrected example is passed to the above components step-by-step, finally a new grammar is inferred. The corrected example and new grammar will be described in the following subsections.

6.6.2 Exhaustive vs. Partial Learning

When an initial grammar is generated, there exist two alternative strategies to learn another example: learning from an entire example and learning from a partial example. To explain them, let us consider the above examples again.

In Figure 6.10 the content of the element `location` is empty, the content of elements `author` and `publication` are recognized only partial, and the unrecognized parts of the content are included by the fallback rule `anything`. Therefore, the user will correct them. The form of the corrected example depends on the above two learning strategies. Figures 6.11 and 6.12 show the user-corrected example using the two strategies, respectively:

```
<!DOCTYPE bibdoc>
<bibentry><code>pr</code>
<author><fname>P.</fname>␣<lname>Buneman</lname></author>
<author><fname>M.</fname>␣<lname>Atkinson</lname></author>
<title>Inheritance and Persistence in Database Programming Lan-
guages</title>
<source><publication>ACM SIGMOD</publication>
<date>1986</date></source>
<category>DBDquery</category>
</bibentry>
```

Fig. 6.11 The user-corrected example using the exhaustive learning strategy

```
<!DOCTYPE bibdoc>
<bibentry><code>pr</code>
<author><fname>P.</fname>␣<lname>Buneman</lname></author>
<author><fname>M.</fname>␣<lname>Atkinson</lname></author>
<location></location>
<title>Inheritance and Persistence in Database Programming Lan-
guages</title>
<source><publication>ACM SIGMOD</publication>
<date>1986</date></source>
<category>DBDquery</category>
</bibentry>
```

Fig. 6.12 The user-corrected example using the partial learning strategy

The difference between the two modifications is that there is no element `location` in the corrected example using the first strategy (Figure 6.11), whereas it is kept in the corrected example using the second strategy (Figure 6.12). The reason will be explained in the following two sections.

6.6.2.1 Learning from an Entire Example

The basic idea of this strategy is to generate a new complete grammar and to learn the whole grammar rules at the content-level and at the structure-level. Since the grammar does not allow the RHS of a rule to be empty (Section 4.3.2), the element `location` has to be deleted, otherwise the system cannot work. For this reason, the rule `location` is removed in Figure 6.11. With the exhaustive strategy, the unified grammar is shown in Figure 6.13, where the

```

<!DOCTYPE bibdoc [

<!ELEMENT bibentry - - code,
                    author+,
                    location?,
                    title,
                    source,
                    category>

<!ELEMENT code - - (anything#, cut(^"\|"), copy([a-zA-Z]+))?>

<!ELEMENT author - - (anything#, cut($^"!|"), fname, lname)?>

<!ELEMENT fname - - (anything#, copy([a-zA-Z]+ | [a-zA-Z]"."))?>

<!ELEMENT lname - - (anything#, cut("|"), copy([a-zA-Z]+))?>

<!ELEMENT location - - (anything#, cut($^"@|"),
                        copy([a-zA-Z]+ "."[a-zA-Z]+))?>

<!ELEMENT title - - (anything#, cut($^"\|"),
                    copy(( [a-zA-Z]+"|")+ [a-zA-Z]+))?>

<!ELEMENT source - - (anything#, cut("|\""$^"/|"),
                    publication, date)? >

<!ELEMENT publication - - (anything#,
                        copy([a-zA-Z]+ | [a-zA-Z]+"|"[a-zA-Z]+)))?>

<!ELEMENT date - - (anything#, cut("|*|"), copy([0-9]+))?>

<!ELEMENT category - - (anything#, cut($^">|"),
                    copy([a-zA-Z]+), cut($^))?>

<!ELEMENT anything - - copy(.#)>

]>

```

Fig. 6.13 Changed rules in DSD based on the exhaustive learning strategy

bold parts indicate the differences between the old rules which are in Figure 6.8 and the new inferred rules.

With the algorithm 4.2, one gets an abstracted-list A_{fname} of the rule `fname`, $A_{fname} = [a-zA-Z]^+, [a-zA-Z]"."$. Here it is translated as an alternative operator in DSD $([a-zA-Z]^+ \mid [a-zA-Z]"."$), otherwise, DSD cannot manipulate it.

6.6.2.2 Learning from a Partial Example

Generating a new grammar for a new example and learning the whole rules are redundant for the elements whose structure does not deviate from the old one and whose contents can be recognized by the existing abstracted strings. To avoid unnecessary processing, an alternative learning strategy has been implemented – learning from part of an example, that is, the system can compile one of the elements which is corrected by the user in the tentatively marked-up example.

To get a new complete grammar, the partial learning strategy will be repeated several times. Each repetition is called a *learning-unit*. How many learning-units are required in an example depends on how many elements are not correctly recognized by the old grammar or deviate from the old grammar, that is, a learning-unit corresponds to a changed element. For this reason the empty element `location` is kept in Figure 6.12. In Figure 6.10 there are merely three elements `author`, `location`, and `publication` which are new with respect to the old grammar, therefore, there are three learning-units. Note that if an element occurs more than once in the example, they all are treated as one learning-unit, that means the repeated elements are learned at the same time.

Applying the partial learning strategy for the example in Figure 6.12, the learning can be done in three steps, where each of the learning processes is isolated, in other words, the sequence of executing the steps does not influence the final learning result. For instance, whether `location`, `author` or `publication` is learned first, the final result is the same. Let us learn `location` first.

When the user highlights the empty element `<location></location>`, the changed rule related to Figure 6.8 is shown in Figure 6.14:

Since new rules for `code`, `author`, `fname`, `lname`, `location`, `title`, `source`, `publication`, `date`, and `category` are not changed, they are not rewritten here.

The new DSD indicates the element `location` as an optional element described in the element `bibentry`. When the user selected the corrected element `author` to learn, the changed rule related to Figure 6.8 and 6.14 is shown in Figure 6.15:

```

<!DOCTYPE bibdoc [
<!ELEMENT bibentry - - code,
                        author,
                        location?,
                        title,
                        source,
                        category>
.
.
.]>

```

Fig. 6.14 Changed rule in DSD for learning an optional element on the basis of the partial learning strategy

```

<!DOCTYPE bibdoc [
<!ELEMENT bibentry - - code,
                        author+,
                        location?,
                        title,
                        source,
                        category>
.
.
.
<!ELEMENT fname - - (anything#, copy(([a-zA-Z]+ | ([a-zA-Z]"."))))?>
.
.
.]>

```

Fig. 6.15 Changed rules in DSD for learning a *repetition element* on the basis of the partial learning strategy

For the same reason as above, new rules for code, author, lname, location, title, source, publication, date, and category are not reproduced here. In the same way the user can select publication to learn.

```

<!DOCTYPE bibdoc [
    .
    .
    .
    <!ELEMENT publication - - (anything#,
                                copy([a-zA-Z]+ | [a-zA-Z]+ " " [a-zA-Z]+) )?>
    .
    .
    .
]>

```

Fig. 6.16 Changed rules in DSD for learning a new string on the basis of the partial learning strategy

Combining the changed rules in Figures 6.15 and 6.16 with the unchanged rules in Figure 6.8, a new DSD is obtained which is equal to the DSD in Figure 6.13. Since there is no redundancy, the partial learning strategy is more efficient than the exhaustive learning strategy, especially, if a grammar consists of many elements, where only a few of them need to be corrected.

6.7 Experimental Evaluation of the System

This section describes some experiments with the system carried out to measure the effectiveness of *MarkItUp!*'s approach to grammar learning. The experimental setting is as follows:

As an example document collection a portion of the bibliography on databases compiled by Gio Wiederhold is taken (the most recent version is available from <ftp://db.stanford.edu/pub/siroker/biblio.txt>). The randomly chosen portion consists of 260 entries. Examples from this bibliography have been used throughout this thesis. Below one example is reproduced for better clarity:

Document-Sample 6.3 Example of a bibliographic entry

```
\l rp\n
!l M.L.l Brodie\n
!l D.l Ridjanovic\n
"l Functional Specification and Verification of Database Trans-
actionsl "\n
/l report Oct.1984\n
>l DBDmodel.0\n
>l DBDtrans.4\n
```

While at the top-level this is a fairly well behaved example, with elements like author, title, etc. clearly separated by unambiguous delimiter characters, the structuring levels below show quite a few irregularities which make the intellectual specification of a grammar non-trivial. For example, the inner structure of authors can contain both, abbreviated first names, as well as fully spelled out first names, and middle names may be missing. As another example, the inner structure of bibliographic source information is very heterogeneous across the individual bibliographic entries, with subelements like publishing date, editor information, volume number, etc. occurring rather arbitrarily.

For this example collection, three grammars have been incrementally trained with 1, 5, and 10 example bibliographic entries. Each grammar has been translated into a DREAM DSD, which has been applied to the entire document collection. The resulting SGML-marked-up documents have been evaluated along the following dimensions:

(1) the number of (correctly or incorrectly) recognized elements vs. the number of all elements including the elements that had to be accepted by the fallback rule `<anything>` (`copy(.#)`). This ratio indicates the overall performance of a learned grammar.

(2) the number of *anything* elements vs. the number of correct elements. This ratio indicates the recall of the learned grammar, and thereby gives a measure for the amount of additional human refinement needed in the mark up process.

(3) the number of incorrect elements vs. the number of correct elements. This ratio measures the precision of the learned grammar. Minimizing this ratio is the most important goal, because incorrectly tagged elements need the biggest effort to be *detected* and corrected.

To judge the influence of the predefined concept set used for copy-string abstraction on the learning process, two different concept sets have been used: $S_1 = \{[a-zA-Z]^+, [0-9], [.,\-\:], [*\sqcup], [\[\]\backslash(\)]\}$ and $S_2 = \{[a-zA-Z.,\-\:]^+, [0-9], [*\sqcup], [\[\]\backslash(\)]\}$. The sole difference between S_1 and S_2 is that the character sets $[a-zA-Z]$ and $[.,\-\:]$ are defined as separate concepts in S_1 , whereas they are combined to a single, more general concept in S_2 (see also sections 3.3 and 4.2.5.3 for examples of grammars (Grammar-Sample 3.2 and Grammar-Sample 4.1) that have been generated on the basis of different concept sets). Both concept sets have been applied to the same sequence of examples.

Table 6.1 gives the results of the concept set S_1 .

analysis assumption the results number of manually marked examples	the number of recognized elements vs. the number of all elements includ- ing fallbacks (%)	anything elements vs. correct elements (%)	incorrect elements vs. correct elements (%)
1	9.7	55	66
5	82	7.3	2.4
10	97	2.6	1.8

Table 6.1 The experiment results generated by applying concept set S_1

In all three columns we see a clear asymptotic improvement achieved by increasing the number of provided examples. Marking up only one example leads to very bad results. These are mainly due to a small bug in the version of DREAM used for the experiment, which for the highly ambiguous grammars generated by *MarkItUp!* accepted some document portions in a fallback rule even if there was a more specific rule applicable in this context (this bug has been corrected at the time of writing). Thus only about 10% of the entire document has been encapsulated by a tag (column 1), more than half of the elements have only been accepted by a fallback rule (column 2), and almost two thirds of the marked-up elements have been tagged

incorrectly (column 3). After five marked-up examples, however, the generated grammar is much less ambiguous, and anticipates many more structural deviations. Consequently, over 80% of the document has been accepted by a specific element rule, less than 10% have required manual refinement, and as little as 2.4% of the elements have been incorrectly marked up. Finally, after 10 examples a quite satisfactory performance has been reached, needing rather small amounts of human correction.

Table 6.2 shows the results achieved by applying the concept lattice S_2 .

analysis assumption the results number of manually marked examples	the number of recognized elements vs. the number of all elements includ- ing fallbacks (%)	anything elements vs. correct elements (%)	incorrect elements vs. correct elements (%)
1	39	45	40
5	86	6.9	2.3
10	97	2.8	1.9

Table 6.2 The experiment results generated by applying concept set S_2

Clearly, the initial performance with this more generic lattice is significantly better than for the concept lattice S_1 . The generic pattern $[a-zA-Z.,\backslash-:]$ accepts in particular abbreviated names, containing “.”, and titles, containing “–”, before they have actually occurred as names or titles in a marked-up example. However, the results after 10 examples are a little bit worse than for S_1 . In particular, the recall and the precision of applying the more generic lattice have both decreased a little bit. Here the more generic pattern $[a-zA-Z.,\backslash-:]$ fails to distinguish some subelements in the bibliographic source information, such as “Sep.” indicating a date from “ACM” indicating the conference of a bibliographic entry. This shows that as soon as the content of elements, as opposed to delimiter characters, becomes important for classifying elements, a more elaborate concept lattice leads to overall better results. On the other hand, the rather small differences in performance also demonstrate that for syntactically structured sources, *MarkItUp!* is fairly robust with respect to the concept lattice used, because it can better rely on its powerful inductive capabilities on the structural parts. As a consequence, concept-lattices can stay fairly generic and be applied to many application domains, as long as the structure to be recognized can rely on the syntactic context.

This result is inline with the evaluations performed in the context of the CLIP-ing project [42]. In this project, the system TATOE has been used to detect linguistic categories, such as proper nouns or temporal expressions, in a corpus consisting of German news messages in order to transform them to the SGML-based News Industry Text Format (NITF). Like DREAM, the parser underlying *MarkItUp!*, TATOE uses named regular expressions for structure specification, such as natural language phrases. TATOE extends these with access to linguistic components, which comprise a morphological analysis, and a lexicon indicating the semantic role of proper nouns. These components can be seen as an elaborate substitute for the concept lattices used by *MarkItUp!*. In the CLIP-ping domain, the detection of linguistic categories needs to rely much more on the linguistic resources than on syntactic structure. And indeed, the experiments of TATOE showed that the lexical resources required much more intellectual refinement than the set of (manually specified) syntactic rules. The comparison of these application domains also indicates an important line of further research in using machine learning for document recognition: Where the content of structural elements needs to contribute to disambiguation, a resource corresponding to the concept lattice may not be fixed a priori, but needs to be trainable too.

6.8 Summary

The implementation of the *MarkItUp!* learning system has been described. With a friendly user interface, the system builds an easy to use environment to implement incremental learning. The different learning strategies provide a more flexible method for considering possible revisions to the grammar rules. The evaluation shows that in particular for syntactic structuring tasks, the learned grammars achieve an acceptable performance after rather few manually marked-up examples.

Related Work

Conceptually, the learning approach adopted by this thesis draws mainly from two fields: Editing-By-Example, and more generally, inductive learning.

7.1 Editing-By-Example

Editing-By-Example was first introduced by R. Nix [39, 40]. It derives generic string transformation programs from a few editing operations on examples in a text editor. Two approaches can be distinguished: Function approaches, which regard only the input and the output of editing operations, and procedural approaches, which reason about the editing operations themselves.

7.1.1 Function Approaches

The goal of function approaches is to synthesize a transformation program on the basis of a pair of input/output examples in a text editor. An example of the function approach is the system EBE (Editing-By-Example) developed by R. Nix. The aim of EBE is to solve repetitive text editing problems.

In that system, the user specifies a set of input/output pairs exemplifying a text transformation, and the EBE system attempts to infer common patterns to the pairs and synthesizes a program to perform the transformation in general. The user may then execute this program to perform further transformations, or may give further examples to the system.

The program synthesized by EBE is called a *gap program* which consists of a *gap pattern* that matches a portion of the text and parses it into fields (constant or variable), and a *gap replacement* that copies, rearranges, or deletes the fields (and may introduce a new constant field). For instance, the user specifies the following two input/output pairs:

```
Braves 4, Brewers 12.
```

```
=> Game[winner 'Braves', loser 'Brewers', scores[4, 12]];
```

```
Orioles␣1,␣Cardinals␣5.
=> Game[winner␣'Orioles',␣loser␣'Cardinals',␣scores[1,␣5]];
```

The EBE system synthesizes the gap program of the above input/output pairs that has the gap pattern:

```
-1-␣-2-,␣-3-␣-4-. eol
```

and the gap replacement:

```
Game[winner␣'-1-',␣loser␣'-3-',␣scores[-2-,␣-4-]]; eol
```

where `eol` is a special *constant* which matches the end of the line. Numbers are used as *variables*. The other symbols are constants in the gap program.

This gap program matches any line that consists of a word (the first word) and a number (the first number) separated by a blank, followed by a comma and a blank, followed by a word (the second word) and a number (the second number) separated by a blank, followed by a point, and replaces each such line with a new constant field `Game[winner␣'`, followed by the first word, followed by a new constant field `',␣loser␣'`, followed by the second word, followed by a new constant field `',␣scores[`, followed by the first number and the second number separated by a comma and a blank, followed by a new constant field `]];`.

In [39] and [40] Nix has shown many results related to the inference of gap programs. However, gap programs are less expressive than regular expressions. Thus with this approach only fairly simple transformations (string to string) can be generated.

7.1.2 Procedural Approaches

The goal of procedural approaches is also to synthesize a transformation program from editing examples in a text editor. Unlike the function approach, the procedural approach synthesizes the program from *traces* which record *editing operations*. The system developed by D. H. Mo and I. H. Witten [38] is an example of the procedural approach that is also used to solve repetitive editing problems.

In this system, synthesizing the transformation program involves two steps: (1) Users edit some text block by editing operations defined in a simple interactive point-and-click editor, at the same time, the sequence of editing operations with their parameters (text strings) and their

positional information (context, distance and position) is recorded by the editor; (2) Then this sequence is generalized by combining some operations into higher level operations using heuristic rules, by abstracting their parameters and context. For abstracting concrete strings they use a mechanism similar to ours (Section 4.2.4.1 and Figure 4.2). The synthesized program is applied to a larger class of inputs. If the program does not behave appropriately on a new example in the class of inputs, the user has to modify the program manually and the program is extended automatically to accommodate the new example.

In [38] the authors describe detailed strategies to synthesize the transformation program on the basis of editing operations and their trace. Currently, however, the synthesized transformation programs appear to be limited to the treatment of flat structures, i.e., the sequence of editing operations is not further nested.

MarkItUp! restricts itself to structuring operations rather than arbitrary editing operations. Thus by analyzing the *output* of a number of structuring steps, it can determine more expressive recognition programs than gap programs. On the other hand, the structuring operations can be easily deduced from the marked-up examples, thus there is a closer correspondence between the editing *procedure* as perceived by the user and the generated *grammar*. These nested grammars also are more expressive than the procedures described in [38].

7.2 Inductive Learning and Learning Methods

One of the most widely studied forms of machine learning is learning from examples, or *inductive learning*, as it is more concisely called [34]. The task of learning is to induce general descriptions (or concepts) that explain the given input examples provided by a teacher or the environment and are useful for predicting new examples. Here, a *concept* can be regarded as an abstract description of a class of objects.

In this thesis the learning approach refers to two kinds of inductive methods: *grammatical inference* and *version spaces*.

7.2.1 Grammatical Inference

Grammatical inference uses formal grammars to represent the learned concepts and learns a grammar from a set of examples by drawing *inductive inferences*, which attempt to derive a complete and correct description of a given phenomenon from specific observations of that phenomenon or of parts of it. In [8] the author provides a general form of inductive inference

problems. The task of grammatical inference is to determine a formal grammar that can generate a given set of symbol strings.

The most important criterion of success of the inference methods is the *identification in the limit* which is defined by Gold [17]:

Definition 7.1 An inductive inference method M *identifies* a language L *in the limit* if, after a finite number of examples, M makes a correct guess and does not alter its guess thereafter. A class of languages is identifiable *in the limit* if there exists a method M such that given any language of the class and given any admissible example sequence for this language, M identifies the language in the limit.

To learn a correct grammar from a set of examples, Gold [17] proves that:

Theorem 7.1 Any class of languages containing all finite languages and at least one infinite language can not be identified *in the limit* from positive examples.

The theorem means that the class of languages can not be learned from positive examples. However, Gold [17] also points out that if a learning system generalizes the representation with some restrictions on the allowed result of the generalization in the form of background knowledge, an adequate language could be learned. For example, if the system could ask a teacher who always knows whether or not a given string is grammatical, the true language could be learned. This strategy is applied in the *MarkItUp!* learning system to generalize grammar from positive examples. Angluin [7] gave a complete characterization of the families learnable from positive examples.

In [11], the authors summarize *four* methods for grammatical inference. One of them is *refinement methods* which formulate a hypothesis grammar and then refine it on the basis of simplification heuristics and new training examples. B. Knobe and K. Knobe [31] address a refinement-method schema which repeatedly accepts new grammatical strings from a teacher, for each new string, the learning program generates a set of candidate productions which accept the new string and selects one of production as a new production to add to the grammar or merges old and new productions using heuristic rules.

The *MarkItUp!* system architecture is designed on the basis of the refinement methods of grammatical inference. The initial hypothesis grammar in *MarkItUp!* is generated in the learning cycle: markup cycle, in which the grammar incrementally accepts new grammatical

strings and is further refined by a set of concepts (Section 4.2) and a set of learning rules (Section 4.3).

A related approach to generate grammar in the field of document processing is presented by Ahonen *et al.* [4]. They collect a set of examples of structured documents, use a set of finite-state automata describing the examples, and choose generalization conditions to merge and modify the example automata so that general automata are generated. The resulting automata are transformed into regular expressions to get a readable grammar. Although they define some interactive operations the approach is not intended to be used for incremental learning.

7.2.2 Version Spaces

Given a set of training data and a language in which the desired concept must be expressed, Mitchell [35, 36] defines a version space to be “the set of all concept descriptions within the given language which are consistent with those training instances” (Mitchell [36]). The word *consistent* means that the concept description matches all the positive examples and none of the negative examples.

Mitchell noted that the generality of concepts imposes a partial order that allows efficient representation of the version space by the boundary sets S and G representing the most specific and most general concept definitions in the space. The S - and G -sets delimit the set of all concept descriptions consistent with the given data. The *candidate-elimination algorithm* manipulates the boundary set representation of a version space to create new boundary sets that represent a new version space consistent with all the previous instances plus the new one. For a positive example of the unknown concept the algorithm generalizes the elements of the S -set as little as possible so that they cover the new instance yet remain consistent with past data, and removes those elements of the G -set that do not cover the new instance. For a negative instance the algorithm specializes elements of the G -set so that they no longer cover the new instance yet remain consistent with past data, and removes from the S -set those elements that mistakenly cover the new, negative instance. When the S -set and G -set have the same single element, the element is the desired concept.

The shortcoming of Mitchell’s algorithm was that the set of candidate concept definitions must reflect strict consistence with data. Given some set of training data, only those concept definitions that correctly classify all instances are considered. If no such definition exists, the version space is empty.

Hirsh [26] developed a new algorithm, called *incremental version-space merging*, that generalizes Mitchell's notion of version space beyond strict consistency with data and proposes an incremental learning method. Given old and new version spaces based on different sets of information (about the same concept), the algorithm will find their intersection in S - and G -sets. The resulting version space reflects all the information of the given spaces and may contain many concept definitions in a concept description language representable by boundary sets.

In this thesis, the version-space approach is applied to the problem of discovering string patterns common to a set of strings (Section 4.2). Since string patterns are learned from positive examples, this version-space approach considers only the S -set. For each terminal in the grammar, there is a corresponding version space, called a reduced abstracted-list. Similarly with Hirsh's approach, there may be more than string patterns in the abstracted-list which are no more-specific-than the other. Since the string patterns are represented by regular expressions, the incremental version-space algorithm can simply compare a new string pattern with the string patterns in the existing abstracted-list and modify the abstracted-list so that the modified abstracted-list is reduced. It does not require a complex algorithm, such as the Hirsh's algorithm [26], to intersect the version spaces.

7.3 Application to Wrapping Semi-Structured Data

Recently, the extraction of structure from semi-structured electronic documents has gained considerable attention for the realization of wrappers. The basic functionality of wrappers is to translate queries and data from one data model into another [51].

There is no theoretical definition of semi-structured data [1]. BibTex is a kind of semi-structured data, as well as Web-documents. All of them are sources with implicit structure, not as rigid, static, or regular as standard database systems.

In order to query these semi-structured sources in a database-like fashion on the basis of their underlying structure, it is required to wrap them. With the help of wrappers queries can be converted into queries that can be processed by the underlying source and the native results are transformed into a format understood by the application.

One of applications to wrapping semi-structured data is to query Web-documents directly in a database-like fashion [9, 23].

In [22], Hammer et al. describe an approach to wrap Web-documents. It was be done in two steps: First, they developed a configurable extraction program for extracting semi-struc-

tured data from a set of WWW pages into object model. The extraction process is based on a specification file that consists of text patterns that characterize the data of interest on the WWW pages and the desired conversion into an object model with explicit structure. No learning takes place, thus all patterns need to be specified explicitly. The specification file is written by the user. The outcome of the extraction process is an object exchange model that contains the extracted data together with their structure and contents. To query the extracted result with predicates that are not originally supported by the source, the second step is to use the wrapper generation tools [22] to generate wrappers. On the basis of the wrapper, the contents on the WWW pages can be queried with an SQL-like language and the result is shown on the similar form. For a new source, the user has to write a new specification file.

In [9], Ashish et al. include machine learning techniques into their extraction program and try to generate wrapper semi-automatically from a few examples. This is accomplished in three steps: First, they give some general rules to identify tokens indicating different types (heading or sections) on a web page. With the help of the identified types and the format information, the system outputs a grammar describing the nesting hierarchy of sections in a page; Second, a parser for the learned pages for the source is generated. Such a parser can extract any selected section(s) from the page. Finally, a communication functionality is added to the wrapper, so that the wrapper is able to map function from queries to URLs, to fetch pages over a network, and to communicate between the wrapper and a mediator. In contrast to the approach described in this thesis, however, the learning heuristics are not generically applicable but only to Web-documents.

Extracting information from multiple heterogeneous data and integrating them in order to provide information is a challenging research topic in the database area, information retrieval, and data exchange. Wrappers provide integrated software components for accessing heterogeneous data sources, in which extracting information from the multiple heterogeneous sources is a basic step. With the dramatically increasing of all kinds available electronically data, extracting information from the data will play more important role in various application areas of data processing. For non-web documents, Smith et al. [44] and Klein et al. [30] separately introduce a document abstract model and automation parsing to extract another kind of semi-structured data. *MarkItUp!* approach described in this thesis and [15] use editing-by-example strategy to extract logical structure information of non-web documents. The learning strategy, however, could be also applied to Web-documents.

Conclusion

This thesis presents the *MarkItUp!* system for the incremental generation of structure recognition grammar (rules) from example structures. Techniques for generating an initial recognition grammar from marked-up examples, for abstracting concrete strings at the content level on the basis of a set of concepts, and for merging the structure of multiple examples at the structural level on the basis of a set of learning rules have been devised for this purpose. *MarkItUp!* incorporates these techniques into the markup cycle in which, with the help of a simple structure editor, the user can control the learning process and inconsistent electronic documents with repetitive but implicit format can be structurally enriched. The system is not only suitable for non-programmers who have to perform mark up manually, but also useful for programmers who have to write recognition programs to mark up documents.

The main contribution of the *MarkItUp!* system is to demonstrate the feasibility of inferring document-structure recognition grammar from structured examples in the domain of repetitive electronic documents. The primary contribution of this dissertation lies in a unified approach to manually mark up and automatically mark up documents on the basis of machine learning. The *MarkItUp!* system is developed, analyzed, and implemented which turned out to be an effective aid in automating the recognition of a large class of electronic documents.

Another contribution of this dissertation lies in combining the techniques of version spaces and grammatical inference into the practical application of document logical structure recognition.

To implement the techniques in *MarkItUp!*, the following related procedures are realized :

- (1) An algorithm to order concepts predefined by the user according to the more-specific-than relation.
- (2) Algorithms for an incremental version-space approach to abstract concrete contents.

- (3) A set of rewrite rules which *simplify* existing grammars in a prescribed manner – unification and simplification rules.
- (4) A set of rewrite rules which *generalize* existing grammars in a prescribed manner – abstraction rules.

These procedures are implemented in a grammatical inference learning cycle by an object-oriented language: SMALLTALK. The details of the implementation are described in Chapter 6. This implementation provides an example of how such procedures might be implemented in the learning cycle.

The use of an incremental editing-by-example approach to propose optimal new examples to direct grammatical inference was demonstrated in chapter 4. The principle employed there generalizes to other regular languages, but the implementation of the general method requires language-specific routines.

The applicability of the grammar approach to grammatical inference has been demonstrated both theoretically and empirically in *MarkItUp!*. The grammatical inference learning cycle has been illustrated in two different problem domains: (1) learning contents features from given concepts – the content level learning, (2) learning document-structure recognition rules from the entire example (exhaustive learning strategy) or from the parts of an example (partial learning strategy) – the structural level learning.

At the content level, the algorithm begins by organizing all concepts predefined by the user in a concept base, ordering the concepts in the concept base in order to get a sorted list, then learning concepts from the ordered concepts list with the version-space approach. Features of the learning concepts algorithm includes:

- (1) Each terminal has its own version-space: a reduced abstracted-list, which may contain more than one string patterns.
- (2) Learning a new concrete string does not require to consider previously examined example strings. As a result, the system does not need to store old example strings.
- (3) Learning results are independent of the order in which examples are presented.

At the structural level, the rewrite rules provide the basis for a learning procedure that generalizes the document-structure recognition grammar. The features of the rewrite rules are:

- (1) The sequence of unification rules has no special meaning, i.e. it does not express a call-order.
- (2) The unification rules are chosen by the type of the elements (parameters). After applying these rules, the unified grammar is equivalent to the old grammars.
- (3) After applying abstraction rules, the new grammar slightly deviates from the old one.
- (4) Learning results of applying the rewrite rules are dependent on the ordering in which examples are presented.

By the grammatical inference learning cycle, a recognition program acquires the ability to describe what can and cannot be determined by the generalized grammar for new examples.

The discussion of sequences of arbitrary elements gives an approach to generate a general expression of unordered elements. It is significant to complete the *MarkItUp!* system for SGML applications.

MarkItUp! has been mainly used by the author of this thesis. The personal experiences indicate that the chosen approach is very adequate for structuring large amounts of unstructured documents with reasonable effort. As discussed in Section 6.7, especially for documents with some repetition factor, such as bibliographies or schedules, only a few manual markups in the range of 5–10 examples suffice structure over 95% of the complete source correctly.

Future work will be devoted to the following refinements and extensions of the presented approach:

- (1) Implementation of the general expression algorithm for sequences of arbitrary ordering in the *MarkItUp!* system;
- (2) Further optimization of the algorithm for general expressions to make the implementation simple;
- (3) Integration with techniques of structure recognition for scanned documents as opposed to ASCII documents. These techniques utilize specialized rules for dealing with layout information such as font properties and two-dimensional source structure. To generate such rules from intellectually structured exam-

ples the unification & abstraction metarules have to be extended, for example with taxonomies of font properties and a calculus for two-dimensional grammars;

- (4) Development of concepts for detecting and handling ambiguity in the recognition rules. Currently, the degree of abstraction performed at string level and at structural level is hardwired. This can lead to ambiguous recognition rules that do not uniquely characterize the structure of example documents. An approach which automatically detects such ambiguities and accordingly performs less abstraction would be more convenient.

Bibliography

1. Abiteboul, S.,
“Querying SemiStructured Data”,
Proceedings ICDT’97, 1997
2. Aho, A. V., Hopcroft, J. E., and Ullman, J. D.,
Data structures and algorithms,
Addison-Wesely series in computer science and information processing, March, 1979.
3. Aho, A. V., Sethi, R., and Ullman, J. D.,
Compilers – principles, techniques, and tools,
Addison-Wesely series in computer science, 1986.
4. Ahonen, H., and Mannila, H., and Nikknen, E.,
“Forming grammars for structured documents: an application of grammatical inference”,
Grammatical Inference and Applications, Second International Colloquium, ICGI-94, pp.
153-167, September, 1994.
5. American National Standards Institute,
*Information processing – text and office systems – standard generalized markup
language(SGML)*,
ISO 8879-1986(E), ANSI, New York, 1986.
6. Angluin, D.,
“On the complexity of minimum inference of regular sets”,
Information and Control, Vol. 39, pp. 337-350, 1978.
7. Angluin, D.,
“Inductive inference of formal languages from positive data”,
Information and Control, Vol. 48, pp. 117-135, 1980.
8. Angluin, D. and Smith, C. H.,
“Inductive inference: Theory and methods”,
Computing Surveys, Vol. 15, No. 3, September 1983.
9. Ashish, N. and Knoblock, C.
“Wrapper Generation for Semi-Structured Internet Sources”,
Workshop on Management of Semistructured Data, Ventana Canyon Resort, Tucson,
Arizona, May 16, 1997.
10. Brown, H.,
“Standards for structured document”,
The computer journal, Vol. 32, No.6, 1989.

11. Cohen, P. R. and Feigenbaum, E. A.,
The handbook of artificial intelligence, Vol. III,
William Kaufmann, Inc. Los Altos, California, 1982.
12. Conway, A.,
“Page Grammars and Page Parsing – A Syntactic Approach to Document Layout
Recognition”
ICDAR 93 Second International Conference on Document Analysis and Recognition,
Tsukuba Science City, Japan, 1993.
13. Coombs, J. H., Renear, A. H., and Derose, S. J.,
“Markup systems and the future of scholarly text processing”,
Communications of the ACM, Vol. 30, No. 11, Nov. 1987.
14. Dengel, A.,
“Initial learning of document structure”,
ICDAR 93 Second International Conference on Document Analysis and Recognition,
Tsukuba Science City, Japan, 1993.
15. Fankhauser, P. and Xu, Y.
MarkItUp! – An incremental approach to document structure recognition
Electronic Publishing, Vol. 6, No. 4, Dec. 1993, pp. 447-456
16. Furuta, R.,
“Concepts and models for structured documents”,
Structured documents, the Cambridge Series on Electronic Publishing. Cambridge
University Press, pp 7-38, 1989.
17. Gold, E. M.,
“Language identification in the limit”,
Information and Control, Vol. 10, pp. 447-474, 1967.
18. Gold, E. M.,
“Complexity of automaton identification from given data”,
Information and Control, Vol. 37, pp. 302-320, 1978.
19. Golumbic, M. C.,
Algorithmic graph theory and perfect graphs,
Academic Press, 1980.
20. Göttke, T. and Fankhauser, P.,
DREAM 2.0 User Manual,
Arbeitspapiere der GMD 660, July 1992.

21. Götke, T.,
“Strukturmarkierung von Dokumenten aus Informationsdatenbanken”,
Diplomarbeit, Technische Hochschule Darmstadt Fachbereich Informatik, February, 1993.
22. Hammer, J., Carcia-Molina, H., Cho, J., Aranha, R., and Crespo, A.
“Extracting Semistructured Information from the Web”,
Workshop on Management of Semistructured Data, Ventana Canyon Resort, Tucson, Arizona, May 16, 1997.
23. Hammer, J., Brenning, M., Carcia-Molina, H., Nesterov, S., Vassalos, V., and Yerneni, R.,
“Template-based wrappers in the tsimmi system”,
In Proceedings of ACM SIGMOD International Conference on Management of Data (Demonstration Track), Tucson, AZ, 1997
24. Handley, J. and Weibel, S.,
“ADAPT: Automated document analysis processing and tagging”,
Document Manipulation and Typography, Proceedings of the EP’90 Conference, Cambridge University Press, pp. 183-192, 1990.
25. Herwijnen, Eric van,
Practical SGML,
Kluwer Academic Publishers, 1990.
26. Hirsh, H.,
Incremental version-space merging: A general framework for concept learning,
PhD thesis, Stanford University, 1989.
27. Hopcroft, J. E. and Ullman, J. D.,
Introduction to automata theory, language, and computation,
Addison-Wesley, Reading, Mass., 1979.
28. Ingold, R., Bonvin, R., and Cory, G.,
“Structure recognition of printed documents”,
Document Manipulation and Typography, the Cambridge Series on Electronic Publishing
Cambridge University Press, pp. 59-70, 1988.
29. Ingold, R. and Armangil, D.,
“A top-down document analysis method for logical structure recognition”,
ICDAR 91 First International Conference on Document Analysis and Recognition,
Saint Malo, France, pp. 41-49, 1991.

30. Klein, B. and Fankhauser, P.
“Error tolerant document structure analysis”,
Digital Libraries, 1997, pp. 344-357
31. Knobe, B. and Knoeb, K.,
“A method for inferring context-free grammars”,
Information and control, Vol. 31, pp. 129-146, 1976.
32. Knuth, D. E.,
The T_EXbook,
Addison-Wesley, Reading, Massachusetts, 1984.
33. Marovac, N.,
“Document recognition – concepts and implementations”,
SIGOIS Bulletin Vol. 13, No. 3, pp. 28-38, 1992.
34. Michalski, R. S., Carbonell, J. G., and Mitchell, T. M.,
Machine Learning: An Artificial Intelligence Approach,
Morgan Kaufmann Publishers, Inc., 1983.
35. Mitchell, T. M.,
“Version spaces: a candidate elimination approach to rule learning”,
Proceedings of IJCAI’77, pp 305-310, 1977.
36. Mitchell, T. M.,
Version spaces: an approach to concept learning,
PhD thesis, Stanford University, 1978.
37. Mitchell, T. M.,
“Generalization as search”,
Artificial Intelligence, Vol. 18, pp 203-225, 1982.
38. Mo, D. H. and Witten, I. H.,
“Learning text editing tasks from examples: a procedural approach”,
Behavior & Information Technology, Vol. 11, No. 1, pp. 32-45, 1992.
39. Nix, R.,
Editing by example,
Ph.D. Dissertation, Computer Science Department, Yale University, 1983.
40. Nix, R.,
“Editing by example”,
Proceedings of the 11th ACM Symposium on Principles of Programming Languages,
pp.186-195, 1984.

41. Ossanna, J. F.,
NROFF/TROFF user's manual,
UNIX Programmers Manual, 1979.
42. Rostek, L. and Alexa M.,
“Marking up in TATOE and exporting to SGML – Rule development for identifying NITFG categories”,
submitted for special issue on the *ACH-ALLC '97 Conference*, to be published by
Computers and the Humanities, March 1998.
43. Schmidt, J. and Putz, W.,
“Knowledge acquisition and representation for documents structure recognition:
the CAROL project”,
Proceedings of the Ninth IEEE Conference on Artificial Intelligence in Applications,
Orlando/Florida March 1-5, 1993, IEEE Computer Society Press 1993.
44. Smith, D. and Lopez, M.,
“Information extraction for semi-structured documents”,
Workshop on Management of Semistructured Data, Ventana Canyon Resort, Tucson,
Arizona, May 16, 1997.
45. Srihari, S. N. and Zack, G. W.,
“Document image analysis”,
Proceedings of the 8th International Conference on Pattern Recognition, Paris, France,
pp. 434-436, 1986.
46. Stallman, R.,
GNU emacs manual,
Oct., 1986.
47. Tang, Y. Y., Suen, C. Y., Yan, C. D., and Cheriet, M.,
“Document analysis and understanding: A brief survey”,
ICDAR 91 First International Conference on Document Analysis and Recognition,
Saint Malo, France, 1991.
48. Toyoda, J., Nouguchi, Y., and Nishimura, Y.
“Study of extracting Japanese newspaper article”,
Proc. 6th Int. Conf. on Pattern Recognition, pp. 1114-1115, 1982.
49. Vanlehn, K. and Ball, W.,
“A version space approach to learning context-free grammars”,
Marching Learning, Vol. 2, No. 1, pp. 39-74, 1987.

50. Warmer, J. and Egmond, S. V.,
“The implementation of the Amsterdam SGML parser”,
Electric Publishing, Vol. 2, No. 2, pp. 65-90, July 1989.
51. Wells, D.,
“Wrappers Survey”,
URL: <http://www.objs.com/survey/wrap.htm>
52. Whiteside, M.,
IMSYS, the Intelligent Markup System,
Avalanche Development Company, Boulder, Colorado, 1986.
53. Wilcox, L. D. and Spitz, A. L.,
“Automatic recognition and representation of documents”,
Document Manipulation and Typography, Proceedings of the EP’88 Conference,
Cambridge University Press, pp. 47-57, 1988.

Appendix: List of Figures and Tables

Fig. 2.1	Example of a DAG	14
Fig. 3.1	The system architecture of DREAM	23
Fig. 3.2	System overview of MarkItUp!	30
Fig. 3.3	The initial state of the structure editor	32
Fig. 3.4	An example loaded into the structure editor	33
Fig. 3.5	Manually marking up a string in the structure editor	34
Fig. 3.6	The result of manually marking up a string in the structure editor	35
Fig. 3.7	The manually marking up result of the example in Figure 3.4	36
Fig. 4.1	MarkItUp! learning	46
Fig. 4.2	An example of the concept base	50
Fig. 5.1	The permutation graph of $p = [3, 4, 2, 1]$	72
Fig. 5.2	A union graph is a PG	73
Fig. 5.3	A union graph is not a PG	74
Fig. 5.4	A PG	78
Fig. 6.1	System architecture of MarkItUp!	82
Fig. 6.2	Organization of the structure editor	83
Fig. 6.3	An example of structure editing	84
Fig. 6.4	Views and menus of concept editor	88
Fig. 6.5	Prototype of a concept editor	89
Fig. 6.6	A graphic description of the concept base in Fig. 6.5	90
Fig. 6.7	A prototype of concept editor for the concepts in Fig. 4.2	92
Fig. 6.8	DREAM DSD of the example in Figure 6.3	98
Fig. 6.9	The tagged document of Document-Sample 6.1 is created by the DREAM DSD in Figure 6.8	99
Fig. 6.10	A tentatively marked-up example generated by the DREAM DSD in Figure 6.8	100
Fig. 6.11	The user-corrected example using the exhaustive learning strategy ..	101
Fig. 6.12	The user-corrected example using the partial learning strategy	101
Fig. 6.13	Changed rules in DSD based on the exhaustive learning strategy	102
Fig. 6.14	Changed rule in DSD for learning an optional element on the basis of the partial learning strategy	104
Fig. 6.15	Changed rules in DSD for learning a repetition element on the basis of the partial learning strategy	104
Fig. 6.16	Changed rules in DSD for learning a new string on the basis of the partial learning strategy	105
Table 6.1	The experiment results generated by applying concept set S1	107
Table 6.2	The experiment results generated by applying concept set S2	108