A System for Document Analysis, Translation, and Automatic Hypertext Linking

by

Guillermo Andrés Averboch

Thesis submitted to the faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science and Applications

©Guillermo Andrés Averboch and VPI & SU 1995

APPROVED:

Dr. Lenwood S. Heath, Chairman

Dr. Edward A. Fox

Dr. James D. Arthur

June, 1995

Blacksburg, Virginia
A System for Document Analysis, Translation, and Automatic Hypertext Linking

by

Guillermo Andrés Averboch

Committee Chairman: Dr. Lenwood S. Heath

Department of Computer Science

(ABSTRACT)

A digital library database is a heterogeneous collection of documents. Documents may become available in different formats (e.g., ASCII, SGML, typesetter languages) and they may have to be translated to a standard document representation scheme used by the digital library.

This work focuses on the design of a framework that can be used to convert text documents in any format to equivalent documents in different formats and, in particular, to SGML (Standard Generalized Markup Language). In addition, the framework must be able to extract information about the analyzed documents, store that information in a permanent database, and construct hypertext links between documents and the information contained in that database and between the document themselves. For example, information about the author of a document could be extracted and stored in the database. A link can then be established between the document and the information about its author and from there to other documents by the same author. These tasks must be performed without any human intervention, even at the risk of making a small number of mistakes.

To accomplish these goals we developed a language called DELTO (Description Language for Textual Objects) that can be used to describe a document format. Given a description for a particular format, our system is able to extract information from documents in that format, to store part of that information in a permanent database, and to use that information in constructing an abstract representation of those documents that can be used to generate equivalent documents in different formats.

The system originated from this work is used for constructing the database of Envision, a Virginia Tech digital library research project.
ACKNOWLEDGEMENTS

I would like to thank Dr. Heath, my advisor, for always believing in me and in what I was doing, when not too many more people did, for his continual support during the year and a half that this work demanded, and, of course, for his help in doing this thesis. I would also like to thank Dr. Fox for always being ready to help and to provide ideas, for his very valuable comments that significantly improved this thesis, and for serving in my committee. I would also like to thank Dr. Arthur for his important comments and for serving in my committee. Thanks also to Scott Guyer, a member of the Envision group. It was always easier to work when Scott was around, so that I could take a break and go bother him.
# TABLE OF CONTENTS

1 INTRODUCTION

1.1 The Envision Project ................................................. 1
1.2 Scope ........................................................................ 2
  1.2.1 Conversion to SGML ............................................. 2
  1.2.2 Object Extraction .............................................. 3
  1.2.3 Construction of Hypertext Links ............................. 4
1.3 Problem Statement .................................................... 4
1.4 Overview of the Solution ......................................... 5
1.5 Thesis Outline ......................................................... 7

2 OVERVIEW OF RELATED WORK ........................................ 8

2.1 Data Conversion and Translation ................................. 8
2.2 Text Extraction ....................................................... 12
2.3 Summary ............................................................... 15

3 SYSTEM ARCHITECTURE .............................................. 16

3.1 Overview ................................................................... 16
3.2 Object Database System (ODB) ................................. 18
3.3 DELTO Compiler ..................................................... 21
3.4 DELTO Analyzer ...................................................... 24
  3.4.1 Lexical Analyzer ................................................ 26
  3.4.2 Object Repository Generator ............................... 30
  3.4.3 Object Repository Analyzer ............................... 32
3.5 DELTO Translator ..................................................... 34
CONTENTS

4 DELTO: LANGUAGE REFERENCE

4.1 Introduction ........................................... 37
4.2 The Syntax of DELTO ................................. 37
4.3 Format Descriptions ................................. 39
4.4 Class Definitions ..................................... 41
  4.4.1 Attributes ....................................... 43
  4.4.2 Components ...................................... 45
  4.4.3 Expressions and Predicates .................... 49
  4.4.4 Predicates ...................................... 52
  4.4.5 Referencing Attributes and Components ...... 58
  4.4.6 Functions ...................................... 65
  4.4.7 Regular Expressions ............................ 67
  4.4.8 Combining Expressions ......................... 78
  4.4.9 Non-Textual Objects ............................ 83
  4.4.10 String Component Descriptions ............... 84
  4.4.11 Object Database Interface Description ....... 85

5 RETARGET: LANGUAGE REFERENCE ...................... 99

5.1 Introduction .......................................... 99
5.2 Templates ........................................... 99
5.3 Template Statements ................................ 101
  5.3.1 Attribute and Component References ........... 102
  5.3.2 Process Statement ............................... 104
  5.3.3 Call Statement ................................ 108
  5.3.4 If Statement ................................... 110
  5.3.5 For Statement ................................... 111
  5.3.6 Assignment Statement ........................... 112
CONTENTS

6 EXPERIMENTAL RESULTS 115
   6.1 Format Descriptions and Translation Specifications .......... 116
   6.2 Pseudo-SGML Full Text Articles .......................... 118
   6.3 SGML Bibliographic Documents ............................. 122
   6.4 Strengths and Weaknesses ................................ 128

7 CONCLUSIONS AND FUTURE WORK 130
   7.1 Conclusions .............................................. 130
   7.2 Recommendations for Future Work .......................... 132
   7.3 Concluding Remarks ...................................... 133

A MAN PAGES 137
   A.1 The DELTO Directory ...................................... 137
   A.2 ODB Interface ............................................ 137
   A.3 Dictionary Interface ..................................... 141
   A.4 DELTO Compiler .......................................... 147
   A.5 Lexical Analyzers ........................................ 148
   A.6 Object Repository Generator .............................. 151
   A.7 Object Repository Analyzer ............................... 151
   A.8 DELTO Translator ........................................ 152
   A.9 DELTO Execution Utility ................................ 153

B HINTS FOR WRITING DELTO FORMAT DESCRIPTIONS 157
   B.1 Recognizing Objects ..................................... 157
      B.1.1 Recognizers ......................................... 158
   B.2 Bindings ................................................... 161

C DELTO FORMAT DESCRIPTIONS USED FOR ENVISION 164
   C.1 General Format Description ............................... 164
   C.2 Format Description for SGML Bibliographic Documents ......... 174
CONTENTS

C.3 Format Description for Pseudo-SGML Full-Text Articles ............... 184

D RETARGET SPECIFICATIONS USED FOR ENVISION 202
D.1 Retarget Specification for SGML Bibliographic Documents ............. 202
D.2 Retarget Specification for Pseudo-SGML Full-text Articles ............ 205

E ODB CLASSES DEFINED FOR ENVISION 212
E.1 Definition for the Document Class .................................... 212
E.2 Definition for the Person Class ...................................... 213
E.3 Definition for the Institution Class ................................. 213
E.4 Definition for the Publication Class ................................ 214
E.5 Definition for the Indexing_term Class ............................... 214

F DELTO GRAMMAR 215

G RETARGET GRAMMAR 223
# LIST OF FIGURES

1.1 The problem ................................................................. 5
3.1 High level architecture of the DELTO System ..................... 19
3.2 ODB class definitions for document and person objects .......... 22
3.3 ODB objects .............................................................. 23
3.4 Architecture of the DELTO Analyzer ................................ 25
3.5 Output generated by the SGML lexical analyzer ................... 29
4.1 EBNF conventions used to present the grammar of DELTO ......... 38
4.2 Syntax rules for format descriptions ............................... 39
4.3 Syntax rules for class definitions .................................... 42
4.4 Syntax rules for attribute blocks ................................... 46
4.5 A definition for two attributes ..................................... 46
4.6 Syntax rules for component blocks .................................. 48
4.7 Components definition for a bibliographic entry class .......... 49
4.8 Syntax rules for expressions ......................................... 51
4.9 Syntax rule for references to attributes .......................... 58
4.10 Syntax rules for references to components ....................... 60
4.11 Components for the date class ..................................... 61
4.12 Syntax rules for function invocations ............................ 65
4.13 Syntax rules for regular expressions .............................. 68
4.14 Scoping example ...................................................... 70
4.15 Scoping example ...................................................... 71
4.16 Component bindings example ...................................... 72
LIST OF FIGURES

4.17 Syntax rules for top level expressions ........................................ 78
4.18 Syntax rules for strength blocks .................................................... 80
4.19 Usage of strength indicators ......................................................... 80
4.20 Syntax rule for frequency blocks .................................................... 80
4.21 Frequency blocks ........................................................................ 81
4.22 Syntax rule for description blocks ................................................... 82
4.23 A description block ..................................................................... 82
4.24 A class definition for person ......................................................... 83
4.25 Syntax rule for string component descriptions ................................ 84
4.26 A class definition for university_name .......................................... 86
4.27 Syntax rule for object database interface descriptions .................. 88
4.28 Syntax rules for access keys ........................................................... 90
4.29 Access keys definition .................................................................. 90
4.30 Access keys definition with strength indicators ............................ 92
4.31 Match description example ............................................................ 93
4.32 Syntax rule for match descriptions ................................................ 93
4.33 Syntax rules for action blocks ....................................................... 94
4.34 Actions that index an ODB object ................................................ 97
4.35 A complete ODB Interface ........................................................... 98

5.1 Syntax rules for translation specifications and templates ............... 100
5.2 A translation specification with no statements ............................... 101
5.3 Output generated by the translation specification of Figure 5.2 ....... 101
5.4 Syntax rules for template statements .............................................. 102
5.5 A translation specification that includes a component reference .... 102
5.6 An input document ...................................................................... 103
5.7 Output generated from the document of Figure 5.6 using the specification of Figure 5.5 ................................................................. 103

ix
LIST OF FIGURES

5.8 Syntax for component references .................................. 104
5.9 A translation specification that references odbid components .......... 104
5.10 Syntax rules for the process statement .............................. 105
5.11 A translation specification that uses the process statement .......... 106
5.12 Output generated from the document of Figure 5.6 using the specification of Figure 5.11 ................................................. 106
5.13 A translation specification that uses the process-ignore statement .... 107
5.14 Output generated from the document of Figure 5.6 using the specification of Figure 5.13 ................................................. 107
5.15 Syntax rule for call statements ........................................ 108
5.16 A translation specification that uses the call statement .......... 108
5.17 A modified input document ............................................ 109
5.18 Output generated from the document of Figure 5.17 using the specification of Figure 5.16 ................................................. 109
5.19 Syntax rules for if statements ......................................... 110
5.20 Use of the if statement ................................................ 110
5.21 Syntax rules for for statements ....................................... 111
5.22 Use of the for statement ................................................ 111
5.23 Output for the specification of Figure 5.22 ............................ 111
5.24 Use of the if and for statements ..................................... 112
5.25 Output for the specification of Figure 5.24 ............................ 112
5.26 Syntax rules for assignment statements .............................. 113

6.1 A portion of a Pseudo-SGML full text article .......................... 119
6.2 Document objects created for the full-text article ..................... 120
6.3 Person objects created for the full-text article ........................ 121
6.4 Indexing_Term objects created for the full-text article ............... 121
6.5 Full text article translated to SGML conforming to the Envision DTD ... 123
LIST OF FIGURES

6.6 Three SGML bibliographic entries ........................................ 124
6.7 Document objects created for the SGML bibliographic entries .......... 125
6.8 Person objects created for the SGML bibliographic entries .............. 126
6.9 Resulting output for the SGML bibliographic entries .................... 127

A.1 Starting menu of the ODB interface .................................... 138
A.2 ODB menu for accessing read-write databases ............................ 139
A.3 ODB menu for accessing read-only databases ............................ 140
A.4 First menu displayed by the dictionary interface ....................... 142
A.5 Dictionary interface menu for read-write dictionaries .................. 143
A.6 Dictionary interface menu for read-only dictionaries ................... 144
A.7 Entries added into a dictionary ........................................... 145
A.8 A dictionary data file ..................................................... 146
A.9 A substitutions data file .................................................. 147
A.10 A replacement table for escaping some characters ...................... 154
A.11 Usage of the DELTO execution utility .................................. 154
LIST OF TABLES

3.1 Dictionary statistics ........................................... 28

6.1 Statistics of processed documents .............................. 117
Chapter 1

INTRODUCTION

1.1 The Envision Project

Project Envision [4, 17] was launched in 1991 at Virginia Tech. It aims at constructing a digital library that can provide on-line access to a multimedia collection of computer science literature with full-text searching and full-content retrieval capabilities.

The Envision database stores documents in SGML (Standard Generalized Markup Language) [14, 31]. However, incoming documents may be in any format, which means that they have to be converted to SGML. As part of the Envision effort, an SGML DTD (Document Type Definition) has been developed [3]. This DTD defines the SGML representation used to store all documents that compose the Envision database. SGML used in combination with our DTD is an effective way to represent the logical content of a document, as opposed to the presentation procedures to display the document. Actually, most existing word processor formats represent presentation information instead of information about the document structure [7].

An important goal of Envision is to identify objects from the contents of the documents stored in its database, such as persons, institutions, journals, conferences, or publications. SGML is an ideal representation scheme to mark those kinds of objects within documents. Once objects are identified, hypertext links can be established between related objects. These links may be followed by Envision users while they view documents, enabling them to access documents that are related in different ways to the one that they are currently viewing.
CHAPTER 1. INTRODUCTION

1.2 Scope

Initially, document conversions in Envision were handled in an ad hoc way. Translators were constructed specifically for some of the formats in which incoming documents were received. These translators were able to generate an equivalent SGML representation of the incoming documents. However, there was a need to develop a new translator each time that documents in new formats were received, and the results generated by those translators were not always correct, which meant that some manual editing of the resulting documents had to be done. This document conversion process was extremely time-consuming and labor-intensive. In addition, the cost of this process was going to become prohibitive as the size of the database increased, and as more sources of documents (each with its own document format) were available for Envision.

No document analysis was performed initially except for what was needed to convert the documents to SGML. No objects were extracted from the documents, no hypertext links were established, and there was no provision made to accomplish these complex tasks.

Therefore, the following problems need to be addressed:

• conversion to SGML;

• object extraction;

• construction of hypertext links.

1.2.1 Conversion to SGML

We need a way to convert a large volume of textual data to SGML (and in particular to SGML conforming to our DTD). We need an extremely flexible system to accomplish this task, one that is able to translate documents in different formats to SGML and that is not restricted to a fixed number of input formats. We do not know what the formats of the future are going to be. We need our conversion system to be able to accommodate any formats, even ones that we do not currently know. A non-fixed target format is also
CHAPTER 1. INTRODUCTION

desirable. This would give Envision the possibility of migrating to document representations other than the one currently used. In particular, it would allow Envision to update its DTD, or to use a completely different one.

1.2.2 Object Extraction

An important part of the Envision project is to identify computer science objects, that is, objects that may appear in the literature and that may be of interest to computer scientists. A partial list of these kinds of objects includes

- persons
- institutions
- journals
- conferences
- publications
- algorithms
- theorems
- formulas

An automated way to extract these kinds of objects from documents is also needed. The task of extracting objects from textual documents can be done by humans, but for a project like Envision, with a database that is expected to grow rapidly, the cost of such a process becomes prohibitive. Again, a flexible system is needed, one that is able to recognize different kinds of objects and that is not restricted to certain classes of objects. With such a system, the set of relevant objects for Envision may change as time goes by.

Identified objects may serve different purposes. For example hypertext links may be constructed between a document object and the person objects that represent the authors
CHAPTER 1. INTRODUCTION

of the document. An algorithm object could, if it is appropriately identified and interpreted, be associated with an executable version or with an animation of it.

1.2.3 Construction of Hypertext Links

Given the objects recognized from all the documents in the Envision database, an automated way to construct hypertext links between them is needed. We also need a flexible system that is not tied to some fixed set of hypertext links, but that can be easily modified for generating different kinds of links. We also need a way to represent and store those links. There are two kinds of links that we consider: those between objects located in the same textual document, and those between objects that exist in different documents.

1.3 Problem Statement

As can be seen from the previous section, the problems that we had to address are complex and ill-defined. The three problems described are different, though they are related. For example, to generate an SGML document that conforms to the Envision DTD, a certain understanding of the structure of the input document must be acquired. Otherwise it is impossible to tag parts of a document as, for example, a chapter or to determine who the authors are. A similar kind of understanding is needed to extract information about objects from textual documents and to construct hypertext links between those objects.

These three different, but related, problems lead to the following problem that this thesis intends to solve.

Design and implement a flexible framework to analyze textual documents in any format, with the results of the analysis used to generate an equivalent document in SGML or in another structured format, to extract different kinds of objects from those documents, and to establish links between the extracted objects (and, indirectly, between the documents in which those objects occur).

As we mentioned before, this framework must be flexible enough so that practically any
CHAPTER 1. INTRODUCTION

Figure 1.1: The problem

input format can be accommodated and that the set of objects to be extracted from input documents is not fixed but is specifiable through a high-level language. The kinds of links to be established must also be specifiable through that language. A picture that represents the problem is given in Figure 1.1.

1.4 Overview of the Solution

To solve this problem we created DELTO (Description Language for Textual Objects). DELTO is used to describe document formats in terms of the classes of objects that can occur within documents in the described format. The DELTO System can process documents in any format for which there is a DELTO format description. DELTO is a mostly declarative language. A format description does not specify how to process a document but rather describes how it is organized. Using the information provided in the format description, the
CHAPTER 1. INTRODUCTION

DELTO System is able to construct an abstract representation of an input document. The quality of this representation is directly related to the quality of the format description.

The abstract representation of an input document is a database of objects extracted from the document. Some of the objects extracted from a document may be stored in a permanent database of objects. The specification of what objects are to be stored also is given in the format descriptions. From this abstract representation, a new representation of the input document, in any format, may be constructed. For this purpose we designed Retarget, a simple procedural language that specifies how to translate the contents of a database of objects (the abstract representation of a document) into a new document, typically in a format that is different from the format of the input document.

Our aim is not to achieve a perfect translation but rather a high-quality one. Perfection is not achievable, given the complexity of the problem and the kinds of documents that we need to deal with. Some of the documents that we have to process are no more than raw English text, with little or no markup at all. In addition, the contents of these documents are usually "dirty", that is, names may be misspelled, parts of a document may be missing, and a variety of other flaws may be present. DELTO itself allows for the specification of uncertain facts, that is, facts that may or may not be true for all documents in a specific format. For example, we could specify that a document in a particular format must always include front matter and that it frequently includes back matter, although a document without back matter could still be a valid one. Weights are assigned to all objects recognized by the DELTO System to indicate the confidence in the correctness of the performed process.

The entire system is supported by a very simple, flexible, and powerful database named ODB that we developed. All databases generated by the DELTO System, including those that represent documents and the permanent database of objects, are managed by ODB.
CHAPTER 1. INTRODUCTION

1.5 Thesis Outline

The main body of the thesis consists of Chapters 2 through 7. In Chapter 2, we present a review of solutions found by other researchers to some of the problems (or to related problems) addressed by this thesis. Chapter 3 presents a description of the architecture of the system that was developed as part of this thesis. In Chapter 4, we introduce the syntax and semantics of DELTO, the language used to describe document formats, and the most important contribution of this work. Chapter 5 provides a description of Retarget, the translation specification language that describes the output formats of the system. Chapter 6 presents the results that have been achieved by our system. In Chapter 7, we give our conclusions and suggestions for future work.

Appendix A describes how to use the software components that we have developed for this thesis. Appendix B provides some suggestions that may facilitate the process of writing DELTO format descriptions that can be processed by our current implementation. Appendix C contains three DELTO format descriptions that we have used for our evaluation. Appendix D presents two Retarget specifications that have been also used for our evaluation. Appendix E describes the ODB classes that we developed to be used with the format descriptions introduced in Appendix C. Appendices F and G present the complete grammars of DELTO and Retarget, respectively.
Chapter 2

OVERVIEW OF RELATED WORK

For the purpose of reviewing previous research related to this thesis we classify research in two categories:

- data conversion and translation;
- text extraction.

Within the data conversion and translation category, we include those efforts aimed at translating textual documents from one format to another one. We do not include research related to translations from one human language to another one, because this is outside the scope of our work.

The text extraction category includes efforts aimed at extracting useful information from textual documents. The extracted information may be used for different purposes, like constructing a database or categorizing the document from which the information is extracted.

2.1 Data Conversion and Translation

Mamrak et al. [24, 25] developed Chameleon, which they claim to be a comprehensive data translation system. Among its principal characteristics, Chameleon is based on a formal model of the translation task and it supports the building of translation tools.

For a given data format, once the appropriate specifications are provided, Chameleon generates two translators: one that can convert from the specified data format to a standard form, and another one that takes a document in the standard form and generates an equivalent one in the specified format (also called the variant form). Standard forms accepted
CHAPTER 2. OVERVIEW OF RELATED WORK

by Chameleon must be braced languages. In a braced language, the beginning and end of every piece of information must be explicitly marked. In particular, Chameleon uses SGML as the standard form.

Translation from a given format into the standard form is called translation up, while translation in the other direction is called translation down. Translation down can be accomplished automatically with Chameleon, but translation up may need human intervention to help the system decide between a number of possible choices at different points of the translation process.

There are several problems and limitations with Chameleon. For one thing, there is a need to specify a context free grammar for every format for which translators are needed. Of course, specifying a grammar that can describe all documents in a given format is not a simple task. Documents within the same format do not always follow the same patterns. There may be errors in the documents, or things that have not been accounted for in the specified grammar. Chameleon translators will usually abort if the input documents cannot be appropriately parsed with respect to the given grammar.

Chameleon translators may require human intervention when going from a variant format into the standard form (translation up). These translators are LR parsers generated by yacc [23]. Sometimes, due to ambiguity problems, these parsers may encounter conflicts (reduce-reduce, or shift-reduce). In the event of these conflicts, Chameleon translators prompt the user for a choice. For one thing, to be able to select the correct choice a person must understand the way in which LR parsers work, and the nature of the conflicts that may occur. It is also necessary to have a complete knowledge of the grammar used to describe the input documents. In addition, this kind of human intervention would be unacceptable for a project like Envision, in which enormous quantities of data must be converted. It does not seem very convenient to have a person constantly monitoring the translation process and ready to give answers to the system. Manirak et al. admit that the number of times a user must make a choice can be excessive if the documents are long. In our opinion, a translation system should be able to make any decision by itself, even at the risk of making
CHAPTER 2. OVERVIEW OF RELATED WORK

an incorrect decision. It also should be robust, in the sense that it should be able to continue working appropriately, even after a wrong choice is taken. As opposed to this, Chameleon translators may abort if a user makes an incorrect choice.

For the reasons described above, we concluded that Chameleon was not appropriate as a solution to the problem addressed by this thesis. Chameleon can still be useful for solving simpler data conversion problems, as it relieves the user of having to write new translators each time a new format appears.

David Sklar [29] points out that up conversions from word-processor formats to an SGML document conforming to a useful DTD typically involve two phases: extraction and interpretation of formatting codes in the word processor format, and identification of content and structure. As an aid to the first stage, extraction of the word processor formatting codes, he developed Rainbow [30]. Rainbow is a publicly available SGML DTD. Together with Rainbow, a number of “Rainbow Makers” also are available. These makers convert documents in some word processor formats, such as RTF (Microsoft Word), into SGML documents conforming to the Rainbow DTD.

Rainbow insulates organizations that adopt it from the complexities of the word processor formats for which there are Rainbow Makers available. Once documents conforming to Rainbow are generated, the second phase, identification of content and structure, can be applied over those documents. This phase has to be performed by custom made tools developed by each organization. Sklar makes clear that although Rainbow is an SGML format, it is not appropriate for the permanent representation of data, because it contains no more structure than that found in the original word processor document, it only contains presentation oriented information.

A number of commercial tools are available to help the process of converting documents in different formats into SGML. We evaluated OmniMark [10, 11]. OmniMark is a language that can make the development of custom translators easier and faster. Still, a new translator must be developed for each particular translation problem, which means that it does not provide us the flexibility that we need. As opposed, within our framework we only have to
CHAPTER 2. OVERVIEW OF RELATED WORK

write a new format description for each translation problem, which should be a much easier process than implementing a new OmniMark translator. OmniMark is also rather cryptic and not very natural. We concluded that OmniMark would be a good choice if the domain of the conversions that are needed was perfectly known. If that was the situation, then we could write translators for each kind of conversion needed and then use those translators over and over again. In the case of Envision, we do not know the kinds of conversions that we may encounter in the future.

Other products are available, such as FastTag (copyright Avalanche Development Company). Due to lack of time and resources we were not able to evaluate these other products.

More recently a product called Fred has been made publicly available [28]. Fred incorporates a powerful scripting language that allows for the specification of arbitrary translations from tagged formats to any other formats. Even though it seems very appropriate for translating tagged data (like data in SGML) it does not address the problem of translating non-tagged documents.

A particularly interesting issue (vol. 38, no. 4) of the Communications of the ACM was devoted to the subject of Digital Libraries. In this edition an overview of many digital library projects is presented, including Envision [17]. Some of these overviews mention the problem of textual data conversion. In particular, Entlich et al. [9] have converted scanned pages of journals published by the American Chemical Society into SGML, though they do not mention the techniques used for this conversion. The overview of the Dienst project [22] also mentions the use of format conversion tools but does not describe them.

We do not believe that any of the available translation systems have the flexibility we require. Keith Shaffer [28] makes the following observation.

A few general translation tools are available, but most force users to map into a predefined DTD (which may be difficult or impossible to do) or do not offer sufficient options to meet the translation needs at OCLC.

We believe that the DELTO System is a general, flexible, comprehensive translation tool,
CHAPTER 2. OVERVIEW OF RELATED WORK

that can meet very complex translation needs.

2.2 Text Extraction

There are numerous research efforts that have concentrated on extracting different pieces of information from text databases for various purposes. Many of those efforts utilize the information extracted from text databases for very different purposes from ours. However, we believe that these research efforts are related to ours in the sense that extracting the information is a more difficult problem than utilizing it once it has been obtained.

Paul Jacobs has conducted many of those research efforts, for example one that led to the NLDB system [26]. This system categorizes news messages based on the identification of words, phrases, and more complex patterns within the analyzed messages. NLDB works on the basis of human-developed rules plus other rules that are automatically acquired through statistical methods. These statistical methods are applied over a large database of news messages and result in new, simple rules. Among the many interesting features of NLDB we can mention the more-than-one-pass processing of analyzed documents, which resembles that of the DELTO System. NLDB first pre-processes text in order to, among other things, identify proper names. For NLDB, it is extremely important to correctly identify company names. For this purpose it uses an algorithm devised by Lisa Rau [27]. An important difference between NLDB and DELTO is that this algorithm is embedded in NLDB, as opposed to the DELTO philosophy in which there are no specific recognition algorithms hardcoded, and everything is specified through a declarative language. The company algorithm developed by Rau looks for company names indicators, such as "Incorporated", "Corporation", or "International", and looks backwards for the first non-capitalized word. This heuristic fails on around 10 input document is all in uppercase. For this reason, the algorithm further accounts for many special cases that increase the algorithm ability to identify correct company names.

SCISOR, also developed by Paul Jacobs and Lisa Rau [21], processes stories about
CHAPTER 2. OVERVIEW OF RELATED WORK

company mergers and acquisitions and stores the information extracted from those stories, such as target, suitor, and price, in a database. Jacobs [19] describes methods that can be used for database generation from textual documents and that were actually used in the context of the SCISOR system. The processing utilized by Jacobs always includes some kind of parsing of the analyzed documents and the use of templates that are activated based on particular words or on words in a particular context.

Another text categorization system has been developed by Apté et al. [1]. In this case the system induces all the text classification rules that are used, as opposed to the systems developed by Jacobs and to another classical text categorization system [16], which are based on rules written by human experts. Apté's system has been tested on documents both in English and German. The authors of this system claim to have obtained very good results with both languages, which might suggest that their approach is language independent.

The CODER system [13], developed at Virginia Tech, analyzes heterogeneous documents, such as electronic mail messages. It was proposed as a way to investigate the use of knowledge-based approaches for handling the task of analysis and retrieval of composite documents. CODER is able to reason about the type and organization of messages and about the data elements that compose those messages, such as names, dates, and addresses. From the information extracted by this analysis, documents can be indexed and then retrieved. Different layout features are interpreted by CODER, such as justification, indentation, centering, capitalization, and use of punctuation marks. This interpretation helps CODER deduce the conceptual structure of a document. In addition, CODER identifies other features, such as names, titles, dates, times, and addresses, to obtain a better understanding of the structure and type of the analyzed messages. CODER utilizes a blackboard to combine the results obtained by different experts. Each expert module is able to recognize particular instances of the various types of document components (such as bibliographic entries). For example, there are experts that are able to recognize and interpret dates and times in many different formats. This is a bit reminiscent of the approach taken by DELTO. The difference
CHAPTER 2. OVERVIEW OF RELATED WORK

is that in DELTO the experts are created from high level specifications instead of being directly hard coded into the system.

The INQUERY system [2] views retrieval and routing of information as a probabilistic process of inference that compares text representations to representations of information needs (queries). INQUERY indexes documents using a variety of techniques, including the use of feature recognizers, written with the UNIX flex utility that can recognize objects such as company names, person names, names of countries, and names of US cities.

Conrad et al. [6] developed a system that automatically extracts features from text databases and constructs relationships (links) between those features based on statistical indications of the significance of those relationships. Features like company names and person names can be recognized. Each type of feature is identified by a special type of recognizer specifically designed for that kind of feature.

Huser et al. [18] describe a project that aims at the construction of hypermedia encyclopedias. This project uses automatic text processing techniques like pattern-oriented parsing and full text analysis. In particular, dictionary biographies are fully parsed. This is possible, though, because biographies are highly structured according to editorial guidelines. For the purpose of this project, those guidelines are encoded as rules of parsing and text-to-object conversion tools.

A system called COBATEF [26] identifies document structures, such as paragraphs, sentences, and titles. However, the rules for the extraction of those textual objects are hardcoded in the system, which means that users of the system cannot add their own rules or heuristics.

Hamill et al. [15] utilize statistical techniques to classify documents according to the 80 sections and five major section groupings of Chemical Abstracts. Only the words occurring within titles are used for the classification process. The system uses pattern matching techniques combined with heuristics.

Automatic thesaurus construction is another field that is based on text extraction systems. Virtually all automatic thesaurus generation systems are based on the statistical
CHAPTER 2. OVERVIEW OF RELATED WORK

co-occurrence of word types within textual documents [8, 5].

2.3 Summary

We could not find a system that solves the problem described in Section 1.3, and we believe that there does not exist one.

The biggest limitation of the conversion systems that we evaluated lies on the way that they view the process of translation: as simple mappings, insertions, and deletions of strings, which not always allow to restructure a document in the way that might be needed. There are problems also in the way that the conversions must be specified, that is, either with very inflexible descriptions of the input (i.e. context free grammars), or through complicated procedural instructions. In Chapter 3 we describe the architecture of the DELTO System, which we believe is able to solve the problem described in Section 1.3.
Chapter 3  
SYSTEM ARCHITECTURE

3.1 Overview

The DELTO System consists of a number of different components. In this chapter we describe each of these components and the relationships among them.

The DELTO System is composed of these three major components:

DELTO Compiler Reads a DELTO format description and generates a compiled version of it that includes instructions specifying how to process a document based on the compiled format description. This component is described in Section 3.3.

DELTO Analyzer Given a compiled format description and a text document, it generates an abstract representation of the document that includes information about objects that were identified using the information specified in the format description. The DELTO Analyzer is described in Section 3.4.

DELTO Translator Processes the abstract representation generated for a particular document by the DELTO Analyzer and generates a new document, based on the original one, but probably in another format or incorporating information that was not present in the original one. The translation process is specified through a Retarget specification. We describe the DELTO translator in Section 3.5.

The DELTO Analyzer is itself composed of these three components:

Lexical Analyzer Partitions a text document into a series of lexical tokens. A lexical token abstracts a small portion of the document. For example, the string "Virginia"
CHAPTER 3. SYSTEM ARCHITECTURE

may be abstracted both as a person name and as a state. Only one lexical analyzer is used for processing a single document. However, the DELTO Analyzer has various different lexical analyzers. Each lexical analyzer is designed to process a particular kind of input format. For example, there is a lexical analyzer for processing plain ASCII documents and another one for processing documents in SGML. Creating a new lexical analyzer and incorporating it into the DELTO Analyzer is a simple task. We describe lexical analyzers in Section 3.4.1.

Object Repository Generator Reads the output generated by the lexical analyzer for a particular document and generates an object repository for the document. An object repository is a database of objects extracted from a text document. It can be viewed as an abstract representation of the processed document. This component adds one object to the repository for each lexical token recognized by the lexical analyzer. We describe the object repository generator in Section 3.4.2.

Object Repository Analyzer Processes the object repository generated by the object repository generator using the information specified in a DELTO format description. It identifies new objects and stores each identified object into the object repository. Some of the identified objects (those that are relevant outside the scope of a single document and that can be used to construct hypertext links, such as persons, institutions, or journals) also are stored in the Envision permanent object database. Note that this is a permanent database with information about all documents processed by the DELTO System, as opposed to an object repository which is a temporary database that contains objects associated with only the document being processed. The object repository analyzer is described in Section 3.4.3.

The entire system makes use of the functionality provided by ODB, the database management system that we developed specifically for this project. We describe ODB in Section 3.2.

Figure 3.1 illustrates the high-level architecture of the DELTO System. Note that, for
CHAPTER 3. SYSTEM ARCHITECTURE

simplicity, the DELTO Analyzer is pictured as a single component in this figure, that is, the lexical analyzer, the object repository generator, and the object repository analyzer are all combined into a single component that represents the DELTO Analyzer. As can be seen in the Figure, the inputs of the system are a document in any format X, a DELTO format description for that format, and a Retarget specification for the output format Y. The format description for format X is compiled only once and then used for processing every document in that format. The system produces a new version of the input document in format Y and updates the permanent Envision object database with information extracted from the input document.

3.2 Object Database System (ODB)

The object database system (ODB) was developed with the idea of serving as a manager of the database of objects and links to be extracted from Envision documents. It not only fulfilled this role appropriately, but it also served other purposes within the architecture of the DELTO System.

ODB is a very simple and flexible extended relational database system. It incorporates features that are not part of the relational model and that substantially increase its power. Traditional relational databases are collections of tables. Each table is composed of records, where each record has a fixed number of fields that can be assigned specific types of values. Records are often of fixed length. In ODB, we define classes. Classes also are defined in terms of fields and every field is associated with a particular type, in the same way as records in Pascal or structures in C. An object of a particular class is only a collection of values for each of the class fields. However, ODB fields can be assigned not just one value but any number of values, according to the way the fields are specified within the class definition. In this way, ODB fields resemble variable size arrays. As a consequence, ODB objects do not have a fixed length.

The other major difference between ODB and the relational model lies in the types
Figure 3.1: High level architecture of the DELTO System
CHAPTER 3. SYSTEM ARCHITECTURE

of values that a field may be assigned. ODB incorporates \textit{links} as a possible field type. ODB links always connect two different objects and two specific fields within the linked objects. All ODB links are automatically bidirectional. For example, we could establish a link between the \texttt{author} field of an ODB object of the class \texttt{document} and the \texttt{author.of}\ field of a \texttt{person} ODB object. Four parameters are required to establish a link: one object and a field within it together with another object and a field within it. If a link is deleted from one object, it is automatically deleted from the other object. If an object is deleted, all the links to it also are removed by the system.

The other possible field types for ODB objects are \texttt{string}, \texttt{date}, and \texttt{object-id}, though in the current implementation, for simplicity, \texttt{date} and \texttt{object-id} fields are treated in the same way as those of type \texttt{string}. Each ODB object has a unique ID with respect to all the other objects in the same database. ODB also supports indexing of objects on arbitrary strings, which allows object retrieval to be performed not only on the basis of an object ID but also on arbitrary strings. For example, a person object could be indexed on the name of the person that it represents. This object is then accessible both through its ID and through the name of the person.

It should be noted that, even though our terminology refers to this system as the \texttt{object database system}, it is not an object-oriented system. As mentioned before, it is an extended relational database management system. In particular, it does not support inheritance, a typical characteristic of object-oriented database systems.

We developed a complete library of C functions that can be invoked by any program that needs access to ODB databases. In addition, we created a simple text interface to ODB that allows users to create databases; to add, update and delete classes and objects; and to index objects; among other possibilities. This interface is described in Appendix A. A World-Wide Web presentation server was developed by Scott Guyer for Envision. This server allows objects from any ODB database, and in particular from the Envision permanent ODB database, to be viewed through the World-Wide Web with the possibility of following links just by clicking on anchors.

20
CHAPTER 3. SYSTEM ARCHITECTURE

In Figure 3.2, we present two possible ODB class definitions, as depicted by the ODB text interface. Note the specification of the number of values that can be assigned to each of the defined fields. The 0..1 specifier indicates that one value may be assigned to the defined field, and that it also is valid not to assign any value to it. The 0..N range indicates that any number of values may be assigned to the field, although it is still valid not to assign any. We specify 1..N if we want to enforce that at least one value is assigned to a field for an object to be valid. Any combination of numbers and the symbol N can be used to specify a range (as long as the lower bound is less than or equal to the upper bound). The ODB System checks that each object satisfies the constraints imposed by these kinds of definitions before an object is stored into a database.

Two objects, one for each of the classes defined in Figure 3.2, are presented in Figure 3.3. In this Figure, a field value such as 185->author represents a link to the author field of the object whose ID is 185. Note the existence of an author <-> author_of link between the two objects displayed in the Figure.

ODB was developed using the gdbm library of database management functions, developed by the Free Software Foundation.

3.3 DELTO Compiler

The DELTO Compiler reads in format descriptions, analyzes them, and generates a target representation of the descriptions that is used by the DELTO Analyzer. This is a two pass compiler, due to the fact that DELTO classes can be referenced before they are defined. A description of how to invoke the compiler that we have developed can be found in Appendix A.

The compiler checks the syntactic correctness of the input descriptions and attempts to continue its processing after each error whenever possible. It also performs type checking to make sure that the DELTO rules of type compatibility are enforced by the compiled descriptions. DELTO is a strongly typed language, and the compiler enforces this characteristic.
### CHAPTER 3. SYSTEM ARCHITECTURE

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th># Values</th>
<th>Internal ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>STRING</td>
<td>0..1</td>
<td>0</td>
</tr>
<tr>
<td>item_type</td>
<td>STRING</td>
<td>0..1</td>
<td>1</td>
</tr>
<tr>
<td>author</td>
<td>LINK</td>
<td>0..N</td>
<td>2</td>
</tr>
<tr>
<td>institution</td>
<td>STRING</td>
<td>0..N</td>
<td>3</td>
</tr>
<tr>
<td>published_in</td>
<td>LINK</td>
<td>0..N</td>
<td>4</td>
</tr>
<tr>
<td>edition</td>
<td>STRING</td>
<td>0..1</td>
<td>5</td>
</tr>
<tr>
<td>series</td>
<td>STRING</td>
<td>0..1</td>
<td>6</td>
</tr>
<tr>
<td>date</td>
<td>STRING</td>
<td>0..1</td>
<td>7</td>
</tr>
<tr>
<td>volume</td>
<td>STRING</td>
<td>0..1</td>
<td>8</td>
</tr>
<tr>
<td>number</td>
<td>STRING</td>
<td>0..1</td>
<td>9</td>
</tr>
<tr>
<td>chapter</td>
<td>STRING</td>
<td>0..1</td>
<td>10</td>
</tr>
<tr>
<td>pp</td>
<td>STRING</td>
<td>0..1</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th># Values</th>
<th>Internal ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>STRING</td>
<td>0..1</td>
<td>0</td>
</tr>
<tr>
<td>also_known_as</td>
<td>STRING</td>
<td>0..N</td>
<td>1</td>
</tr>
<tr>
<td>institution</td>
<td>LINK</td>
<td>0..N</td>
<td>2</td>
</tr>
<tr>
<td>address</td>
<td>STRING</td>
<td>0..N</td>
<td>3</td>
</tr>
<tr>
<td>phone</td>
<td>STRING</td>
<td>0..N</td>
<td>4</td>
</tr>
<tr>
<td>email</td>
<td>STRING</td>
<td>0..N</td>
<td>5</td>
</tr>
<tr>
<td>rsrch_interests</td>
<td>STRING</td>
<td>0..N</td>
<td>6</td>
</tr>
<tr>
<td>biography</td>
<td>STRING</td>
<td>0..N</td>
<td>7</td>
</tr>
<tr>
<td>author_of</td>
<td>LINK</td>
<td>0..N</td>
<td>8</td>
</tr>
<tr>
<td>current_as_of</td>
<td>DATE</td>
<td>0..1</td>
<td>9</td>
</tr>
<tr>
<td>editor_of</td>
<td>LINK</td>
<td>0..N</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 3.2: ODB class definitions for document and person objects
CHAPTER 3. SYSTEM ARCHITECTURE

Object ID: 185
Object Class ID: dc
Stored in Object File #: 1

title[1] = Learning and Remembering Command Names
item_type[1] = InProc
author[1] = ia6->author_of
author[2] = ia7->author_of
published_in[1] = 1d7->document
series[1] = Naming Commands
date[1] = 1982
pp[1] = 8 11

Object ID: 1a6
Object Class ID: pr
Stored in Object File #: 1

name[1] = Black, John B.
author_of[1] = 185->author
author_of[2] = 1e8->author
author_of[3] = 1eg->author
author_of[4] = 1el->author
author_of[5] = 1kp->author
author_of[6] = 1oj->author

Figure 3.3: ODB objects
CHAPTER 3. SYSTEM ARCHITECTURE

of the language.

The output that the compiler generates is an ODB database that exactly mirrors the data structures that are generated as a result of parsing and processing an input description. Pointers within those data structures are represented by links in the database. This is not the most efficient way to save the results generated by the compiler, either in time or space. However, it is a very convenient way to do so, in the sense that the exact representation of the generated data structures is saved into the object database and reconstructed by the DELTO Analyzer with minimum implementation effort. The other alternative would have been to design a format to store the output generated by the compiler. This would have resulted more efficient, but it would have also required more time to complete the implementation. Using ODB to save the results produced by the compiler also helped us examine the generated data structures because we can easily traverse their ODB representations using the text interface to the ODB system. It also helped us test the robustness of the ODB system for dealing with large amounts of data. The text interface to the ODB system is described in Appendix A.

The compiler also generates a listing file illustrating the generated data structures in a human-readable form. This file was only used for testing purposes, but we did not eliminate its generation because the information that this file contains may be of future interest. This is just an ASCII file, as opposed to the main output generated by the compiler in the form of an ODB database.

There is a syntax-check-only option that makes the compiler only check the syntax of the given format description and generate the listing file, though it does not create the target format description, in the form of an ODB database, needed by the DELTO Analyzer.

3.4 DELTO Analyzer

The DELTO Analyzer, as mentioned before, is composed of three components:

- Lexical Analyzer;
CHAPTER 3. SYSTEM ARCHITECTURE

Figure 3.4: Architecture of the DELTO Analyzer

- Object Repository Generator; and
- Object Repository Analyzer.

In this section we describe each of these components. A diagram showing the pieces of the DELTO Analyzer and the relationships among them is presented in Figure 3.4.
CHAPTER 3. SYSTEM ARCHITECTURE

3.4.1 Lexical Analyzer

A lexical analyzer reads in a document and breaks its contents into a set of tokens. Each token abstracts a certain piece of text of the input document, and a given piece of text may be abstracted by more than one token. For example, the string "Virginia" may be abstracted both as a person name and as the name of a state. DELTO lexical analyzers also are able to abstract pieces of text composed of more than one word. For example, the string "Massachusetts Institute of Technology" may be abstracted as a university name. The generated tokens may also contain a number of attributes, such as the position in the document in which they occur. A word token may, for example, contain an attribute that indicates if the word is capitalized or not. The values of token attributes may be referenced at later stages of the DELTO Analyzer (see discussion about attributes in Chapter 4).

As mentioned in Section 3.1, there are a number of lexical analyzers that can be used by the DELTO System, each designed for a particular document format. Each DELTO lexical analyzer is a lex program. We have developed a basic skeleton for all DELTO Analyzers, together with a library of lexical analyzer C functions. This makes creating a new lexical analyzer a very simple task, where the only thing that needs to be determined is the lex specification to identify specific patterns and, possibly, one or more functions that generate the tokens associated with those patterns. Such functions should be very similar to the ones that are already defined to process other kinds of tokens. A description of how to invoke the lexical analyzers that we have developed can be found in Appendix A.

There also is a dictionary system we developed that is used in combination with the lexical analyzers. We use the dictionary to associate words or word sequences with semantic categories, such as person names, names of cities, and names of institutions. The dictionary also could have been used to store syntactic categories of words, such as nouns or verbs, but we decided that this is not necessary to process Envision documents. Such capability can be added if at some point in time we determine that it will improve the quality of document
CHAPTER 3. SYSTEM ARCHITECTURE

processing that we can achieve. Our dictionary system is not restricted to a fixed set of categories. A new category can be added to classify a word or word sequence at any time. We present a description of this dictionary system in Appendix A.

Each time a word is identified by a lexical analyzer, a word token is generated. In addition, a dictionary is checked. If one or more entries are found for the word being processed, additional tokens are generated to represent the categories associated with the entries encountered in the dictionary. From the contents of a dictionary the lexical analyzer is able to identify pieces of text as person names, or country names, or any other kind of semantic category. Suppose the lexical analyzer encounters the sequence of words "Virginia Tech". The lexical analyzer first looks up the word "Virginia". The dictionary may tell the lexical analyzer that the word represents both a person name and a state, and that in addition there is a relevant sequence of two words that starts with that word. Given this information the lexical analyzer looks ahead for the next word and now passes the two word sequence to the dictionary. The dictionary checks whether the sequence encountered by the lexical analyzer matches the one that it contains. Continuing with our hypothetical situation, if the sequence "Virginia Tech" is encountered by a lexical analyzer, there would be three tokens generated for "Virginia" (word, person.name, and US_state), one token for "Virginia Tech" (university.name) and also one token for "Tech" (word). The dictionary may contain an entry for the word "Tech" categorizing it as a university_indicator, in which case an additional token of that category would be returned by the lexical analyzer.

A lexical analyzer uses only one dictionary when processing a particular document. However, it may use different dictionaries for processing different kinds of documents.

We also used gdbm to develop the dictionary. We could have directly used ODB for this purpose, but we developed the dictionary before we developed the ODB system. We loaded our dictionaries with person names, surnames, university names, and other interesting categories, mostly from large lists available on the Internet. It was necessary to prune most of the lists gathered from the Internet because some of the words that were categorized as person names were not very appropriate, for example “development”, or “technology".
CHAPTER 3. SYSTEM ARCHITECTURE

<table>
<thead>
<tr>
<th>Category</th>
<th>Number of Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given Names</td>
<td>25,617</td>
</tr>
<tr>
<td>Last Names</td>
<td>14,214</td>
</tr>
<tr>
<td>University Names</td>
<td>706</td>
</tr>
<tr>
<td>Company Names</td>
<td>34</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>40,571</strong></td>
</tr>
</tbody>
</table>

Table 3.1: Dictionary statistics

We also keyed in many additional entries. Table 3.1 summarizes the contents that our dictionary currently has. However, the dictionary is not a static entity and more entries are constantly added to it.

Each time a lexical analyzer is executed it updates a file of *suggestions*, that is, words that were not found in the dictionary but that the lexical analyzer suggests as possible candidates to be added to it. We use a simple heuristic for generating these suggestions: they are capitalized words that do not start sentences. The dictionary system can then process this suggestion file automatically, prompting the user, probably the database administrator, for each word that has been suggested. The user has the choice of adding the word to the dictionary and specifying the categories that the word is associated to, or skipping the word if it is not reasonable to enter it into the dictionary. This process can be performed after one or more documents are processed with the DELTO system.

The DELTO dictionary system also stores abbreviations and their associated unabbreviated meanings. As a result of this, it is not necessary to define "VA", for example, as the name of a state but simply as an abbreviation for "Virginia". If a lexical analyzer finds the string "VA Tech" it recognizes that it is a university name, given that "VA" has been entered as an abbreviation for "Virginia", and that "Virginia Tech" has been entered as a university name.

The output generated by the lexical analyzers is in the form of LISP-like balanced strings. The reason for this is that originally we intended to develop the rest of the DELTO Analyzer using LISP, though this decision was changed later on. The lexical analyzers themselves are, as we already mentioned, written in lex and the supporting functions in C.
CHAPTER 3. SYSTEM ARCHITECTURE

Figure 3.5: Output generated by the SGML lexical analyzer

Figure 3.5 gives an example of the output that the SGML lexical analyzer would generate for the string "<institution>Virginia Tech</institution>", assuming that the used dictionary is loaded with the appropriate entries. The output of a lexical analyzer is a sequence of lists. Each of these lists contains one or more lexical tokens for a word or for one or more word sequences that start with that word. (Note the lexical tokens for "Virginia" and for "Virginia Tech".) Each lexical token is also a list whose first element indicates the category of the token. Additional sublists may be included within a lexical token. These lists represent attribute-value pairs. For example, an attribute that is generated for every lexical token is pos, which indicates the byte offset with respect to the origin of the document where the portion of text represented by the token starts.
CHAPTER 3. SYSTEM ARCHITECTURE

3.4.2 Object Repository Generator

The results generated by the lexical analyzer are still flat text files that can only be sequentially scanned (the operation most translation tools perform).

The Object Repository Generator uses the document token list generated by a lexical analyzer to create an object repository. This document object repository is a database that represents the contents of the analyzed document. Each token produced by the lexical analyzer becomes an object in this database. The text of the entire document also is stored in this database, which means that the original document can be reconstructed from this repository.

The power behind this object repository is that it is no longer a flat, one dimensional, representation of a document. The position of an object within the analyzed document is not given by the physical organization of the repository but by two attributes that indicate the beginning and ending positions of the object (pos and endpos). For this reason we can have objects that overlap other objects, which provides a multi-dimensional view of the analyzed document. Each object is indexed on its starting and ending positions. For example, all the objects that represent pieces of text that start at position 17414 are indexed by the string "@[17414]". (It is guaranteed that only position index strings can start with the "@" symbol.) All the objects that represent pieces of string that end at position 17771 are indexed by the string ">@17771". In this way we can conveniently access objects located at any position within the analyzed document.

Objects may also be indexed on their class. For example, all objects of class person.name could be indexed on a string such as "#person.name" (where the "#" symbol is guaranteed to occur only on class index strings). We can then retrieve all the person.name objects recognized within a document in one single operation, as opposed to having to scan the whole document to look for them.

Each piece of text that is represented by a token may be indexed by the piece of text itself. For example, all the occurrences of the string "university" within a document may
CHAPTER 3. SYSTEM ARCHITECTURE

be indexed on the string "university". We can then retrieve all the occurrences of that string within the analyzed document in one operation.

We mentioned that objects may be indexed on their classes. The reason why this kind of indexing is optional is because not all classes of objects need to be indexed for a particular analysis task and because it would be a waste of time and space to index all the objects that do not need to be indexed. For example, it is very likely that there are going to be more objects of class word than of any other class within the object repository of a document. If certain kinds of accesses to those objects are not needed (the kinds of accesses that require an index), then there is no need to index all words on their class (though they are still indexed on their beginning and ending positions). The same concept applies for indexing pieces of text. For example, it may not be necessary to index all occurrences of the word "the" on that word itself. There may be numerous occurrences of that word and they may not be accessed through the index at any time. On the contrary, it may be very useful to index all the occurrences of the word "university" as this word can usually help the object repository analyzer (described in Section 3.4.3) identify university names that have not been recognized by the lexical analyzer.

The object repository generator takes the output generated by a lexical analyzer and creates an object repository for it. It allows different ways to specify which classes and which pieces of text should be indexed. However, the information about which classes and strings have to be indexed is generated by the DELTO Compiler for each particular format description. Therefore, the only thing that is needed is to give the appropriate files, generated by the DELTO Compiler, to the repository generator, depending on which format description is being used to process the document whose repository is about to be constructed. We present a description of how to invoke the object repository generator in Appendix A.

Objects repositories also are supported by our ODB system. This, again, simplified the programming of the object repository library and also simplified the testing of the functions in this library, including the repository creation process, as the contents of object
CHAPTER 3. SYSTEM ARCHITECTURE

repositories are browsable using the ODB text interface. The object repository library has been developed in C, while the repository generator is a yacc program that parses the LISP-like output produced by the lexical analyzers and invokes the functions defined in the object repository library in order to create and populate a new repository.

3.4.3 Object Repository Analyzer

The object repository analyzer is the most important piece of the DELTO Analyzer and the most complex software component developed for this thesis. Appendix A provides a description of the invocation procedures for the object repository analyzer.

The repository analyzer takes as input a target DELTO format description (generated by the DELTO Compiler) and an object repository generated by the object repository generator. The repository analyzer is the component that actually executes the actions implied by a DELTO format description. Format descriptions, which are described in chapter 4, do not concretely specify actions but rather describe objects. The compiler interprets those descriptions and generates the appropriate actions for the repository analyzer to execute in order to recognize objects. As a very simple example, a sequence of a given name and a surname identified by the lexical analyzer (and therefore represented by two objects in the object repository) may result in a new person name object that encompasses that sequence of objects.

The object repository analyzer repeatedly examines the contents of the repository and generates new objects in terms of the ones that previously existed. The new objects are added to the repository, and they may be used again to generate new, higher-level objects. For all practical purposes, the repository analyzer does not distinguish between objects created by itself and those created by the repository generator. The nature of the object repository and the nature of operations that can be applied to it facilitates immensely the tasks of object recognition and creation. That is, the numerous non-sequential ways to access repository objects via indexes makes this task possible without having to scan the whole document each time, as would be necessary with a flat representation of the
CHAPTER 3. SYSTEM ARCHITECTURE

processed documents. In addition, adding new objects anywhere in the document is an extremely simple task, due to the fact that what determines the position of an object within the analyzed document is the value of the pos and endpos attributes as opposed to the physical place in which the object is stored. Adding objects would be very complicated if we used a flat text representation of the document.

The object repository analyzer also is responsible for updating the permanent database of objects extracted from documents (the database for which ODB was originally developed). For some objects identified within the processed documents, representations are stored in this permanent database. DELTO format descriptions specify which objects are to be stored in this database and what information is to be stored for each class of objects. For example, an object of class person is typically stored in the permanent database and is linked to the documents that the person is an author of. The same person may be the author of more than one document, which means that the object repository analyzer is likely to encounter the same names over and over again. In such a situation it is desirable not to create a new representation for the person in the permanent database but to update the one that already exists. The object repository analyzer performs this task, based on information provided on the DELTO format descriptions for this purpose. This information should be considered a set of heuristics. The better and the more complete these heuristics are, the more likely the repository analyzer will be able to correctly identify previously existing objects in the permanent database. Chapter 4, which is devoted to describing the DELTO language, explains how this information can be given within a format description.

The object repository analyzer generates a log of all operations applied to the permanent database so that some or all of those operations can be undone if necessary. In particular, the log indicates which objects have been created and which previously existing objects were selected for update. For the latter objects, the log also includes the weight on which the repository analyzer based its decision to select them instead of creating new ones. The weight, in this context, is an integer between 0 and 100 that expresses the confidence that the system has that the object has been correctly selected. This value is the result
CHAPTER 3. SYSTEM ARCHITECTURE

of evaluating the heuristics that we mentioned in the previous paragraph. Currently, the object repository analyzer only selects for update objects that obtain a weight above 70.

No parsing of the input document is performed by the repository analyzer. The most common operation that is performed is pattern matching. However, there are many different ways to specify pattern matching through format descriptions. Patterns can match both specified strings (as most text extraction systems allow) and also objects that may have to satisfy specific properties. Approximate string matching is also available. We implemented approximate string matching using a modified version (two level approximate matching: characters and words as opposed to characters only) of an algorithm developed by Sun Wu and Udi Mamber [32, 33]. In addition, not all the given patterns have to be satisfied for an object to be considered of a specific class. All the information is given through format descriptions in a declarative way.

The object repository analyzer is robust: failure to recognize objects does not make it stop or abort. It continues trying to accomplish the best possible job. Every identified object is assigned a numeric weight that expresses the confidence that the analyzer has that the object has been correctly identified (note that this is a different kind of weight than the one assigned to objects in the permanent database that are candidates for being updated).

3.5 DELTO Translator

The object repository analyzer produces a repository containing a great deal of information. Some of that information may only be needed to update the permanent object database. Other parts of that information may be needed to generate a new document, equivalent to the original one but in another format, or incorporating more information than the original document had.

All the information in the object repository, including the entire text of the input document and the objects that represent different pieces of that document, is accessible during the final process of the DELTO System: the translation.
CHAPTER 3. SYSTEM ARCHITECTURE

For this purpose, we developed a language that we called Retarget. A description of this language is given in Chapter 5 of this thesis. Retarget is able to concisely specify what the output document should look like, what information from the object repository has to be included in the output document, and in what order.

A big difference between our translation process and any previous translation systems lies in the input that we use, the object repository. As we already mentioned, the object repository is not a flat file; it is a complete database that represents the original document, with different access paths to each of its component objects. Most conversion systems sequentially scan an input file and sequentially map strings in the input into the output. On the contrary, our system is able to access the contents of the object repository in any way, possibly accessing the same pieces of information more than once, and ignoring others. This means that a complete restructuring of the document is possible; the output document does not have to follow the structure of the input one.

Retarget allows access to everything contained in an object repository: text and objects. It allows for the convenient specification of how to use that information in the construction of a new document. It is a very simple and natural language. It specifies a template for the output document and how to fill in the blanks left in that template with information originated from the object repository.

Our translator is implemented as a yacc program, that parses a Retarget specification each time a translation is done (the parsing of Retarget is so simple that we decided that it was not worthwhile to have a separate compiler for Retarget specifications). It uses the result of that parsing to generate an output document, given the information contained in the input object repository.

It is possible to give the translator a table of string mappings to be used when generating the output document. This table is not used to map strings in the input into strings in the output, but only in combination with strings already generated for the output. Before anything is written into the output, it is checked against this table (if the table is given to the translator). If an entry for the string being written exists in the table, then the associated
CHAPTER 3. SYSTEM ARCHITECTURE

string is written to the output, not the one that was originally going to be produced. This can be useful in situations in which it may be necessary to escape certain characters in the output format. For example, we may want to produce the sequence \& each time the & character is written into the output, or we may want to generate SGML entities in place of the characters that have a special meaning for SGML. This table isolates the translation process, and the Retarget specifications, from these complexities derived from the output format. Characters with special meaning in the output format can still be generated by the translation process, and only before they are written into the output they are mapped into their correct output format strings based on the contents of this table. This table is a text file whose name may be given to the translator as a command line argument. The format that this table should have, together with the invocation procedures of the translator are described in Appendix A.8 of this thesis.
Chapter 4

DELTO: LANGUAGE REFERENCE

4.1 Introduction

In this chapter we describe the syntax and the semantics of DELTO. We introduce the syntax of the language piece by piece, as needed to explain the semantics. A complete syntactic grammar for DELTO is presented in Appendix F. In Appendix C we introduce three complete examples of DELTO descriptions.

4.2 The Syntax of DELTO

We use Extended Backus-Naur Form (EBNF) [12] as the formalism to introduce the syntax of DELTO. The conventions that have been adopted to present the syntax of DELTO are depicted in Figure 4.1, where string represents an arbitrary sequence of nonterminals and terminals.

Some EBNF symbols are also terminals for DELTO. Whenever one of these symbols appears in the grammar as a terminal it is enclosed in single quotes, as in '1'. These EBNF symbols are the following: '[', ']', '{', '}', '(', ')', '|', '<', and '>'. No quotes enclose them when they are used as EBNF symbols.

DELTO is case insensitive. Separators (spaces, tabs, and line breaks) may be used freely throughout a format description. They do not affect the meaning of a description. They are only relevant inside string constants.

Identifiers, non-negative integer constants and string constants are treated as terminals in the grammar. Non-negative integer constants are the only kind of numeric constants allowed in DELTO.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>terminal</td>
<td>The terminal string <em>terminal</em></td>
</tr>
<tr>
<td>&lt;nonterminal&gt;</td>
<td>The nonterminal named <em>nonterminal</em></td>
</tr>
<tr>
<td>[string]</td>
<td>The inclusion of string is optional</td>
</tr>
<tr>
<td>{string}</td>
<td>Zero or more occurrences of string</td>
</tr>
<tr>
<td>(string)</td>
<td>Grouping</td>
</tr>
<tr>
<td>string1</td>
<td>string2</td>
</tr>
<tr>
<td>&lt;nonterminal&gt; ::= string</td>
<td>The <em>nonterminal</em> is defined by string</td>
</tr>
</tbody>
</table>

Figure 4.1: EBNF conventions used to present the grammar of DELTO

Identifiers are represented by the terminal *ident* in the grammar. An identifier must begin with a letter and can continue with letters, digits, underscores, or dashes. There is no maximum length, which means that all characters that form an identifier are significant.

Non-negative integer constants are represented by the terminal *integer*. Such a constant is only composed of digits.

String constants are represented by the terminal *literal*. They consist of an arbitrary sequence of characters enclosed either by double quotes or by single quotes. If a single quote is part of the string, then the enclosing symbols should be double quotes, and vice versa.

Constant identifiers are represented by the terminal *const_ident*. They must begin with the '$' symbol and can continue with letters, digits, underscores, or dashes. They must have, at least, one character following the '$', and there is no maximum length.

Throughout the grammar all the terminals except *ident*, *integer*, *literal*, and *const_ident* are written in uppercase though, as mentioned before, DELTO is case insensitive, which means that terminals can be written in any way. We present them in uppercase in the grammar to improve its readability.

It is not explicitly indicated where constant identifiers may be used (to simplify the grammar). The rule is simple: wherever the terminal *integer* or *literal* appears in the grammar, it may be replaced by the terminal *const_ident*.

Comments may be included anywhere in a format description. They have to be between the markers /* and */; in the same way as in the C programming language.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

\[
\begin{align*}
\text{<frmt\_description>} & \quad ::= \quad \text{[USE} : \text{literal} \{, \text{literal}\};]\text{[LEX} : \text{literal};]\text{[UNDEFINE} : \text{<class\_name>} \{, \text{<class\_name}> \}];\text{[CONSTANTS}: \text{const\_ident} (\text{integer} \mid \text{literal}) \quad \text{\{.const\_ident} (\text{integer} \mid \text{literal}) \}];\text{[<class\_def}> \{; \text{<class\_def>}'
\end{align*}
\]

\text{ENDFMTD}

\text{<class\_name>} \quad ::= \quad \text{ident}

Figure 4.2: Syntax rules for format descriptions

4.3 Format Descriptions

Format descriptions are the highest level DELTO entities. A format description is always contained in a file, and a file can only contain one format description. A format description for DELTO is the equivalent of a program for Pascal, for example. Format descriptions are composed of class definitions. A format description may use other format descriptions. All the class definitions included in the used format descriptions are automatically inherited by the format description that uses them. From now on we shall refer to used format descriptions as general format descriptions or simply general descriptions, and to format descriptions that use other descriptions as specific format descriptions or specific descriptions. It should be noted that a particular format description can be general and specific at the same time if it uses other format descriptions and it is itself used by one or more format descriptions. General and specific are, therefore, relative concepts. A format description can be general with respect to another format description and specific with respect to a different one. General and specific descriptions allow the construction of hierarchies of format descriptions.

Specific format descriptions can redefine, undefine, or modify class definitions inherited from their general format descriptions.

The rules that define the syntax of format descriptions are presented in Figure 4.2.

The keyword USE indicates what the general descriptions of a specific format description
CHAPTER 4. DELTO: LANGUAGE REFERENCE

are. If utilized, it must be the first piece of information included in a format description (with the exception of comments). For example, a format description that uses three other format descriptions may begin in the following way:

USE: "description_a", "description_b", "description_c";

Different formats may require different lexical analyzers. A format description may indicate which lexical analyzer is to be used during the processing of documents in the described format. If no lexical analyzer is indicated in a format description, then a default one is used by the DELTO System. The keyword LEX may be used to indicate which lexical analyzer to use. We can complete the previously initiated format description as follows:

USE: "description_a", "description_b", "description_c";
LEX: "lx.sgml";

Another keyword that may be used in a format description is UNDEFINE. It undefines one or more class definitions defined in general format descriptions. An undefined class definition becomes invisible in the entire format description. Assuming that a class named Class_1 is defined in the format description "description_b", Class_1 is automatically defined in the first of the two following format descriptions, but a definition for it does not exist in the second one:

USE: "description_a", "description_b", "description_c";
LEX: "lx.sgml";
ENDFMTD;

USE: "description_a", "description_b", "description_c";
LEX: "lx.sgml";
UNDEFINE: Class_1;
ENDFMTD;

Constant identifiers may be defined within a format description. These are just a convenience so that strings or numeric constants that are used several times can be represented by a constant identifier. Constant identifiers defined in general format descriptions are inherited by specific format descriptions, although specific descriptions may change the values that inherited constants represent. An example of a constant identifier definition is:

40
CHAPTER 4. DELTO: LANGUAGE REFERENCE

CONSTANTS: $String_Const "Hello", $Numeric_Const 5;

4.4 Class Definitions

A class definition may describe what the textual objects that are instances of the class look like in the document format being described, what the internal structure is of the instance objects, and how that structure is constructed. A class definition may also indicate how to update a permanent object database with information originated from objects of the defined class.

All classes must have a name, and there must always be a class named document. Duplicate definitions for the same class are not allowed. A class name can be any valid identifier.

From now on we name general class definitions or simply general definitions those class definitions that are part of a general format description, and specific class definitions or specific definitions those class definitions that are part of a specific format description. There may be both a general definition and a specific definition for the same class, or just one kind of definition for a class. There are three cases to consider:

- There is a general definition for class A, but no specific definition for it.
- There are general and specific definitions for class A.
- There is a specific definition for class A, but no general definition for it.

In the first case the specific format description inherits the general class definition. This class definition can be undefined, in the way that has already been described in Section 4.3 of this chapter.

In the second case, the two definitions are combined. All component and attribute definitions, expressions, and object database interfaces are merged together. (All of these are described in Sections 4.4.1, 4.4.2, 4.4.3 and 4.4.11 of this chapter.) Expressions defined in the general definition may be undefined by the specific definition. Expressions may be labeled
CHAPTER 4. DELTO: LANGUAGE REFERENCE

<class_def> ::= CLASS ( DEFINITION | REDEFINITION ) : <class_name>
               [PERSISTENT INSTANCES]
               [UNDEFINE <label> {, <label> }]
               [ <attrs_block> ]
               [ <components_block> ]
               [ <odb_interface_block> ]
               [ <description_block> ]
               { <siring_comp_descr> }
ENDDEF

<class_name> ::= ident

<label> ::= ident

Figure 4.3: Syntax rules for class definitions

with an identifier (the scope of each label is limited to a class definition). Labeled expressions
defined within general class definitions may be undefined by specific class definitions. For
example, if the general class definition A contains the expressions labeled E1 and E2, a specific
class definition for A may undefine expression E1 while retaining expression E2. Note that
unlabeled expressions occurring within a general class definition cannot be undefined in a
specific definition for the same class.

The third case is straightforward. The class definition is only in the specific description,
which means that it does not have to be combined with any other definition.

The syntactical rules for class definitions are given in Figure 4.3.

If the keyword REDEFINITION is used within a specific format description, then any
general definition available for the class being defined is ignored (this is the way to redefine
a general definition). If the DEFINITION keyword is used in a specific format description,
then if a general definition for the class is available, this definition is inherited by the class
being defined (this is the way to modify a general definition). Note that the REDEFINITION
keyword can be used only within a specific format description for the purpose of redefining
a general class definition. On the other hand, the DEFINITION keyword can be used within
either a general or a specific format description.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

The objects that need to be saved into the permanent ODB database are called persistent objects. If the instances of the class being defined are persistent, then the keywords PERSISTENT INSTANCES must be included. A list of expression labels may follow the keyword UNDEFINE. Any expressions defined in a general definition and marked with any of these labels disappear from the resulting class definition.

After this class definition header, the most important parts follow attributes block, components block, object database interface block, description block, and string component descriptions. The concepts behind these syntactic constructions are all described in this chapter. Section 4.4.1 describes attributes and attributes blocks, while Section 4.4.2 explains components and components blocks. Numerous sections are devoted to the different parts that may compose a description block: Section 4.4.3 defines expressions and predicates, Section 4.4.4 explains each of the available DELTO predicates, Section 4.4.5 describes how to reference attributes and components, Section 4.4.6 describes each of the DELTO functions, Section 4.4.7 introduces DELTO regular expressions, Section 4.4.8 describes how to combine various pieces into a description block, and Section 4.4.9 talks about non-textual objects. String component descriptions are presented in Section 4.4.10 and Section 4.4.11 describes object database interface blocks.

4.4.1 Attributes

Each object has a series of attributes that describes properties of the object and relationships between the object and the document in which it occurs. Some of those attributes are automatically defined for every class of objects that is defined, while others have to be defined within class definitions. Attributes may be bound to values, that is, they are associated with values.

The following is a list of automatically defined attributes:

ID The DELTO System binds this attribute to a unique value (with respect to the values assigned to the ID attribute of all the other objects identified in the same document)
CHAPTER 4. DELTO: LANGUAGE REFERENCE

for each object that it finds. By binding the ID of an object to an attribute of another object, links within the same document can be established.

POS The starting position (byte offset) of the character string represented by the object, relative to the beginning of the document. The first character in the document has position 0.

ENDPOS The ending position (byte offset) of the character string abstracted by the object, relative to the beginning of the document.

HPOS The starting horizontal position (byte offset) of the character string abstracted by the object, relative to the beginning of the line where it occurs. As opposed to the way that the values for attributes POS and ENDPOS are computed (each character counts always as 1), tab characters occurring as separators make this value be incremented to the next multiple of 8. For example, an object at the beginning of a line has this attribute bound to 0, while if it is preceded by one tab character the value that is bound to this attribute is 8.

LHPOS The logical horizontal position. It is the same as HPOS, but objects that represent markup (an SGML tag, for example) are ignored for the purpose of computing the values that are bound to the LHPOS attributes of all objects in the same line.

LENGTH The length of the character string that the object abstracts.

Another kind of automatically defined attributes is a statistical attribute. One statistical attribute for each class defined in a format description is automatically defined for each object that is recognized by the DELTO System. These attributes are named by concatenating a pound symbol ('#') to the name of a class. Each of these attributes is bound to a non-negative integer representing the number of times that an object of their associated class occurs within the object to which they belong. For example, given the object class word, each object, no matter of which class, has an automatically defined attribute named
CHAPTER 4. DELTO: LANGUAGE REFERENCE

#word that reflects the number of word objects occurring within the piece of text that it represents (actually word is a lexical token, but lexical tokens are considered as any other object by the DELTO System).

All the automatically defined attributes are read-only; there is no way to change the values that the system binds to them, but these values can be referenced by class definitions. These attributes can be bound to one of these types of values: non-negative integers, strings of characters, or REFIDs. The ID attributes are always bound to values of type REFID.

Automatic attributes are not the only kind of attributes that an object can have. Additional attributes can be defined for each class of objects. Defined attributes can be bound to non-negative integer values, string values, or to REFID values. Attributes of type REFID may be bound to the ID of another object. Such a binding implies the existence of a link from the object with the REFID attribute to the one whose ID is bound to that attribute. As opposed to automatic attributes, the system does not automatically bind defined attributes to any values. Class definitions must specify what values are to be bound to them. Once an object is created, the attributes remain as part of its structure and can be referenced by expressions within other class definitions.

Defined attributes are specified through an attributes block within each class definition. Attributes blocks specify the name of each attribute and their domain, that is, the type of values that they can be bound to. The syntax for attributes blocks is defined by the rules given in Figure 4.4.

For example, in Figure 4.5 we present a possible attribute definition for some class of objects.

4.4.2 Components

Objects can have a series of attributes. In addition, objects may also have components. The difference between attributes and components lies in the kinds of values that they may be bound to, and in the way that they can be bound to those values.

An attribute can be bound to a value that does not correspond directly to the contents
\[ <\text{attrs\_block}> ::= \text{ATTRIBUTES} \\
\quad \{ \\
\quad \quad <\text{attr\_def}> \{ ; <\text{attr\_def}> \} \\
\quad \} ; \]

\[ <\text{attr\_def}> ::= <\text{attr\_name}> : <\text{attr\_type}> \]

\[ <\text{attr\_name}> ::= \text{ident} \]

\[ <\text{attr\_type}> ::= \text{NUMERIC} | \text{STRING} | \text{REFID} \]

Figure 4.4: Syntax rules for attribute blocks

\[
\text{ATTRIBUTES} \\
\quad \{ \\
\quad \quad \text{lowercase} : \text{NUMERIC}; \\
\quad \quad \text{plural} : \text{NUMERIC}; \\
\quad \} \\
\]

Figure 4.5: A definition for two attributes
CHAPTER 4. DELTO: LANGUAGE REFERENCE

of the document itself. A word object, for example, may have an attribute named case that is bound to the value "LOWERCASE". This value is arbitrary. There is no direct relation between the contents of the document (the word represented by the word object) and this value. We could have chosen any other value, for example "LOWERC", or "NOCAPS".

On the other hand, components are bound to values that are directly related to the contents of the document itself. For example, a lexical analyzer may generate the lexical tokens given.name and last.name. Instance objects of a class named person.name may be defined as a sequence of one or two objects of the category given.name followed by one of the category last.name. We may want to keep the structure existing within every instance of the person.name class. To accomplish this, we define two components for every such instance: to one of those we bind the given.name objects that form the person.name object instance and to the other one we bind the last.name object. The values that are bound to these components are not arbitrary; they are actual objects that are part of the structure of the document.

Each object that is bound to a component of another object is done so with its complete structure. Therefore, class definitions that define components that can be bound to other objects can reference all the attributes and components of their component objects.

DELTO attributes correspond to SGML attributes, while DELTO components correspond to SGML elements that can occur within other SGML elements.

Components not only can be of the type of other objects, but they can also be of type STRING. STRING components are those that get bound to actual pieces of the analyzed document text itself.

Another important difference between attributes and components is that attributes can only be bound to one single value, while components can be bound to multiple values. When a component is defined the number of values that it can be bound to must be specified with a range whose upper bound can be left unspecified (in which case there is no upper bound to the number of values that the component can be bound to). We use the term component value or simply value to refer to a particular value that is bound to an object component.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

\[
\begin{align*}
\langle \text{components\_block}\rangle & \::= \text{COMPONENTS} \\
& \quad \{'\}
\quad \langle \text{component\_def}\rangle \{ ; \langle \text{component\_def}\rangle \} \\
& \quad \'}
\end{align*}
\]

\[
\begin{align*}
\langle \text{component\_def}\rangle & \ ::= \langle \text{component\_name}\rangle \\
& \quad \ ::= \langle \text{component\_type}\rangle \ [ \langle \text{range}\rangle ]
\end{align*}
\]

\[
\begin{align*}
\langle \text{component\_name}\rangle & \ ::= \text{ident}
\end{align*}
\]

\[
\begin{align*}
\langle \text{component\_type}\rangle & \ ::= \langle \text{class\_name}\rangle \mid \text{NUMERIC} \mid \text{STRING}
\end{align*}
\]

\[
\begin{align*}
\langle \text{range}\rangle & \ ::= \text{integer} \ '.' \text{integer} \mid \text{integer} \ '.' \text{N}
\end{align*}
\]

Figure 4.6: Syntax rules for component blocks

Each object also has an automatically defined statistical attribute for each component it has in its structure. The numbers that are bound to these attributes represent the number of values that are bound to the associated components. The naming convention for these attributes is the same that was described in the previous subsection. For example, an object of class person.name with a component named given.name has an automatically defined attribute named \#given.name that is bound to a value that indicates the number of values that are bound to the given.name component.

A component named text is automatically defined for every object. The DELTO System binds this component to the exact string that the object abstracts (though it removes line breaks and syllabic hyphens; intermediate additional spaces and tabs are represented by only one space).

Components are defined within a components block. The syntax for components blocks is defined by the rules presented in Figure 4.6.

In Figure 4.7 we present an example of a possible components definition for a bibliographic entry class of textual objects.

If the number of values that a component can be bound to is omitted, the range 1..1 is
CHAPTER 4. DELTO: LANGUAGE REFERENCE

COMPONENTS
{
    author : person 1..N;
title : STRING 1..1;
publication_name : publication_name 0..1;
volume : NUMERIC 0..1;
issue : NUMERIC 0..1;
date : date 0..1;
page_range : STRING 0..1;
}

Figure 4.7: Components definition for a bibliographic entry class

assumed. Ranges having the symbol N as an upper bound have no upper bound. Components associated with such ranges can be bound to an unlimited number of values. Ranges having 0 as a lower bound specify that it is legal not to bind any value to the component. The range 0..1 associated with the publication_name component in our example specifies that this component may be bound to either no value at all or to one value. This is saying that a bibliographic entry can still be a bibliographic entry without a value bound to this component. On the other hand, a single value must be bound to the title component in every instance of the bibliographic entry class.

4.4.3 Expressions and Predicates

An expression indicates a property (or properties) that objects of a particular class satisfy. It may also indicate what values should be bound to the attributes and components of an object. Syntactically, expressions are a combination of predicates and logical operators (AND and OR).

An expression may, for example, specify that the piece of text represented by an object of class A must begin with a capitalized word and otherwise it cannot be represented by an object of that class. This example is certain: an object of class A must satisfy the expression. In addition, DELTO allows expressions to specify non-certain properties. This
CHAPTER 4. DELTO: LANGUAGE REFERENCE

is accomplished by attaching two special indicators (attributes may be a better name, but to avoid confusion with object attributes we call them indicators) to each expression: a frequency indicator, and a strength indicator.

Frequency indicators express how common a characteristic is among objects of a certain class, that is, how frequently that characteristic is expected to occur. There are four possible values for frequency indicators. These are ALWAYS, FREQUENTLY, SOMETIMES, and NEVER.

Expressions attached to an ALWAYS indicator must be satisfied by all objects of a particular class (this corresponds to our last example). Expressions associated with a FREQUENTLY indicator should be satisfied by most object instances of the class being described, though it is possible for some instances not to satisfy those expressions. Non-satisfaction of an expression associated with a FREQUENTLY indicator results in a reduction of the object weight (a numerical value from 1 to 100 that represents the confidence that the DELTO System has that an object has been correctly identified, based on the object class definition). SOMETIMES expressions may not be satisfied by most instances of a class but they do not reduce the weight of those objects that do not satisfy them. NEVER expressions are analogous to those attached to the ALWAYS indicator: they must not be satisfied by the objects of the class.

Frequency indicators do not tell how good a characteristic can be as an indicator of an object being of a particular class. Going back to our example, there may be several classes of objects that must always begin with capitalized words, not just those of class A. In that case, that characteristic is not a very strong indicator of an object being of class A. It must be satisfied for an object to be of class A, but satisfying it does not assure that an object is of class A. The other kind of indicators, strength indicators, are the ones that do give this information. There are five possible values for strength indicators. These are: VERY-STRONG, STRONG, MODERATE, WEAK, and VERY-WEAK.

Strength indicators only serve to add weight to an object instance. Expressions attached to the VERY-STRONG indicator are the ones that add more weight, while the ones associated with the VERY-WEAK indicator do not add weight at all (though they may still reduce the object weight depending on which frequency indicator they are associated with, if they are

50
CHAPTER 4. DELTO: LANGUAGE REFERENCE

<expression> ::= <predicate> { <log_op> <predicate> }

<predicate> ::= <argument> [ <rel_op> <argument> ] |
              <predic_name> '(' [ <arg_list> ']' ')' |
              NOT <predicate> |
              '(' <expression> ')'

<arg_list> ::= <argument> {, <argument> }

<argument> ::= <attribute> | <component> | <function> |
              <regular_exp> | integer | literal

<log_op> ::= OR | AND

<predic_name> ::= ident

<rel_op> ::= < | <= | = | <> | >= | >>

Figure 4.8: Syntax rules for expressions

Expressions are always associated with one frequency indicator and to one strength indicator. The way this association is established is discussed in Section 4.4.8. The rules that define the syntax of expressions are given in Figure 4.8.

Expressions are a sequence of one or more predicates connected by the logical operators OR or AND (from lower to higher precedence level). A negation operator (NOT) may precede each predicate. Parentheses may be used to alter the normal rules of precedence.

Predicates express a condition over objects. They can be viewed as boolean functions that are applied to candidate objects during the object recognition process. The simplest predicates are the six relational operators. There are also a set of predefined predicates, each one having a particular name, and requiring a specific number of arguments of a specific type. A description of the predefined predicates is given in Section 4.4.4.

Predicate arguments can be any of the following: integer or string constants, references to attributes or components, functions, or regular expressions. In particular, relational
CHAPTER 4. DELTO: LANGUAGE REFERENCE

operators can be applied only over numeric and string values. References to attributes and components are described in Section 4.4.5, functions are described in Section 4.4.6, and regular expressions are described in Section 4.4.7.

A very simple expression is $word > 1$ and $word < 10$ which expresses that the number of words in a particular object is between 1 and 10. Another example expression is:

\[
\text{pattern}( \langle \text{first\_name} \rangle \langle \text{middle\_name} \rangle \langle \text{last\_name} \rangle )
\]

In this case the predicate name is pattern and the argument is a regular expression. Though we have not yet talked about regular expressions, nor about the predicate pattern, it should be clear enough that the predicate is describing the object (in this case probably a person name) as a sequence of three different objects.

4.4.4 Predicates

We describe now each of the predicates that can be used in DELTO expressions (with the exception of the six relational operators that have already been described). For each predicate we provide a syntactic description of the arguments that it expects. A description of the semantics of each predicate also is given.

Pattern

Syntax:

\[
\text{pattern}( \langle \text{regular\_expr} \rangle )
\]

Describes the piece of text that can be represented by an object of a certain class in terms of a regular expression.

Begins

Syntax:
CHAPTER 4. DELTO: LANGUAGE REFERENCE

\begin{verbatim}
\begins(<regular_expr>)
\end{verbatim}

Describes the beginning of the piece of text that can be represented by an object of a certain class in terms of a regular expression.

\begin{verbatim}
\ends<br>
\end{verbatim}

Syntax:
\begin{verbatim}
\ends(<regular_expr>)
\end{verbatim}

Describes the end of the piece of text that can be represented by an object of a certain class in terms of a regular expression.

\begin{verbatim}
\isa<br>
\end{verbatim}

Syntax:
\begin{verbatim}
\isa(<regular_expr>)
\end{verbatim}

Specifies that the pieces of text that an object of a certain class represents also are represented by an object of another class (given by the regular expression). For example, we may describe a \texttt{title} as also being a \texttt{paragraph}, though this does not mean that all objects of class \texttt{paragraph} are \texttt{titles}. This is the same as saying that a \texttt{title} is a \textit{specialization} of a \texttt{paragraph}. We specify this property in the following way:

\begin{verbatim}
is_a(<\texttt{paragraph}>)
\end{verbatim}

\begin{verbatim}
\in<br>
\end{verbatim}

Syntax:
\begin{verbatim}
in(<regular_expr>)
\end{verbatim}
CHAPTER 4. DELTO: LANGUAGE REFERENCE

This predicate specifies that the pieces of text that may be represented by a particular class of objects are physically contained within pieces of text represented by objects of another class (specified by the regular expression argument). We may, for example, specify that a person name is always inside a sentence. This could be done in the following way:

\[
\text{in( } <\text{sentence}> \text{ )};
\]

Contains

Syntax:

\[
\text{contains( } <\text{regular_expr} > \{, <\text{regular_expr}> \} \text{ )}
\]

This predicate specifies that the piece of text represented by a particular class of objects contains sub-pieces of text that conform to the given regular expressions. The pieces of text matched by the regular expressions given to the predicate need not be consecutive. (There could be text in between them that does not match any pattern.) However, the piece of text matched by the second regular expression (if any) must be after the piece of text matched by the first regular expression. The same ordering is enforced for all the regular expressions given as arguments to this predicate. There cannot be overlapping between the pieces of text matched by the different regular expressions.

Contains_in_any_order

Syntax:

\[
\text{contains_in_any_order( } <\text{regular_expr} > \{, <\text{regular_expr}> \})
\]

Similar to contains but there is no ordering restriction for the pieces of text that the regular expressions can match. Overlapping is also allowed.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

Caps, Heavy_Caps

Syntax:
caps()
heavy_caps()

Caps describes objects that include a relatively good quantity of capitalized words, while heavy_caps describes objects that have a lot of capitalized words. The current implementation considers that an object satisfies the caps predicate when at least 40% of its words are capitalized. The heavy_caps predicate is satisfied when 70% of the words an object contains are capitalized.

Punctuation, Heavy_Punctuation

Syntax:
punctuation()
heavy_punctuation()

Same considerations as with caps and heavy_caps but related to the use of punctuation. The current DELTO System considers that the punctuation predicate is satisfied when 20% of the number of words and punctuation symbols associated with an object are punctuation symbols. For the heavy_punctuation predicate to be satisfied, the percentage that must be reached is 40.

Indented, Indented_1

Syntax:
indented()
indented_1()
CHAPTER 4. DELTO: LANGUAGE REFERENCE

Indented describes a textual structure as completely (or at least, mostly) left-indented. Indented_1 describes a structure whose first line is left-indented.

Similar-to

Syntax:

\[ \text{similar-to( <argument>,<argument>, integer) } \]

This predicate establishes that two strings, given by its first two arguments, are \( n \)% similar to each other, where \( n \) is given by the last argument. For example, we could use this predicate in the following way:

\[ \text{similar-to( string_component, "The frog is waiting", 75 ); } \]

A concept such as "75% similar to a string" is vague and could be interpreted in different ways. The way in which DELTO interprets it is, approximately, as follows: A string is \( n \)% similar to another one \((0 \leq n \leq 100)\) if at least \( n \)% of the words in the string are the same (and in the same order), where words are considered to be the same if they share \( n \)% of the letters (also in the same order). The approximate string matching algorithm used by the DELTO System is a modified version (2 level matching: words and characters) of the one presented in [32, 33].

For example, the words "frog" and "fog" are 75% similar because there is only a difference of one letter in a four letter sequence. Using the DELTO algorithm the strings "The frog is waiting" and "The fog is waiting" are also 75% similar, because the word "frog" can be converted to "fog" (or viceversa) and then all the four words of the string match. The strings "The frog is waiting" and "The fog is" are also 75% similar using this algorithm. Once "frog" is converted to "fog" there are 3 words out of 4 that match in the string, and they are in the same order.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

Irrelevant

Syntax:

\texttt{irrelevant( \textless\textless reg\_exp\_unit\textgreater\textgreater )}

This predicate establishes that for the purpose of recognizing instances of a class, objects of a specific class are irrelevant and should be disregarded. For example, we could describe a person name as a sequence of given names and last names. It could very well be possible that a person name is split over two different lines in a document, that is, there could be a line break interleaved, at any position, within the given names and last names. An expression like the following:

\texttt{pattern( \textless\textless given\_name\textgreater\textgreater 1..2 \textless\textless last\_name\textgreater\textgreater 0..2 );}

would not match a sequence of given names and last names that is split over two lines because there would be a line break that would not match any of the operands of the regular expression. We could fix this in two different ways. First, we could specify a pattern for each possible case: a line break can be located after the first given name, after the second one, or after the first last name, for example. The number of possibilities grows exponentially on the length and complexity of the regular expression, and also on the number of irrelevant classes.

The other way to solve this problem is using the \texttt{irrelevant} predicate. By defining the objects that represent line breaks (for the lexical analyzers that we developed named n1) as irrelevant, then the following two predicates are enough to recognize names, even those that are split over more than one line:

\texttt{irrelevant( \textless\textless n1\textgreater\textgreater );}

\texttt{pattern( \textless\textless given\_name\textgreater\textgreater 1..2 \textless\textless last\_name\textgreater\textgreater 0..2 );}

An \texttt{irrelevant} predicate must be associated with an \texttt{ALWAYS} frequency indicator and it only affects the recognizing process of the class in which the predicate is specified. In
CHAPTER 4. DELTO: LANGUAGE REFERENCE

\[
\text{<attribute>} ::= \{ ( !, \text{<class_name>} \ [\text{<selector>} \] ) \ i( \text{0 integer} ) \} . \]

\text{Figure 4.9: Syntax rule for references to attributes}

other words, it affects all the expressions given in the same class definition, even the ones that are written before the \text{irrelevant} predicate.

4.4.5 Referencing Attributes and Components

Attribute References

We have shown how to define attributes and components though we have not described yet how to reference the values that they are bound to. Attribute values are simply referenced by the name of the attribute, optionally preceded by an indication of the object to which the referenced attribute belongs. For example, the attribute that indicates the starting position of an object may be referenced simply as \text{pos}. The optional indication of the object to which the attribute belongs is described in the next subsection. The grammatical rule for attribute references is shown in Figure 4.9.

Scoping Environments

DELTO is a statically scoped language. At any position within a DELTO class definition there is a precise, non-empty, chain of \text{scoping environments}, each one associated with a class. During actual document processing (run-time), each scoping environment is instantiated with a particular object of the class associated with the environment.

If the optional object indication is omitted from an attribute (or component) reference, the DELTO System searches (during compile-time) the environments chain, from innermost to outermost, looking for an environment associated with a class that has an attribute or component named in the way indicated by the reference. The relation between the reference and the scoping environment is fixed during compile-time. During run-time the value of the referenced attribute or component is extracted from the object associated with
the referenced environment. The DELTO scoping rules are similar to those of the Pascal
programming language.

There are two different ways to override the default environment selection for attribute
or component references. A specific environment can be selected through the use of the at-
environment operator (the symbol '@index' followed by a non-negative integer). This operator
allows the exact specification of which environment an attribute or component reference
is associated with. The integer following the '@index' symbol specifies the environment, where
0 represents the inner-most environment, 1 the environment outside the inner-most one,
and so on. For example, the reference $@2.address$ returns the value of an attribute or
component named address of the object associated with the environment two levels above
the inner-most one.

When using the at-environment operator, the specified environment must exist, and the
referenced attribute or component must be defined within the class associated with the
specified environment. There are only 3 syntactical constructions in DELTO that add a
new environment to the environments chain. These are:

- Class definitions
- Expression selectors
- Regular expression units

Expression selectors are described in this section and regular expression units are de-
scribed in Section 4.4.7.

The second way to override the default environment selection is through the use of the
global-object operator. This operator specifies the object or objects from which the attribute
or component value must be extracted. An object from any class can be referenced through
this operator. In addition, the referenced object need not be on the environments chain.
This is why we call these kinds of references global references. The first identifier following
the '!' operator is the name of a class. An optional selector (selectors are described in
CHAPTER 4. DELTO: LANGUAGE REFERENCE

\[
\text{<component>} \quad ::= \quad \left[ ( ( \text{!} \text{. <class_name>} [ \text{<selector>} ] ) \mid ( \text{@} \text{integer} ) [ . ] ) \right] \\
\quad \quad \quad \quad \quad \quad [ \text{<component.name>} \quad [ \text{<selector>} ] \quad ] \\
\quad \quad \quad \quad \quad \quad \quad \quad [ \text{. <attribute>} ]] \\
\]

\[
\text{<selector>} \quad ::= \quad \text{'} [ \text{<index.binding>} ] \text{integer } \text{'} ] \quad | \\
\quad \quad \quad \quad \text{'} [ \text{<index.binding>} ] \text{ <sel.keyword} \text{'} ] \quad | \\
\quad \quad \quad \quad \quad \text{'} [ \text{<index.binding>} ] \text{ <expression} \text{'} ] \quad | \\
\quad \quad \quad \quad \quad \quad \text{'} [ \text{<index_var>} ] \quad | \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{'} [ \text{'} ] \\
\]

\[
\text{<index.binding>} \quad ::= \quad \text{<index_var>} \quad \text{.=} \\
\]

\[
\text{<index_var>} \quad ::= \quad \text{ident} \\
\]

\[
\text{<sel.keyword>} \quad ::= \quad \text{EACH} \quad | \quad \text{ALL} \quad | \quad \text{ANY} \quad | \quad \text{LAST} \quad | \quad \text{NEW} \\
\]

Figure 4.10: Syntax rules for references to components

this section) may specify which object or objects of the given class are being referenced. If a selector is omitted, the first object of the given class that was created by the DELTO System is the one that is used for the reference (in particular, !.document[1] always refers to the object representing the entire document being processed). Global references can also be used to create objects that are not directly associated with any piece of text of the input document. This is described in Section 4.4.9. Following the class name and the optional selector, the name of the referenced attribute or component from the specified object or objects may be indicated, otherwise the reference is to the entire object. For example, the global reference !.person_name.text can be used to extract the value of the text component of an object of class person_name (in this case, the first created object of that class, due the lack of a selector).

Component References

The grammatical rules for component value references are illustrated in Figure 4.10.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

COMPONENTS
{
    year : NUMERIC 1..1;
    month: NUMERIC 1..1;
    day  : NUMERIC 0..1;
}

Figure 4.11: Components for the date class

The same scoping rules that apply for attribute references apply for component references. A component can be referenced in the same way as an attribute: just by its name. If we want to reference specific attributes or subcomponents of a component we need to use a qualified name, consisting of the name of the component, followed by a dot, followed by the name of the subcomponent or attribute. A reference can be qualified as many times as we want, to go deep in the structure of a component, though no qualification can follow the name of an attribute (because attributes do not have any finer structure).

For example, assuming the date class has the components defined in Figure 4.11, date.year is a valid reference to one of its components (where date is a component of type date).

Selectors

Specific values can also be referenced, if a component is bound to more than one value. This is done through value selectors. Value selectors are enclosed in square brackets. A value selector can always follow a reference to a component. Multiple selectors can be part of a reference if the reference has several levels of qualification. Object selectors have the same syntax as value selectors. They are used in combination with global references to select one or more objects from all the objects of the class indicated by the global reference.

A number $n$ used as value selector selects the $n$th value bound to the indicated component. The first value is referenced by the number 1, the second by the number 2, and so on. For example, given the components definition presented in Figure 4.7 author[2] is a
CHAPTER 4. DELTO: LANGUAGE REFERENCE

valid reference to the second author of a bibliographic entry.

A number can also be used as an object selector, though it is less useful for that purpose. A number $n$ used as an object selector selects the $n$th object of the indicated class where the $n$th object represents the object of the given class that is created by the DELTO System after exactly $n - 1$ objects of the same class were created. There is no way to control the order in which the DELTO System creates objects. For this reason numbers are rarely used as object selectors. However, the reference $! . \text{document}[4]$ is a convenient way to refer to the entire document being processed.

Selector keywords can also be used to select specific component values or objects. The valid selector keywords are EACH, ALL, ANY, LAST and NEW.

The EACH keyword selects all the values of a component. For example, the expression $A[\text{EACH}] > 7$ is true if each of the values of component $A$ is greater than 7.

EACH also selects all the objects of the indicated class, when used in combination with global references. The reference $! . \text{person}[\text{EACH}]$ returns all the objects of class person created by the DELTO System.

ALL must be used in combination with string type components. It selects a value that is the concatenation of all the values bound to the component, each separated by a space. Assuming that component $B$ is of type string, and that it is bound to four different values ("Hello", "I", "am", "DELTO"), then $B[\text{ALL}]$ represents the concatenation of those 4 values ("Hello I am DELTO").

The keyword ANY also selects all the values of a component, or all the objects of a class (when used in global references). The difference between ANY and EACH is in the way that they are interpreted by the DELTO predicates. While EACH is interpreted as a universal selector, ANY is interpreted as existential. For example, the expression $A[\text{EACH}] > 7$ is true if each value of $A$ is greater than 7. However, $A[\text{ANY}] > 7$ is satisfied if at least one value of $A$, no matter which one, is greater than 7. The expression $A[\text{ANY}] > B[\text{EACH}]$ is true only if at least one value of $A$ is greater than all the values of $B$.

The keyword LAST selects the last value of a component. If, for example, a component
is bound to seven values, \texttt{LAST} selects the seventh value. When used as an object selector, the \texttt{LAST} keyword selects the last created object of the indicated class, unless there has been another reference to an object of the same class within the same class definition. In this case, \texttt{LAST} selects the \textit{last referenced} object. These two different meanings in the semantics of the \texttt{LAST} operator, when used in global references, is not a good feature of DELTO. A revised version of DELTO should incorporate two different selectors: \texttt{LAST} and \texttt{LAST-REFERENCED}.

\texttt{NEW} creates a new, uninstantiated value for the indicated component and selects this newly created value. It must always be used on the left hand side of a binding. For example, the left hand side reference in $A[\texttt{NEW}] := B[\texttt{LAST}]$ binds a new value to component $A$. This operation does not alter any previously existing values that were bound to component $A$, it only adds a new one. The new value is the result of evaluating the right hand side of the binding.

When used as an object selector, \texttt{NEW} creates a new object of the specified class. For example, $!.\texttt{person}[\texttt{NEW}]$ creates a new object of class \texttt{person}. This allows for the creation of objects that are not directly related to the text of the input document. This feature is described in detail in Section 4.4.9 of this chapter.

The empty selector [] may be used as an abbreviation of \texttt{[NEW]}.

Another way in which values of a component can be selected is through the use of expressions. Expressions are described in detail in Section 4.4.3 of this chapter. Expression selectors can be applied over components which are bound to values that are objects themselves (as opposed to string or numeric values). For example, assuming the definition for an object of class \texttt{date} given in Figure 4.11, the reference $\texttt{date[year > 1990]}$ selects date values for which the \texttt{year} component is greater than 1990.

Expressions also can be used together with object selectors, to select the objects of a given class that satisfy the properties specified by the selector expression. For example, $!.\texttt{person}[\texttt{given\_name} = "John"]$ selects all the objects of class \texttt{person} that satisfy the selector expression.

If the selector expression is unable to select any value and the reference is on the left-
CHAPTER 4. DELTO: LANGUAGE REFERENCE

hand side of a binding operation, then a new value (or object if it is a global reference) is created, in the same way that the NEW selector would do.

An expression may select more than one value. If this is the case, then these values are treated by all the predicates as if they had been selected by the EACH selector. For example, consider the expression date[year > 1990].day > 20. For this expression to be true each selected date (year component greater than 1990) must have a value bound to the day component that is greater than 20. If there is one single date with a year greater than 1990 and a day not greater than 20, then the expression is not satisfied.

Furthermore, all selectors that may return more than one value from a component are interpreted in this way, except for the ANY selector, which is always interpreted in the way that has previously been described (existential).

Selectors may also incorporate index variables. Index variables record the positions of selected values from a component and can be used to reference again those values. For example:

\[
A[i .\,=\, \text{ANY}] > 7 \ \text{AND} \ A[i] < B[\text{EACH}]
\]

In the previous example, for the expression to be true there must be at least one value bound to \( A \) greater than 7 and that same value has to be less than every single value bound to \( B \). Note that it would be impossible to express the semantics of this expression without the use of index variables. The following expression, for example, has a completely different meaning:

\[
A[\text{ANY}] > 7 \ \text{AND} \ A[\text{ANY}] < B[\text{EACH}]
\]

In this case, for the expression to be true there has to be at least one value of \( A \) that is greater than 7, and at least one value of \( A \) that is less than all the values of \( B \), but in this case the selected values of \( A \) do not have to be the same. In the previous example an index variable allows one to specify that the selected values of \( A \) must be the same. Index variables can represent more than one value, if they are assigned the results of a selector
CHAPTER 4. DELTO: LANGUAGE REFERENCE

\[
\text{<function> ::= ( <funct_name> '(', [<funct_arg> ', ', <funct_arg> ] ')') } \mid \\
( <funct_arg> ' + ' <funct_arg> )
\]

\[
\text{<funct_arg> ::= <attribute> } \mid \text{<component> } \mid \text{<function> } \mid \text{literal } \mid \text{integer}
\]

\[
\text{<funct_name> ::= ident}
\]

Figure 4.12: Syntax rules for function invocations

expression or of the selector keyword EACH. The scope of index variables always extends to the whole expression in which they are used.

4.4.6 Functions

Functions are one of the possible types of predicate arguments. A function may receive zero or more arguments and returns zero or more values. Function arguments can be string or numeric constants, references to attributes or components, or other functions. The current DELTO implementation incorporates five functions, which are described in this Section. All of these functions return values of string type.

In Figure 4.12 we introduce the syntax for function invocations.

The ‘+’ operator represents the binary string concatenation function. Its syntax is different from the one of all the other functions (infix as opposed to prefix). For example, the following predicate contains this function as one of its arguments:

\[
\text{similar-to( str_comp1 + str_comp2 + str_comp3, "Hello", 90 )}
\]

We describe now the other four functions currently available. For each function we provide a syntactic description of the arguments that it expects, and a description of the values that it computes.

Concat

Syntax:
CHAPTER 4. DELTO: LANGUAGE REFERENCE

concat( <funct.arg> )

Returns a single value that represents the concatenation of all the values referenced by the argument. A space is used to separate each value in the returned string. If, for example, component A is bound to two values, then concat(A[each]) returns the concatenation of those two values with a space separating them.

Initial

Syntax:

initial( <funct.arg> )

Returns a string consisting of the first letter followed by a period for every string referenced by the argument. As opposed to concat that returns only one value regardless of the number of values referenced by its argument, initial returns as many values as it receives. For example, if component A is bound to four values ("John", "Paul", "George", and "Ringo"), then initial(A[each]) also returns four values ("J.", "P.", "G.", and "R.").

Clear-String

Syntax:

clear-string( <funct.arg> )

Returns, for every string referenced by the argument, a new string in which line breaks, additional intermediate spaces (more than one contiguous blank character), and SGML markup (tags and entities) are removed.

This function is useful to clean up data extracted from input documents that must be saved into a permanent object database. For example, a title extracted from a document may contain unwanted intermediate spaces and unwanted SGML markup. Cleaning the string with this function before saving it into the object database results in better data stored into the database, and in data that can be more easily retrieved in the future.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

For example, if the following title has been identified from a document and bound to component title:

Designing a Document System

then clear-string(title) produces the following, "clean" version of the title:

Designing a Document System

4.4.7 Regular Expressions

Regular expressions are one of the possible kinds of arguments that a predicate may take. DELTO regular expressions are in many ways very similar to the ones we already know and in some other ways quite different. As usual, the ‘|’ operator is used to mark different alternatives. This operator has the lowest precedence level of all regular expression operators. The usual occurrence operators are also available in DELTO regular expressions: ‘?’ denotes zero or one occurrence, ‘*’ denotes an arbitrary number of occurrences, and ‘+’ indicates one or more occurrences. Ranges can also be used (in the same way that they are used to define the number of values that can be bound to a component) to specify number of occurrences more precisely. All the occurrence operators, including ranges, have the same level of precedence, which is higher than the one of ‘|’. As usual, parentheses can be used to alter the normal rules of precedence. DELTO regular expressions also provide
CHAPTER 4. DELTO: LANGUAGE REFERENCE

\[
<\text{regular\_exp}> ::= <\text{regular\_oprnd}> \{ <\text{regular\_oprnd}> \}
\{ ';' <\text{regular\_oprnd}> \{ <\text{regular\_oprnd}> \}\}
\]

\[
<\text{regular\_oprnd}> ::= <\text{reg\_exp unit}> [ <\text{occur\_indic}> ] |
<\text{reg\_exp symbol}> |
\{ <\text{component}> ::= <\text{wildcard}> 
[ [: ']' \{ <\text{class\_name}> \} ''] 
[ <\text{occur\_indic}> ] |
\text{literal} [ <\text{occur\_indic}> ] |
\text{similar to} \{ (' literal, integer ') \} [ <\text{occur\_indic}> ] |
'(' <\text{regular\_exp}> ')'; [ <\text{occur\_indic}> ]
\]

\[
<\text{reg\_exp unit}> ::= '<<' <\text{class\_name}> \{, <\text{expression}> \} '>>'
\{ '[' <\text{comp\_binding}> \{, <\text{comp\_binding}> \} ']' \}
\]

\[
<\text{reg\_exp symbol}> ::= \text{BOUNDARY} \mid \text{OBJECT} \mid \text{DOC\_BEG} \mid \text{DOC\_END}
\]

\[
<\text{wildcard}> ::= \text{ANYTHING}
\]

\[
<\text{occur\_indic}> ::= '?' \mid '*' \mid '+' \mid <\text{range}>
\]

Figure 4.13: Syntax rules for regular expressions

ways to bind components to values. The syntax of regular expressions is defined by the
rules given in Figure 4.13.

Regular expressions are required as arguments in most of the predefined predicates. All the
predicates that receive regular expressions as arguments try to match those regular
expressions with some portion of the document being analyzed. A complete list and
description of all the predefined predicates is given in Section 4.4.4.

There are three different kinds of regular expression operands: regular expression units, regular expression special symbols, and strings. We now describe each of them.

**Regular Expression Units**

A regular expression unit (from now on just a unit) represents a textual object recog-
nized by the DELTO System (which also could be a lexical token recognized by the
CHAPTER 4. DELTO: LANGUAGE REFERENCE

lexical analyzer). Units are written enclosed in the symbols ‘<<’ and ‘>>’. A unit such as <<word>> matches any object of class word within the analyzed document. One such as <<paragraph>> matches any instance of an object class named paragraph. The regular expression <<title>><<paragraph>>+ matches an object of the class title followed by one or more objects of the class paragraph.

Units such as <<title>> match any instance of the class title. But, what if we want a unit to match only some instances of a certain class of objects? What if we want a unit to match only instances of a particular class whose attributes or components satisfy certain properties? For example, we may want a unit to match titles that have more than 5 words or paragraphs that take up more than 3 physical lines. We can achieve this kind of matching by including expressions within units that describe more precisely the kind of object that the unit is intended to match. The expressions that can be included in units have the same syntax and semantics as the ones that can be used directly within class definitions.

The following is a regular expression that matches titles that have more than 5 words:

<<title, #word > 5>>

We may also want to match a title that follows a specific pattern. For example, all titles that include one or more consecutive person names. The following regular expression can do the job:

<<title, contains( <<person_name>>+ )>>

In our last example there is a regular expression inside another regular expression. This can be continued to any depth we need, to precisely define a regular expression unit.

There can be any number of expressions, separated by commas, as part of a unit. An object that matches the unit must be of the class indicated by the unit, and must satisfy all the expressions included in the unit.

Regular expression units add a new scoping environment to the environments chain. This new environment is associated with the class represented by the unit and, in particular,
CHAPTER 4. DELTO: LANGUAGE REFERENCE

CLASS DEFINITION: A
  ATTRIBUTES
  {
    cc : NUMERIC;
  }
  /* There may be expressions here */
ENDDEF;

CLASS DEFINITION: B
  ATTRIBUTES
  {
    cc : NUMERIC;
  }
  DESCRIPTION
  {
    /* More expressions may go here */
    pattern( <<A, cc > 20>>+ );
  }
ENDDEF;

Figure 4.14: Scoping example

with each object that is matched against the unit. The added environment becomes the inner-most environment of the chain. Expressions included in units are evaluated under this updated environments chain. In the example of Figure 4.14, the expression included in the unit for class A within the definition of class B is evaluated under an environment chain that consists of class B (the class being defined), and of class A (the class of the unit, and the inner-most environment). Objects of class A do have an attribute named cc. For that reason the reference to cc within the expression included in the unit is associated with the inner-most environment, and during run-time, with the objects of class A matched by the unit.

In the example presented in Figure 4.15, assuming the class definition for A given in Figure 4.14, the expression included in the regular expression is comparing the two cc
CHAPTER 4. DELTO: LANGUAGE REFERENCE

CLASS DEFINITION: B
  ATTRIBUTES
  {
    cc : NUMERIC;
  }
  DESCRIPTION
  {
    /* More expressions may go here */
    pattern( <<A, cc > @1.cc>>+ );
  }
ENDDEF;

Figure 4.15: Scoping example

attributes: the one of the object being recognized (of class B) and the one of the object
matched by the regular expression unit (of class A). The reference to the cc attribute of the
class B object must be preceded with the at-environment operator, which specifies exactly
the environment that the reference is associated with.

Regular expression units may be followed by a series of component bindings. These
component bindings are executed only if the unit associated with the bindings is matched,
if the entire regular expression matches, and for each object that is recognized using the
class definition that contains the regular expression. For example:

    pattern( (<<A>>[Obj_A[NEW] := @0] | <<B>>[Obj_B[NEW] := @0]) <<C>> )

This expression results in the binding associated with the A unit being executed if the
whole regular expression matches and, in particular, if the A unit is matched (a <<A>><<C>>
sequence is matched). The binding associated with the B unit is executed only when the
sequence <<B>><<C>> is matched. Note that for a particular sequence matched by this
regular expression only one of the two bindings is executed.

The scoping environment associated with these component bindings is the same one that
is associated with expressions included within a regular expression unit: it includes, as the
CHAPTER 4. DELTO: LANGUAGE REFERENCE

CLASS DEFINITION: B
    COMPONENTS
    {
    Obj_A : A 1..N;
    cc : NUMERIC 1..N;
    }
    DESCRIPTION
    {
    ALWAYS
    {
    pattern( <<A, cc > 7>>[Obj_A[NEW] := @0,
                               @1.cc[NEW] := cc]+ );
    }
    }
ENDDEF;

Figure 4.16: Component bindings example

inner-most environment, the environment associated with the class of the regular expression unit. In the example given in Figure 4.16, each (note the ‘+’ occurrence operator that also affects the component bindings) matched object of class A is bound to the component Obj_A of the recognized object of class B, and the value of the cc attribute of each matched object is bound to the cc component of the recognized object. Note the use of the unqualified @0 operator to represent the entire objects associated with the inner-most environment (in this case, the objects of class A matched by the regular expression).

Note, in the example of Figure 4.16, the use of the NEW selector that makes the values originated on the right-hand side of binding operations be bound to new values of the components on the left-hand side. Note also that Obj_A is not preceded by an at-environment operator, even though it is associated with the scoping environment related to the B class (environment level 1). The at-environment operator (in this case it would be @1) is not needed because there is no attribute or component with the same name (Obj_A) defined for class A (the class associated with the inner-most environment). The @1 operator could have been used to precede the reference to Obj_A though it would have not made any difference.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

The at-environment operator is required to reference the \texttt{cc} component of the \texttt{B} class objects, because there is an attribute defined with the same name in a scoping environment (the inner-most one, associated with objects of class \texttt{A}) that is inner-located than the one that the reference has to be associated with.

**Literal Strings**

Literal strings are also valid regular expression operands. A literal string appearing as a regular expression operand can be matched to an identical string in the input document. String matching is case insensitive and intermediate white space and line breaks are irrelevant. A valid regular expression can then be, for example, \texttt{\textless \textless person.name\textgreater \textgreater "Ph.D."}.

What the DELTO System does with a regular expression string is partition it into substrings, in the following way: substrings may be words or numbers (containing no spaces and no punctuation or other non-letter and non-digit characters); any punctuation symbol or other non-letter or non-digit character is a substring in itself; intermediate spaces, tabs, and line breaks are ignored. The DELTO System intends to match each of these substrings with the string represented by a token generated by the lexical analyzer. Sometimes it may be necessary to know the way in which the lexical analyzer partitions certain kinds of input. For example, assuming that the lexical analyzer treats "Ph.D." as one single token, the previous regular expression string ("\texttt{Ph.D."}) would not match a token like that, because the DELTO System partitions this string into 4 substrings: "Ph", ".", "D", and ".". To avoid these problems, the ‘$’ character can be used within string regular expressions. The meaning of the dollar symbol is “do not split into substrings until the next dollar sign is found”. The regular expression \texttt{\textless \textless person.name\textgreater \textgreater "$\texttt{Ph.D.}$"} would then correctly match the output generated by the lexical analyzer.

A dollar sign within a regular expression string can be escaped (preceded by the ‘\texttt{\textbackslash}’ character) if it only wants to be considered as a dollar sign character and not as the “do not split” operator.

Sometimes it is good to allow some flexibility with respect to the strings that can be
CHAPTER 4. DELTO: LANGUAGE REFERENCE

matched. That is why DELTO provides also approximate string matching capabilities. A regular expression like the following:

\[ <<\text{person\_name}>> \text{similar\_to} ("is the author of this document", 70 ) \]

matches a person_name object followed by a string that is 70% similar to the string enclosed in the similar_to operand. The `$` operator also may be used in combination with strings given to the similar_to operand.

Note that this regular expression operand is not the same as the similar-to predicate, even though they utilize the same algorithm to determine if two strings are similar to each other. Similar-to is a predicate that compares two given strings, either string constants or values bound to string components, for example:

\[ \text{similar\_to} (\text{string\_component[1]}, "The frog is waiting", 75 ); \]

On the other hand, the similar_to regular expression operand is used within regular expressions in order to approximately match a piece of the input document.

Regular Expression Special Symbols

The following special symbols are also valid regular expression operands: \text{DOC\_BEG}, \text{DOC\_END}, \text{BOUNDARY}, \text{OBJECT}, and \text{ANYTHING}.

The symbols \text{DOC\_BEG} and \text{DOC\_END} represent the beginning and the end of the document, respectively. They can be used to establish that a certain pattern can be matched only at the beginning or at the end of the document. For example:

\[ \text{pattern} ( \text{DOC\_BEG} <<\text{institution\_name}>>2..2 ) \]
\[ \text{pattern} ( <<\text{signature}>>+ \text{DOC\_END} ) \]

The \text{BOUNDARY} symbol is used to provide left and right context to a pattern. In particular, it represents the point, or points, at which an object begins or ends. The \text{BOUNDARY} symbol can be used in combination with the \text{begins}, \text{pattern}, and \text{ends} predicates. We can use the \text{BOUNDARY} symbol in combination with the \text{begins} predicate in the following ways:
CHAPTER 4. DELTO: LANGUAGE REFERENCE

\begin{verbatim}
begins( BOUNDARY <<class1>> <<class2>> );
begins( <<class1>> <<class2>> );
begins( <<class1>> BOUNDARY <<class2>> );
\end{verbatim}

The first pair of \texttt{begins} predicates are equivalent. They specify that the object being defined begins with a sequence of two particular objects (of type \texttt{class1} and \texttt{class2}), and that those objects are actually part of the object being defined. No left-context information is provided by either of the regular expressions. If the \texttt{BOUNDARY} symbol is omitted from a regular expression given to a \texttt{begins} predicate, it is assumed to be located at the beginning of the regular expression. The third predicate expresses that the object begins with \texttt{<<class2>>}, though there must be a \texttt{<<class1>>} immediately before the \texttt{<<class2>>}, even though \texttt{<<class1>>} is not part of the defined object itself.

The \texttt{pattern} predicate can be used in the following ways, in combination with the \texttt{BOUNDARY} symbol:

\begin{verbatim}
pattern( BOUNDARY <<class1>> <<class2>> BOUNDARY );
pattern( <<class1>> <<class2>> );
pattern( <<class1>> BOUNDARY <<class2>> );
pattern( <<class1>> BOUNDARY <<class2>> BOUNDARY <<class3>> );
\end{verbatim}

The first two \texttt{pattern} predicates are equivalent. Two \texttt{BOUNDARY} symbols are allowed in a regular expression acting as argument to the \texttt{pattern} predicate, the first one indicating the beginning and the second indicating the end of the represented textual object. If both are omitted, they are assumed to be located at the beginning and at the end of the regular expression respectively. If only one is present, it is assumed to be the beginning one (the one that provides left-context), while the other one is assumed to be present at the end of the expression.

Finally, the \texttt{ends} predicate and the \texttt{BOUNDARY} may be combined in the following ways:

\begin{verbatim}
ends( <<class1>> <<class2>> BOUNDARY );
ends( <<class1>> <<class2>> );
\end{verbatim}
CHAPTER 4. DELTO: LANGUAGE REFERENCE

```
ends( <<class1>> BOUNDARY <<class2>> );
```

The usage of the BOUNDARY symbol in combination with the ends predicate is analogous to its usage in combination with the begins predicate, with the difference that, in this case, the BOUNDARY symbol provides right-context instead of left-context. Within regular expressions given as arguments to the ends predicate, if a BOUNDARY symbol is not explicitly written in some part of the expression, then it is assumed to be located at the end of the expression. For this reason, the first two expressions given in the previous example are equivalent.

As we already mentioned, the BOUNDARY symbol can only be used in regular expressions that are arguments to the predicates pattern, begins, or ends. In particular, it cannot be used in combination with the predicates contains or contains_in_any_order.

The ANYTHING symbol, or wildcard symbol, matches any single object. It can be followed by a list of object classes that it should not match. For example:

```
pattern( <<word>> ANYTHING:[given_name, last_name] )
```

The previous regular expression matches a word followed by anything except for a given_name or a last_name. In some cases restrictions on the classes of objects that can be matched by an ANYTHING symbol are automatically imposed by the DELTO System. This occurs when the matching of the wildcard symbol is optional and there are other possible regular expression operands that could be used to decide what class of object to match. For example:

```
pattern( <<word>> ANYTHING* <<given_name>> );
```

In this case the wildcard symbol never matches an object of the class given_name, because an object of that class always matches the regular expression unit, representing an object of that class, that follows the wildcard symbol. This makes the use of the ANYTHING symbol safe when combined with occurrence indicators because the extension of the text that can be matched against it is limited by the regular expression operands following it.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

The string matched by the ANYTHING symbol may be bound to a string component, in the following way:

\[
\text{pattern( <<word>> string\_component[NEW] := ANYTHING* <<given\_name>> )};
\]

In the case of the last example, the exact string that is matched by the ANYTHING* symbols is bound as a new value of \text{string\_component}, which must be a component of type \text{string}. Note that the precedence of the occurrence operator '*=' is higher than that of the binding operator '='; which means that only one new value is bound to the component and this value is the entire string that is matched between a \text{word} and a \text{given\_name}.

The current DELTO compiler does not recognize the construction that we just described, though a future version of it should. However, our current compiler recognizes an abbreviated form that has exactly the same semantics. The abbreviated form does not include the ANYTHING* symbols. The following example is equivalent to the one above:

\[
\text{pattern( <<word>> string\_component[NEW] <given\_name>> )};
\]

Note that there should be two precise limits that determine the string that is bound to a string component, like the \text{word} and \text{given\_name} units in the above example. A \text{pattern} expression like any of the following:

\[
\text{pattern( <<word>> string\_component[NEW] );}
\]
\[
\text{pattern( string\_component[NEW] <given\_name>> );}
\]
can be matched up to the end of the document (in the case of the first expression), and up to the beginning of the document (for the second expression). These kinds of expressions are still valid as long as a precise description for the strings that can be bound to the string components is given. This can be done through the use of \text{string component descriptions}, which are defined in Section 4.4.10 of this chapter.

The \text{OBJECT} symbol can be used to define only the left-context and right-context of an object. For example, we could define a paragraph in terms of its context instead of on the basis of its contents (\text{nl} is an object that represents newline characters):
CHAPTER 4. DELTO: LANGUAGE REFERENCE

\[ <\text{toplv}\_\text{exp}> ::= [ <\text{label}> : ] <\text{expression}> \]
\[ \quad [ \{ \text{'} \text{'} \\text{attrib\_bind\'} \\{, \text{'} \\text{attrib\_bind\'} \text{'} \} \text{'} \} \] \]

\[ <\text{attrib\_bind}> ::= <\text{attribute}> \text{'}\text{=}\text{'} \]
\[ \quad ( <\text{attribute}> \mid <\text{component}> \mid <\text{function}> \mid \text{integer} \mid \text{literal} ) \]

\[ <\text{label}> ::= \text{ident} \]

Figure 4.17: Syntax rules for top level expressions

\[ \text{pattern}( \text{<<nl>>} \text{<<nl>> OBJECT} \text{<<nl>>} \text{<<nl>>} ) \]

The OBJECT symbol can only be used in combination with the \text{pattern} predicate. The previous example is equivalent to the following:

\[ \text{pattern}( \text{<<nl>>} \text{<<nl>> BOUNDARY ANYTHING* BOUNDARY} \text{<<nl>>} \text{<<nl>>} ) \]

4.4.8 Combining Expressions

Usually several expressions need to be included as part of a class definition to appropriately describe the properties of the objects of the defined class. In addition, expressions have to be associated with a strength and a frequency indicator, and they may also trigger the bindings of attributes to particular values.

Binding Attributes

We have discussed how to bind \textit{components} to values, though nothing has yet been said about how to bind \textit{attributes} to values. \textit{Top-level expressions} (those appearing directly within a class definition, and not within a regular expression unit) can optionally be followed by an attribute binding construction. If the expression is satisfied, the specified bindings (if any) are performed. The syntax for top level expressions is defined by the rules shown in Figure 4.17.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

Attributes can only be bound to string or numeric values. The types of the attribute and of the value must match for the binding to be legal. The following example binds the numeric attribute type to a value, according to the pattern that is matched:

\[
\text{pattern( <<class1>> <<class2>> ) [type:= 1];}
\]
\[
\text{pattern( <<class2>> <<class1>> ) [type:= 2];}
\]

Recall our discussion about the differences between attributes and components. The values bound in the example to the type attribute are completely arbitrary. We could have chosen any other values instead of 1 and 2. Components, on the other hand, are bound to values that come directly from the document text.

Expression Labels

Top level expressions can be labeled, so that they can be undefined by specific class definitions. The label is simply an identifier. The same identifier can be used to label more than one expression. In this case multiple expressions can be undefined simply by referencing one label. For example, the following expressions:

\[
\text{my_label: #sentence > 30;}
\]
\[
\text{my_label: #word > 200;}
\]

can be both undefined by an UNDEFINE statement such as UNDEFINE my_label;.

Strength Indicators

The rules that define how to specify the strength indicator associated with an expression are given in Figure 4.18

As indicated by the rules, a strength indicator may follow an expression. For example, the expressions in Figure 4.19 could be part of the definition for paragraph objects.

If a strength indicator is not explicitly associated with an expression, the expression is associated, by default, with the moderate strength indicator.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

<strength_block> ::= [ <toplex> [ <strength_indic> ]
                   ; <toplex> [ <strength_indic> ] ]

<strength_indic> ::= ( VERY-STRONG | STRONG | MODERATE | WEAK | VERY-WEAK )
                   INDICATOR

Figure 4.18: Syntax rules for strength blocks

#sentence > 2
began(<<word, case=$begins_upper
      or case=$all_upper>>)           VERY-STRONG INDICATOR;
ends(".");
#line > 3                      MODERATE INDICATOR;

Figure 4.19: Usage of strength indicators

Frequency Indicators

Frequency indicators are specified through frequency blocks. The grammatical rule for
frequency blocks is shown in Figure 4.20.

As opposed to strength indicators, frequency indicators must be explicit; there is not a
default frequency indicator. A frequency indicator may affect one or more expressions, as
opposed to strength indicators that only affect one expression. In Figure 4.21 we present
a complete example, based on the one that we gave in the Figure 4.19 that consisted of a
partial definition for paragraph objects with the addition of frequency indicators. (Please
do not rely on the semantic correctness of this definition, its only purpose is to serve as an
example.)

The example of Figure 4.21 should be read in the following way: A paragraph

<frequency_block> ::= ( ALWAYS | FREQUENTLY | SOMETIMES | NEVER )
                     '{
                     <strength_block>
                     '}'

Figure 4.20: Syntax rule for frequency blocks
FREQUENTLY
{
    #sentence > 2
    VERY-STRONG INDICATOR;
}
ALWAYS
{
    begins(<<word, case=$begins_upper
    cr case=$all_upper>>) MODERATE INDICATOR;
    ends(".");
}
FREQUENTLY
{
    #line > 3
    WEAK INDICATOR;
}

Figure 4.21: Frequency blocks

frequently has more than 2 sentences, and objects composed of more than 2 sentences are very likely to be paragraphs (because of the very-strong indicator). Paragraphs always begin with capitalized words and always end with periods. If any of these two conditions is not satisfied then the object cannot be a paragraph. Anyway, the fact that either of these conditions are satisfied by a candidate object, does not provide more than a moderate indicator that the object is a paragraph. Finally, a paragraph frequently takes up more than 3 physical lines, though this condition does not provide much evidence to classify an object as a paragraph, because it is associated with a weak strength indicator.

Syntactically, the examples show that there is no restriction in the ordering or in the number of frequency blocks that can be given for a class definition. In addition, the ordering or grouping of frequency and strength blocks and of the expressions within them has no relevance at all. This helps DELTO be a more declarative language, because there is no need to worry about the order in which the expressions have to be given for a particular class definition.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

<description_block> ::= DESCRIPTION
                 '{
                    { <frequency_block> }
                 '}

Figure 4.22: Syntax rule for description blocks

DESCRIPTION
{
  FREQUENTLY
  {
    #sentence > 2
    VERY-STRONG INDICATOR;
  }
  ALWAYS
  {
    begins(<<word, case=$begins_upper
          or case=$all_upper>>) MODERATE INDICATOR;
    ends(".");
  }
  FREQUENTLY
  {
    #line > 3
    WEAK INDICATOR;
  }
}

Figure 4.23: A description block

Putting it all together

All frequency blocks of a class definition must be grouped together in a description block. The syntax for description blocks is presented in Figure 4.22.

In Figure 4.23 we present the same example that was given in Figure 4.21 but now as a complete description block. The only difference between the two examples is that in the second one the frequency blocks are all enclosed within a single description block.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

CLASS DEFINITION: Person;
PERSISTENT INSTANCES
COMPONENTS
{
    Name : Person_Name 1..1;
    Institution : Institution_Name 1..1;
}
ENDDEF;

Figure 4.24: A class definition for person

4.4.9 Non-Textual Objects

Sometimes it may be useful to construct objects that do not represent directly a piece of text of the input document, but that are composed of information extracted from different objects. For example, we could define a person class as presented in Figure 4.24.

The class definition for person given in our example does not have a description block. This means that the DELTO System has no way to recognize any instance of person from the text of an input document. However, the DELTO System may still be able to construct objects of this class. The person class has two components, one of class person.name and the other one of class institution.name. When a person.name is identified by the system, the description may indicate that an object of class person must be created, and the identified person.name object becomes bound to the name component of the created person object. If, for example, an institution.name is found next to the person.name, then the description may indicate that that institution.name object must be bound to the institution component of the recently created person object. This can be achieved, for example, in the following way:

```
contains( <<person_name>>[!.person[NEW].name[NEW] := @0]
    <<institution_name>>
        [!.person[LAST].institution[NEW] := @0]? );
```

The global reference operator, combined with the NEW selector, allows for the creation of
any kind of object. The LAST selector can then reference the recently created object, which can be updated. (The sequence of operations is relevant for bindings.)

Non-textual objects can combine textual objects recognized at different locations of the input document into a higher level object. Non-textual objects may be the best candidates to be stored into an object database. Objects like persons, institutions, journals, etc., may be composed of information extracted from different places within a document and then, after all the information is combined together in these abstract objects, they could be saved to the permanent object database.

4.4.10 String Component Descriptions

In Section 4.4.7 we mentioned that, although string components used in regular expressions should be enclosed within two precise operands (units or literal strings), this restriction can be relaxed if a description of the values that can be bound to the string component being used is provided. This description can be expressed by means of the same kinds of expressions that are used to describe objects. The syntax of these descriptions is defined by the rule presented in Figure 4.25.

For example, we can define university names as being composed of one or more university indicators, which are words like "university", or "institute", and by some other words, that can appear before or after the indicators. The university name indicators can be identified easily. But we do not know what the other words may be. For this reason, we use string components to represent them, and we define what kinds of values those string components can be bound to through a string component description. In Figure 4.26 we
CHAPTER 4. DELTO: LANGUAGE REFERENCE

present what could be a complete definition for university names.

The class definition for university names shown in Figure 4.26 specifies that university names always consist of an university indicator preceded or followed by a string, or preceded and followed by two different strings. The class definition describes in more detail those strings that may precede or follow the university indicator. It specifies that those strings frequently contain capitalized words, the word "of", a state (such as "Virginia"), or even another university indicator. (There could be more than one university indicator in a university name, like in "Virginia Polytechnic Institute and State University", where the words "Polytechnic", "Institute", and "University" can all be considered university indicators.) The definition also specifies that those strings frequently begin and end with capitalized words. It also says that sometimes the words "and" or "the" may be contained in those strings, and that there can never be a sequence of two or more uncapitalized words within those strings. Given this description, the DELTO Analyzer can identify university indicators and then examine the surroundings of the identified university indicators trying to find the boundaries of a university name. The information given in the string component description is used in the process of identifying those boundaries.

String components used as operands in regular expressions must be bound to a non-empty string. That is why in the example of Figure 4.26 three pattern predicates are connected by an or logical operator. The three alternatives are needed. A string like "University of Washington" would not match either of the first two alternatives (assuming "University" is the university_indicator) because the start_part component would not be matching any string. The last alternative is the one that would match that string.

4.4.11 Object Database Interface Description

A representation of all persistent objects identified by the DELTO System is stored into a permanent object database. We call these representations stored in the object database ODB objects, as opposed to DELTO objects (the objects identified by the DELTO System).
CLASS DEFINITION: University_Name
 COMPONENTS
 {
   start_part : STRING 0..1;
   end_part : STRING 0..1;
 }

DESCRIPTION
 {
   ALWAYS
   {
     pattern( start_part[NEW]
       <university_indicator>>
       end_part[NEW] )
     or
     pattern( start_part[NEW] <university_indicator>> )
     or
     pattern( <university_indicator>> end_part[NEW] );
     in( <sentence> );
   }
   FREQUENTLY
   {
     in( <front_matter> );
   }
 }

STRING COMPONENT DESCRIPTION FOR: start_part, end_part
 {
   FREQUENTLY
   {
     contains( <word, case=$begins_upper or case=$all_upper> | 
     <word, text = "of"> | 
     <state> | <university_indicator> );
     begins( <word, case=$begins_upper or case=$all_upper> );
     ends ( <word, case=$begins_upper or case=$all_upper> );
   }
   SOMETIMES
   {
     contains( <word, text = "and" or text = "the" );
   }
   NEVER
   {
     contains( <word, case=$lower>2..N );
   }
 }

ENDDEF;

Figure 4.26: A class definition for university_name

86
CHAPTER 4. DELTO: LANGUAGE REFERENCE

For a DELTO object there may or may not be a previously existing ODB object. For example, the author of an article may be identified as a person object by the DELTO System. This person may be the author of other documents previously processed by the DELTO System, in which case there should be a previously existing ODB object for that person. Given that situation, it would be desirable that the DELTO System recognizes the previous existence of an ODB object associated with the identified person object and, instead of creating a new ODB object, retrieves the existing one and updates it.

An object database interface description specifies two things: how to recognize a potential previously existing ODB object associated with a DELTO object, and what information to store in the ODB object (whether newly created or previously existing).

Potential previously existing ODB objects are identified through access keys. Further tests to verify that an ODB object identified through one or more access keys is really associated with the DELTO object being processed can be specified through a match description. From all the potential ODB objects that are identified by the access keys, if any, and that are able to “survive” the tests imposed by the match description, the one that obtains the heaviest weight is selected as the ODB object associated with the DELTO object, as long as the weight is acceptable. If no ODB object is identified for a particular DELTO object being processed, a new one is created.

Once an ODB object is associated with a DELTO object, the sequence of actions specified in the object database interface of the class definition is executed. These actions normally add values to fields of the ODB object (based on the values of attributes or components of the DELTO object), update these field values, link the ODB object to other ODB objects, or index the ODB object on some particular strings, so that they can be identified in the future by other DELTO objects extracted from other documents.

The syntax for object database interface descriptions is given by the rule presented in Figure 4.27.

All expressions included in an object database interface description are evaluated under a chain of scoping environments containing only one environment: the one associated with
CHAPTER 4. DELTO: LANGUAGE REFERENCE

\[
\textit{<odb\_interface>} \quad ::= \quad \textit{ODB\_INTERFACE} \\
\quad \quad \quad \quad ,\{ \quad \textit{<access\_keys\_block> } \} \\
\quad \quad \quad \quad [ \quad \textit{<match\_desc\_block> } ] \\
\quad \quad \quad \quad [ \quad \textit{<actions\_block> } ] \\
\quad \quad \quad \quad \}, '
\]

Figure 4.27: Syntax rule for object database interface descriptions

the persistent DELTO object that is being stored into the object database.

The object database used by the DELTO System in combination with a particular format description must have definitions for all the DELTO persistent classes that are defined in the format description. (ODB classes are defined using the ODB System.) The names of the classes must be the same. In addition, all the components and attributes defined for a DELTO class must be defined as fields in the ODB class. These fields can be defined as either of type string or of type link. The same names must be used for defining these fields. (If there is a component named address defined for a DELTO class, then there must be a field named in the same way, defined for the associated ODB class.)

Within the match description and the actions block of an object database interface description there is a new (automatically defined) component available for the DELTO object: the ODB component. The ODB component represents the ODB object associated with the DELTO object. All the components and attributes defined for the DELTO object also are defined and available under the ODB component. For example, assuming the definition for bibliographic entry objects given in Figure 4.7, the following are valid references within the match description or the actions block of the class definition for bibliographic entry objects:

\[
\textit{ODB\_author} \\
\textit{ODB\_title} \\
\textit{author\_ODB\_author\_of}
\]

The first two examples reference the ODB object associated with the bibliographic entry DELTO object. Binding the component \texttt{ODB.author} to a value actually means adding a
CHAPTER 4. DELTO: LANGUAGE REFERENCE

value to the field named author of the ODB object associated to the bibliographic entry
DELTO object. The last example is referencing the ODB object associated with the DELTO
object that is bound to the author component of the bibliographic entry DELTO object.
For example, it would be possible to build a link (in terms of ODB objects) between the
components ODB.author and author.ODB.author.of. This would link together the ODB
bibliographic entry object with the ODB person object that represents the author of the
document described by the bibliographic entry.

Only persistent objects have an automatically defined ODB component and, as we have
already said, this component (and all its subcomponents) are visible only within the match
descriptions and action blocks of object database interface descriptions.

Statistical components are also available under the ODB component. For example, we
may obtain the number of values already bound to the author.of field of an ODB object
by specifying the reference ODB.#author.of.

ODB fields can be bound, in the same way as DELTO object components, to any number
of values (as long as this number is within the range specified for the field in the ODB class
definition).

Access Keys

There is no limit on the number of access keys that may be tried by the DELTO System
for each persistent object. Access keys are formed by combining strings originated from
attributes or components of type STRING of the DELTO object.

Access keys are defined within an access keys block. The syntax for access keys blocks
is defined by the grammatical rules presented in Figure 4.28.

In Figure 4.29 we present an example that shows what could be the components and
access keys definition for a person name object.

Note the use of the concat and initial functions, in the example of Figure 4.29.

Three different access keys can be generated from our example, though the keys gener-
erated by the expressions are the same each time that there are no values bound to the
CHAPTER 4. DELTO: LANGUAGE REFERENCE

\[
\begin{align*}
\langle \text{access.keys.block} \rangle & \ ::= \ \text{ACCESS KEYS} \\
& \quad \{' \}
\quad \langle \text{key} \rangle \ \{ ; \ \langle \text{key} \rangle \} \\
& \quad \'}
\end{align*}
\]

\[
\langle \text{key} \rangle \ ::= \ ( \ \langle \text{component} \rangle \ | \ \langle \text{attribute} \rangle \ | \ \langle \text{string.funct} \rangle \ ) \\
& \quad \ [ \ \langle \text{strength.indic} \rangle \ ]
\]

Figure 4.28: Syntax rules for access keys

COMPONENTS
{
    given_name : given_name 0..2;
    last_name : last_name 1..2;
    institution : institution_name 0..1;
    email : STRING 0..1;
}

DDB-INTERFACE
{
    ACCESS KEYS
    
    i
    \begin{align*}
    \text{last_name[i].text} + "," + \text{concat( given_name[EACH].text )}; \\
    \text{last_name[i].text} + "," + \text{concat( initial( given_name[EACH].text ) )}; \\
    \text{last_name[i].text};
    \end{align*}
}

Figure 4.29: Access keys definition
CHAPTER 4. DELTO: LANGUAGE REFERENCE

given_name component. For a name like John Peter Smith the following are the keys that our definition generates (keys are case insensitive):

- Smith, John Peter
- Smith, J. P.
- Smith

Once all the keys are generated, the DELTO System retrieves all the ODB objects that are indexed on any of the generated keys. It may happen that more than one ODB object is retrieved. For example, it is likely that the key "Smith" will retrieve many ODB objects. The DELTO System attempts then, based on the information given on the object database interface description, to identify the correct ODB object to be associated with the DELTO object being processed, if any. If it cannot identify any, then it creates a new ODB object.

In the same way as description expressions, keys can also be associated with a strength indicator. ODB objects retrieved by some keys may be more likely to be the “correct” ODB object than ODB objects retrieved by other keys. In the example of Figure 4.29, objects retrieved from the keys generated by the first two expressions are much more likely to be the correct object than those that are retrieved from the key generated by the last expression. The same strength indicators that were used for descriptions can be used in combination with access keys. If no strength indicator is explicitly associated with a key, the moderate indicator is associated with the key by the DELTO System. We can now rewrite our previous example using strength indicators, as is shown in Figure 4.30.

The DELTO System associates each retrieved ODB object with a weight that is based on the strength indicators of the keys that are indexes to the ODB object. The more keys that index a particular ODB object, the higher the weight for that ODB object is. If no match description is given in the object database interface description, this is the only information that the DELTO System can use to select the best ODB object. The ODB object with the highest weight is selected, as long as this weight is considered acceptable (“beyond reasonable doubts”) by the DELTO System.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

ACCESS KEYS
{
    last_name[1].text + ", " + concat( given_name[EACH].text )
    VERY-STRONG;

    last_name[1].text + ", " + 
    concat( initial( given_name[EACH].text ) )
    STRONG;

    last_name[1].text
    WEAK;
}

Figure 4.30: Access keys definition with strength indicators

Match Description

We can provide the DELTO System with additional information, not just access key strength indicators, in helping it select the best ODB object for a DELTO Object. A match description specifies expressions that are evaluated for each candidate ODB object. The same strength and frequency indicators that we described in Section 4.4.3 can be used within match descriptions. In this way we can specify properties that must be satisfied by ODB objects to be selected, properties that should be satisfied but do not eliminate an ODB object from consideration if they are not, and properties that provide the best indication, if satisfied, that an ODB object is the “correct one”.

Within match descriptions we have to use references to the ODB component (and to its subcomponents) of the DELTO object being processed to access the values bound to the fields of the candidates ODB objects. The match description is going to be applied once to each candidate object. Each time it is applied the ODB component represents the ODB object over which the match description is being applied.

For example, for the person objects defined in Figure 4.29, we could specify the match description given in Figure 4.31.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

MATCH-DESC
{
    ALWAYS
    {
        email = ODB.email;
    }
    FREQUENTLY
    {
        similar-to( institution.name, ODB.institution.name, 80 );
    }
}

Figure 4.31: Match description example

<match_desc.block> ::= MATCH-DESC
    '{'
        { <frequency_block> }
    '}'

Figure 4.32: Syntax rule for match descriptions

If several candidate ODB objects are retrieved from the generated access keys, then each of the expressions specified in the match description is applied to each of the candidate ODB objects. With respect to the example given in Figure 4.31, all those objects that do not satisfy the always expression are eliminated as potential candidates. If one or more do satisfy this expression, the second expression, associated with a frequently indicator, is applied to them. From the result of evaluating this expression, and from the weights previously assigned on the basis of the keys that were used to retrieve the objects, a new weight is computed for the remaining candidate ODB objects. Again, the DELTO System selects the one with the highest weight, as long as this weight is acceptable.

The syntax for match descriptions is given by the rule presented in Figure 4.32.

93
CHAPTER 4. DELTO: LANGUAGE REFERENCE

<action_block> ::= '{
    { odb_action } | <if>
'}

<odb_action> ::= odb_action_name
    '(' odb_action_arg {, odb_action_arg} ')' 

<odb_action_arg> ::= attribute | component | function | literal | integer

<odb_action_name> ::= ODB_ADD_VALUE | ODB_UPDATE_VALUE | ODB_ADD_LINK | ODB_INDEX_ON

<if> ::= IF '(' <expression> ')' <action_block>

Figure 4.33: Syntax rules for action blocks

Actions

Once the DELTO System has associated an ODB object (whether new or previously existent) to a DELTO object, the selected ODB object can be updated with information extracted from the DELTO object. Within the actions block the ODB component of the DELTO object being processed always refers to the selected ODB object. Updating this component results in the update of the corresponding ODB object. There are four different kinds of actions that can be performed over an ODB object:

- Add a field value
- Update a field value
- Add a link from a field to a field of another ODB object
- Index the ODB object on a specified string

Each of the 4 mentioned actions can be accomplished by a particular ODB action. These actions are described below. The syntax rules for actions blocks are given in Figure 4.33.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

The following is the syntax for the arguments required by the \texttt{odb.add.value} action:

\begin{verbatim}
odb.add.value '(' \texttt{<odb.action.arg>, <odb.action.arg> '})
\end{verbatim}

This action is used to add new values to an ODB object field. The first argument must indicate the ODB object field to which new values are added, which means that it must include a reference to an ODB component. The second argument specifies the value, or values, that are to be added. It must be of type string or numeric (in the case of numeric values, they are automatically converted to strings before they are provided as arguments to this action). If multiple values are returned by the second argument, all these values are added as different new values of the indicated ODB object field. The following is an example of the use of this action:

\begin{verbatim}
odb.add.value( ODB.Title, Title );
\end{verbatim}

We now present the syntax of the arguments needed by the \texttt{odb.update.value} action:

\begin{verbatim}
odb.update.value '(' \texttt{<odb.action.arg>, <odb.action.arg>, integer '})
\end{verbatim}

This action updates existing field values of an ODB object. In the same way as with the \texttt{odb.add.value} action, the first argument indicates the ODB object field to be updated and the second argument specifies the new value to assign to the specified field. (If multiple values are represented by this argument, only the last one is considered.) The third argument specifies the specific value of the ODB field indicated by the first argument that is to be updated. For example:

\begin{verbatim}
odb.update.value( ODB.Name, Name, 1 );
\end{verbatim}

The syntax of \texttt{odb.add.link} is given by the following description:
CHAPTER 4. DELTO: LANGUAGE REFERENCE

odb_add_link (’, <odb_action_arg>, <odb_action_arg> ’)

This action establishes a link between two fields located within two different ODB objects. Both arguments must reference ODB components and the referenced fields must have been defined of link type within their ODB class definitions. (Usually ODB fields of type link are associated with DELTO object components that have the type of a persistent object.) If multiple values are referenced by any of the arguments the links are established among all the referenced values. The following is an example of the usage of this action:

odb_add_link (ODB.Author, Author[EACH].ODB.Author_of);

In the example, if the second argument returns multiple values, a link is established from the ODB.Author field to all the ODB object fields referenced by the second argument. Assuming the bibliographic entry object definition given in Figure 4.7, the example is building links from a bibliographic entry object to all the authors of the document represented by the bibliographic entry.

Finally, we describe the odb_index_on action. Its syntax is the following:

odb_add_link (’, <odb_action_arg> ’)

This action indexes the ODB object associated with the DELTO object being processed. After an ODB object is indexed it can be retrieved at a later point, when a DELTO object that may be associated with the indexed ODB object is encountered. For this to happen, there must be consistency between the specified access keys and the way an ODB object is indexed. In Figure 4.29 we gave an example consisting of access keys for person objects. We should index each new ODB object that is created in the way pictured in Figure 4.34, so that the described access keys can succeed in retrieving ODB objects.

Conditional statements can be used within action blocks. In this way decisions can be taken whether to perform or not certain actions. All valid predicates described in Section
CHAPTER 4. DELTO: LANGUAGE REFERENCE

odb_index_on( Last_Name.text + "," +
    Concat(Given_Name[EACH].text) );
odb_index_on( Last_Name.text + "," +
    Concat(Initial(Given_Name[EACH].text) );
odb_index_on( Last_Name.text );

Figure 4.34: Actions that index an ODB object

4.4.4 can be used within conditional test expressions. If the test expression of a conditional statement succeeds, then the actions guarded by the conditional statement are executed. Two additional predicates can be used within conditional test expressions:

• found()

• manually_entered()

The first of these predicates, found(), succeeds when a previously existing ODB object has been encountered for the DELTO object being processed. The second predicate succeeds when the ODB object associated with a DELTO object has been manually created through the ODB System, as opposed to automatically created by the DELTO System.

In Figure 4.35 we present a complete object database interface description for the bibliographic entry objects defined in Figure 4.7.
CHAPTER 4. DELTO: LANGUAGE REFERENCE

ODB-INTERFACE
{
ACCESS-KEYS
{
    Title very-strong;
    Concat( Author[EACH].Name ) moderate;
}
MATCH-DESC
{
    ALWAYS
    {
        Similar-to( Title, ODB.Title, 75 );
        ODB.#Author = #Author;
    }
}
ACTIONS
{
    if( NOT FOUND() )
    {
        odb_add_link( ODB.Author,
                        Author[EACH].ODB.Author_of );
        odb_add_value( ODB.Title, Title );
        odb_add_value( ODB.Volume, Volume );
        odb_add_value( ODB.Issue, Issue );
        cdb_index_on( Title );
        odb_index_on( Concat( Author[EACH].Name ) );
    }
}
}

Figure 4.35: A complete ODB Interface

98
Chapter 5

RETARGET: LANGUAGE REFERENCE

5.1 Introduction

In this chapter we describe the syntax and semantics of Retarget. This language is used to specify how to translate the output generated by the DELTO Analyzer into a new document, based on the original one, but probably in another format, or incorporating information that was not present in the original document.

We name the specifications given in Retarget translation specifications. A translation specification is a collection of templates for different parts of the output document. There are blanks that can be left within these templates, together with a specification of how to fill them in using the information generated by the DELTO Analyzer after processing a document.

The formalism utilized to introduce the syntax of Retarget is the same one that was used to describe the syntax of DELTO. This formalism is described in Section 4.2. We introduce the terminal literal.text that represents any text that does not contain the '{' character, unless it is escaped.

A complete grammar for Retarget is presented in Appendix G. In Appendix D we show two complete Retarget specifications.

5.2 Templates

A template specifies how to generate a piece of text that represents appropriately, in the target format, an object generated by the DELTO Analyzer. During Retarget run-time some of the objects generated by the DELTO Analyzer are associated with a Retarget template.
CHAPTER 5. RETARGET: LANGUAGE REFERENCE

\[
\text{<translation_spec>} ::= \{ \text{<template>} \}
\]

\[
\text{<template>} ::= '\{ BEGIN <template\_name> ' }\ 
\{ \text{litera}l\text{.text} | \text{<statement>} \} \ 
'\{ END [ <template\_name> ] '\}
\]

\[
\text{<template\_name>} ::= \text{idem}
\]

Figure 5.1: Syntax rules for translation specifications and templates

The output generated by all template-object associations results in a new document. In particular, the first template within a translation specification is automatically associated with the object representing the entire input document processed by the DELTO Analyzer (the one that can be referenced using the DELTO `\! \cdot \text{document}[1]` construction). The execution of a Retarget translation always begins by making this association and starting to generate the output corresponding to this template-object association.

A template is composed of text that is copied literally into the output and of Retarget statements (which contain the blanks mentioned in the introduction of this chapter) that are not copied into the output but specify how the output should be generated.

Retarget statements are always enclosed within curly brackets. All text in a template not enclosed within curly brackets is considered text to be copied to the output. We present the syntax for translation specifications and for templates in Figure 5.1.

For example, the translation specification presented in Figure 5.2 always results, regardless of what the contents of the input document processed by the DELTO Analyzer are, in the output document pictured in Figure 5.3.

Note that the translation specification given in Figure 5.2 has only one template, named `document\_template`. This template, because it is the first one of the translation specification, is automatically associated with the DELTO object that represents the entire document. Once associated with this object, it is executed. However, there are no statements within the template, which means that the output generated is always the same, regardless of the DELTO object that is associated with it. The execution of this template
CHAPTER 5. RETARGET: LANGUAGE REFERENCE

{begin document_template}
<!doctype document SYSTEM "envision.dtd">
<DOCUMENT>
<ObjectID>1</ObjectID>
</DOCUMENT>
{end document_template}

Figure 5.2: A translation specification with no statements

<!doctype document SYSTEM "envision.dtd">
<DOCUMENT>
<ObjectID>1</ObjectID>
</DOCUMENT>

Figure 5.3: Output generated by the translation specification of Figure 5.2

always results in the literal text that it includes copied into the output document.

5.3 Template Statements

Template statements are the constructions that allow a Retarget translation specification to result in actions that are more complex than just sequential copying of literal text. In particular, they allow the contents of the input document, processed by the DELTO Analyzer, to be used in the output document. There are a number of different statements that can be used. Attribute and component references allow for their values (to which they were bound by the DELTO Analyzer) to be copied to the output. Attribute and component references are described in Section 5.3.1. The process statement copies the text associated with an object to the output and, in addition, it allows to process pieces of text that represent other kinds of objects, and that are physically contained within the text of the object being processed, in different ways. This statement is described in Section 5.3.2. With the call statement, values bound to components of an object can be processed in more complex ways than simply being copied to the output. We describe the call statement in

161
CHAPTER 5. RETARGET: LANGUAGE REFERENCE

<statement> ::= <component> | <process_st> | <call_st> | <if_st> |
             <for_st> | <assignment_st>

Figure 5.4: Syntax rules for template statements

{begin document_template}
<!doctype document SYSTEM "envision.dtd">
<DOCUMENT>
<ObjectID>1</OBJECTID>
{text}
</DOCUMENT>
{end document_template}

Figure 5.5: A translation specification that includes a component reference

Section 5.3.3. The if and the for statements allow for conditional and iterative execution of parts of a template, respectively. The if statement is presented in Section 5.3.4 while the for statement is explained in Section 5.3.5. Retarget also permits assignment statements which may alter the values of a number of global variables that control different aspects of the way in which output must be generated. Assignments are described in Section 5.3.6. The high-level syntax for template statements is given in Figure 5.4.

5.3.1 Attribute and Component References

All the attributes and components of the objects associated with a template can be referenced. For Retarget it is irrelevant if a reference is made to an attribute or to a component. For this reason, we shall refer from now on exclusively to components.

A reference to a component value results in its value (which must be a string or a number) added to the output document. For example, we could introduce a little modification to the translation specification given in Figure 5.2 that would result in a big difference in the output that it generates. The modified specification is presented in Figure 5.5.

We have added a reference to the text component of the objects associated with this
CHAPTER 5. RETARGET: LANGUAGE REFERENCE

We are here to describe the characteristics of this new document analysis and translation system developed by people like Bill Clinton and Newt Gingrich.

Figure 5.6: An input document

<!doctype document SYSTEM "envisio.n.dtd">
<Document>
<ObjectID>1</ObjectID>
We are here to describe the characteristics of this new document analysis and translation system developed by people like Bill Clinton and Newt Gingrich.
</Document>

Figure 5.7: Output generated from the document of Figure 5.6 using the specification of Figure 5.5

template. As we have already said, this template is automatically associated with the DELTO object that represents the entire document. This means that the referenced text component represents the text of the input document. This translation specification results, then, in a document consisting of the unaltered text of the input document enclosed by the literal text given in the template. For example, for the input document pictured in Figure 5.6 a new document like the one shown in Figure 5.7 would be generated.

References can be qualified in order to extract the values of subcomponents. Selectors can also be included after each indicated component name to reference a specific value of a component. If a selector is omitted, the reference is assumed to be to the first value. However, selectors are much simpler than the ones allowed within DELTO format descriptions. Retarget selectors can only be positive integers or references to index variables, which are described in Section 5.3.5 of this chapter. The syntax for component references within Retarget is given in Figure 5.8.

References to statistical attributes can be made in the same way as within DELTO format descriptions. A new component that is accessible only through Retarget specifica-
CHAPTER 5. RETARGET: LANGUAGE REFERENCE

```plaintext
<component> ::= '{' <component_name> [ <selector> ]
               { . <component_name> [ <selector> ]} '}'

<component_name> ::= ['#'] ident

<selector> ::= ['(' integer | <index_var> ')']

<index_var> ::= ident
```

Figure 5.8: Syntax for component references

```xml
{begin document_template}
<!doctype document SYSTEM "envision.dtd">
<DOCUMENT>
<ObjectID>{odbid}</OBJECTID>
{text}
</DOCUMENT>
{end document_template}
```

Figure 5.9: A translation specification that references odbid components

...ions, and that is only available for DELTO persistent objects, is the odbid component. This component is bound to the unique ID of the ODB object associated with a DELTO persistent object. We could then introduce a small change into our last example that would result in the document ODB object id included in between the OBJECTID SGML tags of the output document. The modified translation specification is given in Figure 5.9.

5.3.2 Process Statement

The process statement has the same effect as referencing the text component of the object associated to the template: it copies the text associated to this object into the output. In addition, it allows for more sophisticated processing. The syntax for this statement is defined in Figure 5.10.

We can specify a series of DELTO classes to the process statement. It is required that we provide for each of the specified classes the name of a template (that has to be defined...
CHAPTER 5. RETARGET: LANGUAGE REFERENCE

\[ \text{process \_st} ::= \{\}
\text{PROCESS} \{'\text{template \_sel}'} {, \text{template \_sel}} \} \}' [;]
\text{IGNORE \_TEXT} \{'\text{class \_name}'} {, \text{class \_name}} \'} [;]
\}
\text{template \_sel} ::= \text{class \_name} [\text{rel \_op} \text{integer}] \text{WITH} \text{template \_name}

Figure 5.10: Syntax rules for the process statement

in the same translation specification). If classes are specified, process copies text up to the point in which a piece of text (contained in its entirety within the one represented by the object being processed) is represented by an object of any of the classes specified in the process statement. At this point process gives control to the template that was indicated for the class of the object that was found, and associates the template with this object for immediate execution. Control is returned to the process statement after the indicated template finishes its execution and at the point within the text of the input document where the text represented by the object that activated the template ends. At this point process continues to copy text into the output until it finds another piece of text represented by an object of any of the classes that were specified to it and repeats the process that has just been described.

For example, we may want to enclose every person \_name that the DELTO Analyzer identifies within a document between two particular SGML tags. To achieve this goal we can complete our example translation specification in the way illustrated by Figure 5.11.

Given the translation description of Figure 5.11, and assuming that the DELTO Analyzer is able to identify the two person \_names contained in our example input document and that the ID of the ODB object associated to the DELTO document object is 782, what would be the resulting output document is shown in Figure 5.12.

It is also possible to specify a condition over the weight of the objects that activate other templates. For example, we may want to process with the person \_name \_template only those person \_name DELTO objects that have a weight greater than 70. This can be expressed in the following way:
CHAPTER 5. RETARGET: LANGUAGE REFERENCE

{begin document_template}
<!doctype document SYSTEM "envision.dtd">
<DOCUMENT>
<OBJECTID>{odbid}</OBJECTID>
{process(person_name with person_name_template )}
</DOCUMENT>
{end document_template}

{begin person_name_template}
<P_NAME>{text}</P_NAME>
{end person_name_template}

Figure 5.11: A translation specification that uses the process statement

<!doctype document SYSTEM "envision.dtd">
<DOCUMENT>
<OBJECTID>782</OBJECTID>
We are here to describe the characteristics of this new document analysis and translation system developed by people like <P_NAME>Bill Clinton</P_NAME> and <P_NAME>Newt Gingrich</P_NAME>.
</DOCUMENT>

Figure 5.12: Output generated from the document of Figure 5.6 using the specification of Figure 5.11
CHAPTER 5. RETARGET: LANGUAGE REFERENCE

{begin document_template}
<!doctype document SYSTEM "envision.dtd">
<DOCUMENT>
  <OBJECTID>{odbid}</OBJECTID>
  {process()}
    ignore_text(person_name)}
  </DOCUMENT>
{end document_template}

{begin person_name_template}
  <P_NAME>{text}</P_NAME>
{end person_name_template}

Figure 5.13: A translation specification that uses the process-ignore statement

<!doctype document SYSTEM "envision.dtd">
<DOCUMENT>
  <OBJECTID>782</OBJECTID>
  We are here to describe the characteristics of this new document
  analysis and translation system developed by people like
  and .
</DOCUMENT>

Figure 5.14: Output generated from the document of Figure 5.6 using the specification of
Figure 5.13

{process(person_name > 70 with person_name_template )}

Finally, the process statement also can ignore the text (and consequently not copy it to
the output) represented by objects of specific classes. This can be achieved by including an
ignore_text statement immediately after the process statement, and between the same
pair of curly brackets. If the translation description given in Figure 5.13 is used, then the
resulting output would be that of Figure 5.14.

107
CHAPTER 5. RETARGET: LANGUAGE REFERENCE

\[
\text{call_st} ::= \text{'}\text{CALL}\ <\text{template_name}>\ \text{'}\text{'}\ \text{'}\text{component}\ \text{'}\text{'}\ [;]\ \text{'}\text{'}
\]

Figure 5.15: Syntax rule for call statements

{begin document_template}
<!doctype document SYSTEM "envision.dtd">
<DOCUMENT>
<OBJECTID>{odbid}</OBJECTID>
{call publication_template(publication)}
{text}
</DOCUMENT>
{end document_template}

{begin publication_template}
<JOURNAL>{name}</JOURNAL>
<PUBLISHER>{publisher_name}</PUBLISHER>
{end publication_template}

Figure 5.16: A translation specification that uses the call statement

5.3.3 Call Statement

The call statement activates a template and associates this template with an object indicated by the statement. The object is indicated through a reference to a component value, which must be of the type of an object. Once the activated template finishes execution, control returns to the point that follows the call statement within the original template. The syntax for call statements is given in Figure 5.15.

For example, we could write the translation specification given in Figure 5.16. Assume that the publication component is defined for the document DELTO class, that its type is of some object class that has the components name and publisher_name, and that these components are bound to appropriate values by the DELTO Analyzer. Given the modified version of our input document presented in Figure 5.17, the generated output would be that of Figure 5.18.
We are here to describe the characteristics of this new document analysis and translation system developed by people like Bill Clinton and Newt Gingrich.

Journal of Politicians (Politicians Inc.)

Figure 5.17: A modified input document

<!doctype document SYSTEM "envision.dtd">
<DOCUMENT>
<ObjectID>782</OBJECTID>
<JOURNAL>Journal of Politicians</JOURNAL>
<PUBLISHER>Politicians Inc.</PUBLISHER>
We are here to describe the characteristics of this new document analysis and translation system developed by people like Bill Clinton and Newt Gingrich.

Journal of Politicians (Politicians Inc.)

Figure 5.18: Output generated from the document of Figure 5.17 using the specification of Figure 5.16

109
5.4 If Statement

The if statement allows for the conditional execution of certain parts of a template, based upon the result of a test expression. The test expression can reference any component of the object associated with the template. However, Retarget test expressions are very simple: they can only compare a numeric component with an integer constant and they cannot include logical operators. The syntax for the if statement is presented in Figure 5.19.

For example, we could use an if expression to generate different SGML constructions based upon the value bound to a component of the object associated with the template, as shown in Figure 5.20.

Note that we do not indent the template in order to make it more readable because the white space of the indentation would be considered literal text and, therefore, it would be copied into the output. Line breaks immediately following a statement are not copied to
CHAPTER 5. RETARGET: LANGUAGE REFERENCE

\[ \text{<for_stmt> ::= '{' FOR <index_var> := integer TO <component> '}'\]
\[\{\text{literal_text} \text{ <statement> }\}\]
\[\{' \text{ ENDFOR } '{\} \]

\[<\text{index_var}> ::= \text{ident}\]

Figure 5.21: Syntax rules for for statements

{for i := 1 to #author}
<AUTHOR>{author[i].name}</AUTHOR>
{endfor}

Figure 5.22: Use of the for statement

the output.

5.3.5 For Statement

We use the for statement to specify repeated execution of certain parts of a template. The Retarget for statement has the same semantics that the for construction of most programming languages has. The syntax of the for statement is given in Figure 5.21.

The index variable can be used only within component selectors, within the for statement that defines it. For example, we could use the piece of template pictured in Figure 5.22 to generate output for each of the authors of a document.

The piece of template shown in Figure 5.22 could result in the output of Figure 5.23, if the DELTO Analyzer recognizes four different authors for an input document.

We may combine the if and the for statements, as shown in Figure 5.24, to obtain the

<AUTHOR>Bill Clinton</AUTHOR>
<AUTHOR>George Bush</AUTHOR>
<AUTHOR>Ronald Reagan</AUTHOR>
<AUTHOR>Jimmie Carter</AUTHOR>

Figure 5.23: Output for the specification of Figure 5.22

111
CHAPTER 5. RETARGET: LANGUAGE REFERENCE

{if #author > 0}
  <AUTHOR>{author[1].name}</AUTHOR>
{for i := 2 to #author}
  and <AUTHOR>{author[i].name}</AUTHOR>
{endfor}
{endif}

Figure 5.24: Use of the if and for statements

<AUTHOR>Bill Clinton</AUTHOR>
and <AUTHOR>George Bush</AUTHOR>
and <AUTHOR>Ronald Reagan</AUTHOR>
and <AUTHOR>Jimmie Carter</AUTHOR>

Figure 5.25: Output for the specification of Figure 5.24

output given in Figure 5.25.

5.3.6 Assignment Statement

There are five global Retarget variables that control some aspects of the way in which the output is generated. All five variables have default values assigned by Retarget. These values can be changed whenever needed. The changes in these values affect all output that is generated until they are changed again. This means that changes to these values do not affect only the output generated by the template in which the changes are made but also any other templates, including the one that invoked the template where the change is made.

The syntax of assignment statements is presented in Figure 5.26.

Output_width variable

The output_width variable controls the width of the output lines that are generated. The default value for this variable is 80. An output line is broken when the first separator (blank, tab, or line break) is found after going over the value specified by this variable. This
CHAPTER 5. RETARGET: LANGUAGE REFERENCE

<assign.s> ::= '{' {<assignment> [;]} '}'

<assignment> ::= <global.var> := ( <integer> | <boolean> )

<global.var> ::= OUTPUT_WIDTH | IGNORE_LINE_BREACKS | IGNORE_SPACING | DEFAULT_SPACING | IGNORE_SGML_INPUT

<boolean> ::= TRUE | FALSE

Figure 5.26: Syntax rules for assignment statements

A variable can be set to a different value in the following way:

{output_width := 70}

Ignore_line_breaks variable

When the ignore.line_breaks variable is set to false (the default value), an output line is always broken when a line break is found in the input, even if the length of the current output line does not reach the value specified by the output_width variable. When the value of ignore_line_breaks is set to true, line breaks in the input are treated as if they were spaces. In this case line breaking is only controlled by the output_width variable. However, if two or more consecutive line breaks are found in the input then two line breaks are copied into the output. This variable can be set in the following way, for example:

{ignore_line_breaks := TRUE}

Ignore_spacing variable

The ignore_spacing variable indicates whether multiple consecutive spaces (blanks, tabs, or line breaks when ignore.line_breaks is set to true) should be all copied into the output, or if the number of spaces indicated by the default_spacing variable should be copied instead. If ignore_spacing is set to false all spaces are copied into the output. If it set to true all consecutive spaces are replaced by the number of blanks indicated by the default_spacing variable. The default value for this variable is false.
CHAPTER 5. RETARGET: LANGUAGE REFERENCE

Default spacing variable

The default_spacing variable is only relevant when ignore_spacing is set to true as was previously described. The default value for this variable is 1.

Ignore_sgml_input variable

When the ignore_sgml_input variable is set to true, SGML entities and tags originated from the input document are ignored and not copied to the output. This does not include SGML tags and entities generated from literal text contained in a template, which is always copied to the input regardless of the setting of this variable. The default value for the ignore_sgml_input variable is false.
Chapter 6

EXPERIMENTAL RESULTS

Up to this point, we have evaluated DELTO with documents in two different formats. We first evaluated DELTO with SGML bibliographic files (conforming to the Envision DTD). These SGML documents had been previously converted to SGML using the ad hoc tools mentioned in Section 1.2. Given that these documents are already in SGML, there is not much conversion to be done with them. The main goal that we have for these bibliographic documents is to load information about them into a permanent object database and to construct links among related documents. There is still a need to slightly update the input documents with anchor tags so that a portion of text that is abstracted by a permanent object can point to its associated object in the database. These portions of text can then become hot links when the documents are displayed.

We then evaluated DELTO with full text articles from various ACM journals represented in a pseudo-SGML format. Although already marked up, these documents do not have any information about their logical organization. The tags are only used to represent presentation information. For example, there are tags that indicate how much vertical or horizontal space must be left, but there are no tags that characterize a portion of a document as a chapter or as the front matter. These documents must be converted to SGML conforming to the Envision DTD to be incorporated into the Envision collection. We also want information about these documents to be incorporated into the object database, together with links to and from them.

The results with both formats are very promising. We have been able to load our object database with information about the documents we processed and also to generate correct SGML versions of them that conform to our DTD. In this chapter we describe what we
CHAPTER 6. EXPERIMENTAL RESULTS

have achieved with both kinds of documents. We utilize examples of input documents, objects in the database, and output documents to show our results. We also present some considerations regarding the results of our evaluation. In particular, we mention strengths and weaknesses that we discovered in DELTO during our evaluation process.

6.1 Format Descriptions and Translation Specifications

We developed three DELTO format descriptions for our evaluation process. These three format descriptions are included in Appendix C. The first one of them is a general description that contains definitions for the classes of objects that we expect to encounter in most document formats that we process. Among those classes we included the five classes of persistent objects (those objects that are stored in the permanent object database) that we are currently using for Envision: documents, persons, institutions, publications, and indexing terms. The definitions within the general format description specify components and attributes information for every class, and the object database interface for classes of persistent objects. We did not include expressions that describe the way that instances of the defined objects may appear within documents. These expressions should be specified within specific descriptions for each particular format.

For the five persistent classes mentioned above, we specified ODB classes. The definitions for these classes is given in Appendix E.

The second format description that we developed is oriented towards the SGML bibliographic documents. It uses the definitions specified in our general format description and incorporates information about the way in which the information is structured within the SGML bibliographic documents. Given that these documents already exhibit their logical structure, writing this description proved to be a very simple task. In general, it is very simple to write a format description that can adequately serve to process documents in any well-structured format.

Our last format description was developed for the pseudo-SGML full text articles. It also
CHAPTER 6. EXPERIMENTAL RESULTS

<table>
<thead>
<tr>
<th>Documents</th>
<th>Number</th>
<th>Total Size</th>
<th># Objects Created</th>
<th># Links Established</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bibliographic</td>
<td>396</td>
<td>19,461,363 Mb.</td>
<td>68,377</td>
<td>174,281</td>
</tr>
<tr>
<td>Full-Text</td>
<td>53</td>
<td>3,697,825 Mb.</td>
<td>3,412</td>
<td>4,368</td>
</tr>
</tbody>
</table>

Table 6.1: Statistics of processed documents

uses the information provided in our general description. This one proved to be the most difficult format description to write. As mentioned before, a document in this pseudo-SGML format is ill-structured and does not incorporate any information about its logical structure. The logical structure has to be extracted by the DELTO system from the presentation information. The format description that we wrote served this purpose.

We also wrote two Retarget translation descriptions: one for generating new SGML documents from the SGML bibliographic documents and the other one for creating proper SGML documents for the pseudo-SGML ones. The translation specification for the SGML bibliographic documents only adds information about links to the original documents. On the other hand, the translation description for the full text, pseudo-SGML articles restructures the input documents completely. The translation specifications for these documents, and the results that we obtained, proved the power behind the DELTO-Retarget combination and, in particular, the power of operating a translation process over a database representing the original document, as opposed to over a flat file from which strings are sequentially mapped into output strings. The Retarget specifications that we developed are presented in Appendix D.

Table 6.1 presents a summary of the documents that we have processed up to now using the DELTO format descriptions and Retarget specifications mentioned in this chapter. It also shows the number of objects and links that were created in the permanent object database.

We first describe, in Section 6.2, the results we achieved with the pseudo-SGML full text articles and then, in Section 6.3, the results we obtained with the SGML bibliographic documents. We chose this order to present our results because it makes it easier to understand the hypertext links that were constructed.
CHAPTER 6. EXPERIMENTAL RESULTS

6.2 Pseudo-SGML Full Text Articles

Figure 6.1 contains a portion of one of the full text articles in pseudo-SGML format that we processed. The portion of this document includes the front matter, with information about the title of the document, authors, and indexing terms. It then includes part of the body of the document, and finally a bibliographic reference. Note that the bibliographic reference is referenced in the short section of the body that has been included. Certain parts of the included portion of the document have been modified due to figure size constraints.

Given the document of Figure 6.1 and the wav.gral.dlt description presented in Appendix C, the DELTO system creates two objects of class document in the object database: one for the full-text article, and another one for the cited document. These two objects are linked with a cites-cited relationship. In Figure 6.2 we present a view of these two objects. The DELTO system also creates six person objects: three for each of the authors of the full-text article, and another three for each of the authors of the cited book. Each of the person objects is linked to the document that it is author of. Two of these objects are shown in Figure 6.3. Objects of type indexing_term also are created and linked to the object representing the full-text article. We show two of these objects in Figure 6.4. Note the IDs of the objects in the figures mentioned in this paragraph, and how those IDs appear in links at other objects.

The output document generated by the DELTO Translator using the translation specification given in Appendix D.2 is shown in Figure 6.5. This new document conforms to the Envision SGML DTD. Note the information about the journal and date in which the article has been published. It is located at the end of the input document and it has been moved to the front matter of the output document. This is a simple example of the freedom that we have to restructure a document as necessary. Note also the anchor tags, which enclose portions of text that are related to objects stored in the permanent database. For this purpose, the objectid attribute of those tags indicates the ID of the object that a portion of text is related to. Anchor tags may also indicate a relationship within the text of the
CHAPTER 6. EXPERIMENTAL RESULTS

Engineering a Simple, Efficient Code-Generator Generator

CHRISTOPHER W. FRASER

AT&T Bell Laboratories

DAVID R. HANSON

Princeton University

TODD A. PROEBSTING

The University of Arizona

Categories and Subject Descriptors: D.3.4 [Programming Languages]: Compilers and language processing; code generation; translator writing systems and compiler generators

General Terms: Languages

Additional Key Words and Phrases: Code generation, code-generator generator, dynamic programming, Icon programming language, tree pattern matching

INTRODUCTION

Many code-generator generators use tree pattern matching and dynamic programming (DP) [4]. They accept tree patterns and associated costs, and semantic actions that, for example, allocate registers and emit object code. They produce tree matchers that make two passes over each subject tree.

REFERENCES


Figure 6.1: A portion of a Pseudo-SGML full text article
<table>
<thead>
<tr>
<th>Object ID:</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Class ID:</td>
<td>dc</td>
</tr>
<tr>
<td>Stored in Object File #:</td>
<td>1</td>
</tr>
<tr>
<td>title[1]</td>
<td>Engineering a Simple, Efficient Code-Generator Generator</td>
</tr>
<tr>
<td>item_type[1]</td>
<td>JrnArt</td>
</tr>
<tr>
<td>author[1]</td>
<td>5-&gt;author_of</td>
</tr>
<tr>
<td>author[2]</td>
<td>6-&gt;author_of</td>
</tr>
<tr>
<td>author[3]</td>
<td>7-&gt;author_of</td>
</tr>
<tr>
<td>published_in[1]</td>
<td>g-&gt;document</td>
</tr>
<tr>
<td>date[1]</td>
<td>September, 1992</td>
</tr>
<tr>
<td>volume[1]</td>
<td>1</td>
</tr>
<tr>
<td>number[1]</td>
<td>3</td>
</tr>
<tr>
<td>pp[1]</td>
<td>213 226</td>
</tr>
<tr>
<td>crcat[1]</td>
<td>h-&gt;document</td>
</tr>
<tr>
<td>crcat[2]</td>
<td>i-&gt;document</td>
</tr>
<tr>
<td>term[1]</td>
<td>l-&gt;document</td>
</tr>
<tr>
<td>keyword[1]</td>
<td>m-&gt;document</td>
</tr>
<tr>
<td>cites[1]</td>
<td>3-&gt;cites</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Object ID:</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object Class ID:</td>
<td>dc</td>
</tr>
<tr>
<td>Stored in Object File #:</td>
<td>1</td>
</tr>
<tr>
<td>author[1]</td>
<td>8-&gt;author_of</td>
</tr>
<tr>
<td>author[2]</td>
<td>a-&gt;author_of</td>
</tr>
<tr>
<td>author[3]</td>
<td>b-&gt;author_of</td>
</tr>
<tr>
<td>cited_in[1]</td>
<td>i-&gt;cites</td>
</tr>
</tbody>
</table>

Figure 6.2: Document objects created for the full-text article
CHAPTER 6. EXPERIMENTAL RESULTS

Object ID: 5
Object Class ID: pr
Stored in Object File #: 1

name[1] = FRASER, CHRISTOPHER W.
author_of[1] = 1->author

Object ID: 8
Object Class ID: pr
Stored in Object File #: 1

name[1] = Aho, A. V.
author_of[1] = 3->author

Figure 6.3: Person objects created for the full-text article

Object ID: j
Object Class ID: it
Stored in Object File #: 1

term[1] = compilers
document[1] = 1->crca
term[1] = Code generation
document[1] = 1->keyword

Figure 6.4: Indexing.Term objects created for the full-text article
CHAPTER 6. EXPERIMENTAL RESULTS

document itself. Note, for example, the anchor tag enclosing the citation within the body of the article. In this case the attribute is called indocid and it is bound to a value that represents the ID of another SGML tag within the same document. The indocid attribute of that anchor relates the citation with the bibliographic reference in the same document.

6.3 SGML Bibliographic Documents

In Figure 6.6 we show three bibliographic entries extracted from the SGML bibliographic documents. As opposed to the pseudo-SGML document, every piece of these bibliographic entries is tagged according to its meaning.

The DELTO system, given the bibs.dtd format description (presented in Appendix C), generates three document objects for the bibliographic entries of Figure 6.6, one for each entry. These three objects are shown in Figure 6.7. Each object has links to its authors. In this case, the same two persons are the authors of the three documents. Note that the object IDs of the person objects that the document objects are linked to, are the same. These two person objects are given in Figure 6.8. Note that these two persons were also the authors of one of the documents for which an object was created when the DELTO system processed the pseudo-SGML document. The DELTO system does not create new objects for these two persons, but it rather updates the two previously existing objects, adding links to the new documents that the persons are authors of.

The resulting output bibliographic entries, generated by the DELTO translator based upon the Retarget specification given in Appendix D.1 are presented in Figure 6.9. Note that, in this case, the only differences between the input and output are the presence in the output of the anchor tags, and the IDs of each of the entries, which in the output represent the ID of the ODB document object that they are associated with.
CHAPTER 6. EXPERIMENTAL RESULTS

Many code-generator generators use tree pattern matching and dynamic programming (DP). They accept tree patterns and associated costs, and semantic actions that, for example, allocate registers and emit object code. They produce tree matchers that make two passes over each subject tree.

Figure 6.5: Full text article translated to SGML conforming to the Envision DTD
CHAPTER 6. EXPERIMENTAL RESULTS

Figures 6.6: Three SGML bibliographic entries
CHAPTER 6. EXPERIMENTAL RESULTS

---

Object ID: s
Object Class ID: dc
Stored in Object File #: 1

   Parsing
item_type[1] = Book
author[1] = 8->author_of
author[2] = 7->author_of
series[1] = Series in Automatic Computation
date[1] = 1972

---

Object ID: t
Object Class ID: dc
Stored in Object File #: 1

   Compiling
item_type[1] = Book
author[1] = 8->author_of
author[2] = 7->author_of
series[1] = Series in Automatic Computation
date[1] = 1973

---

Object ID: u
Object Class ID: dc
Stored in Object File #: 1

title[1] = Principles of Compiler Design
item_type[1] = Book
author[1] = 8->author_of
author[2] = 7->author_of
date[1] = 1977
keyword[1] = x->document

---

Figure 6.7: Document objects created for the SGML bibliographic entries

125
Figure 6.8: Person objects created for the SGML bibliographic entries
CHAPTER 6. EXPERIMENTAL RESULTS

Figure 6.9: Resulting output for the SGML bibliographic entries
CHAPTER 6. EXPERIMENTAL RESULTS

6.4 Strengths and Weaknesses

The evaluation process showed that we were able to meet the goals that we set in Section 1.3. That is, we were able to convert textual documents to SGML (and in addition, to any other formats), extract information about objects from those documents, store that information in a permanent database, and construct hypertext links between those objects. All these tasks are specified through high-level languages, as it was our goal, and they are accomplished by a completely automatic system that does not require any human intervention.

Writing DELTO format descriptions for the SGML bibliographic entries proved to be a very easy and natural process. Given that these documents already contain information about their logical organization, DELTO format descriptions can describe that organization very naturally. Given the format description that we wrote, the DELTO System was able to successfully extract information from those documents, store that information in our permanent database, and make links between related objects. In general, we believe that writing a DELTO format description for well-structured documents is a very simple task for someone who understands the structure of the documents.

On the other hand, writing the format description for the ill-structured pseudo-SGML full text articles was not so easy. We still were able to write a good description that is able to generate, for a pseudo-SGML document, an object repository than can be easily translated into SGML conforming to the Envision DTD. We also were able to extract objects from these documents, store them in a database, and establish links between them. However, writing this description was a demanding process. The problems that complicated this process are related with our implementation, and not with the DELTO language itself. The way in which component bindings are executed by our current DELTO system is not natural (see Appendix B). This complicates the task of specifying component bindings because the writer of a DELTO format description must consider the way in which bindings are executed to determine the correct way to write bindings. A person writing a format
CHAPTER 6. EXPERIMENTAL RESULTS

description also should know the mechanism that the DELTO Analyzer utilizes to recognize textual objects. Without this knowledge it is easy to write regular expressions that will make the system spend weeks trying to identify objects. For these reasons we include in Appendix B a series of suggestions that may help write correct format descriptions under the current implementation. We believe that these problems are not inherently related to DELTO, and that a better implementation could fix them.

Our current DELTO system is not very efficient, due to the utilization of the ODB system to manage the object repositories that it constructs. Some documents, especially large ones, may take more time to be processed than we would like. However, we did not set any specific goals concerning efficiency when we started this effort. We believe that the framework is the important concept, and we believe that we have developed an extremely powerful one. Efficiency can be improved once the correct framework is identified.

Retarget specifications have been very easy to write, regardless of the characteristics of the input documents. We believe Retarget to be an extremely powerful language, even though it is remarkably simple. A document can be translated into almost any imaginable format through a Retarget specification, given that the DELTO system is able to construct a good object repository for it.
Chapter 7
CONCLUSIONS AND FUTURE WORK

7.1 Conclusions

We have developed a comprehensive text translation and text extraction system. To make this system possible we designed two languages: DELTO and Retarget. With DELTO we can describe documents formats in terms of the objects that can occur within documents in the described formats. These descriptions are then converted into procedural specifications that are used to analyze a document and generate a database of objects that represents it. This database then is used as input to a translator that can generate a new document, equivalent to the input one, but in any format. The translation from the database of objects to the output document is specified through a Retarget specification.

We believe that our approach to translation is a breakthrough, when compared to the traditional approaches of sequentially mapping strings in the input to strings in the output. An object repository for a given document is not a flat, one-dimensional representation of the document, but one that can be accessed in many non-sequential ways. As a result of this, a complete restructuring of the document can be achieved. We were able to show this when we successfully processed pseudo-SGML full-text articles that had to be completely restructured to be translated to SGML conforming to the Envision DTD.

We developed a complete implementation for our framework. The implementation can be improved in many ways, but we still believe that it is a very good and robust one. It was not easy to implement the DELTO System. More than 50,000 lines of C code were written. We consider it a success to be able to generate procedural specifications from the mostly declarative format descriptions. It also has been a success creating a software component
CHAPTER 7. CONCLUSIONS AND FUTURE WORK

that is able to execute those very complex procedural specifications.

We have evaluated our system with documents from the Envision database and have succeeded in converting those documents to SGML conforming to our DTD, in automatically building a database of objects extracted from those documents, and in building links among those objects. Our approach to updating this database is flexible, given that all specifications about what to update and how to update it are given through DELTO format descriptions. More work is still needed with other types of documents from the Envision database to complete our evaluation process.

During our evaluation process we concluded that writing DELTO format descriptions for well structured formats proved to be an easy and natural process. In particular, structurally tagged documents can be processed in a very effective way by the DELTO System. Information can be easily extracted from those kinds of documents and stored in a permanent object database.

DELTO descriptions for less structured documents are not so easy to write under the current implementation. We still were able to write a good format description for the very ill-structured pseudo-SGML documents that we are processing, but writing that description was a more demanding task than we would like. The biggest problems are implementation dependent and are not related with the DELTO language. Some knowledge of the way in which DELTO recognizes objects based on a class definition is needed to write correct format descriptions. For this reason we have included in Appendix B of this thesis a description of concepts that should be known about writing format descriptions. A better implementation than the current one could isolate implementation dependent issues from the language and turn DELTO into a truly declarative, implementation independent, language. The weakest point about the implementation is related with component bindings. The way in which bindings are executed is not the most natural one as described in Appendix B.
CHAPTER 7. CONCLUSIONS AND FUTURE WORK

7.2 Recommendations for Future Work

Both languages, DELTO and Retarget, could be improved. Constructions that can provide more flexibility over component bindings could be incorporated into DELTO. In particular, statements for specifying conditional and iterative component bindings would significantly improve the text extraction capabilities of DELTO. The distinction between attributes and components may be artificial and could probably be removed to simplify the language.

Being an extremely simple language, it is surprising how powerful Retarget is. It allows generating of a document text in almost any format imaginable, given that a good object repository for the document is constructed by the DELTO Analyzer. Retarget conditional statements are very restrictive. They only allow comparison of a numeric component against an integer constant. Generalizing conditional statements to string components and allowing them to include logical operators would not hurt the simplicity of Retarget and would increase its expressive power.

A tool that could be constructed is an object repository browser. Without this tool it is not possible to examine the results of the analysis of a document before executing the DELTO translator. This problem is alleviated by the fact that it is always possible to examine the contents of the permanent object database to verify how it has been affected by a document processed by the DELTO System. It is also possible to examine the log file that it is generated containing information about the actions performed over the permanent object database.

Our current implementation is not very efficient. It may take more time than we would like to process some documents. This problem is due to the utilization of the ODB system to manage the object repository. It would be necessary to rewrite the object repository library so that it uses a more efficient representation to store information. The bottleneck is in the time that gdbm uses to store information, not in the retrieval, which is extremely fast. Our implementation is completely modular, which means that if the interface of the
CHAPTER 7. CONCLUSIONS AND FUTURE WORK

object repository library is not modified, then changing its storage schema would not affect at all the other components of the DELTO System.

7.3 Concluding Remarks

The DELTO system is still far from being a commercial product, but some work in improving its efficiency, in improving the languages in the ways described above, and especially in breaking the ties between the implementation and DELTO (mainly regarding the execution of bindings) could turn it into a very useful production system.
REFERENCES


REFERENCES


REFERENCES


Appendix A
MAN PAGES

In this Appendix we describe how to use the various software components that have been developed as part of this thesis. We describe the text interface to the ODB system in Section A.2, the text interface to the dictionary system in Section A.3, the DELTO compiler in Section A.4, the lexical analyzers in Section A.5, the object repository generator in Section A.6, the object repository analyzer in Section A.7, and the translator in Section A.8. In Section A.9 we describe an utility that invokes the entire document analysis and translation process. The user can invoke only this utility and forget about all the software components that participate in the process.

A.1 The DELTO Directory

Many of the software components that we developed look for certain files in the DELTO directory. This directory may be specified through the DELTO_DIRECTORY environment variable or through a command line argument. A command line definition always has precedence over the one specified by the environment variable.

A.2 ODB Interface

This is a very simple, text oriented interface that provides access to all the functions defined in the ODB library. To invoke this program simply type odms at the UNIX prompt. The program does not expect any command line arguments. The menu that is shown in Figure A.1 will appear. If menu option number 1 (Open a Database with Read-Write access) is selected, and provided that the user has read-write access to the specified database, the
APPENDIX A. MAN PAGES

Envision Object Database

------- ------- -------

No Database in use

Options:
1 - Open a Database (Read-Write)
2 - Open a Database (Read-Only)
3 - Create a Database
0 - Exit

Your Option:

Figure A.1: Starting menu of the ODB interface

The menu pictured in Figure A.2 appears. If the user has only read access, or if option 2 of the first menu is selected, then the second menu contains a reduced number of options, as shown in Figure A.3.

From the second menu, options between 11 and 14 provide class manipulation operations. Using these options, classes can be created, updated, and listed. Options between 21 and 29 focus on objects. They allow for the creation, update, deletion, and listing of objects. They also allow for objects to be indexed and retrieved on arbitrary strings. When an object is created through this interface an ID for the new object is suggested. The user is given the choice to accept this ID or to supply his own. For databases used together with the DELTO system it is always recommended to accept the suggested ID because it is guaranteed that this ID does not overlap with the IDs assigned by the DELTO system for objects that it creates.

Options between 30 and 35 of the second menu are associated with operations applied over the entire database. ODB objects of different classes are stored in the same file. This is completely transparent to the user of ODB. The user is only concerned with providing a valid object ID to the system who returns the associated object, regardless of where it is stored. Only one file may be used to store all the objects of a database. However, the file
Envision Object Database

In use Read-Write Database: envision
New Objects will be Stored in Object File # 1
New Indexes will be Stored in Index File # 1

Options:

11 - Add Class
12 - Get Class Info
13 - Update Class
14 - List Classes
21 - Add Object
22 - Get Object Info
23 - Update Object
24 - Delete Object
25 - List Objects
26 - Index Object
27 - Index Class of Objects
28 - Unindex Object
29 - Get Object Info from Index String
30 - Change New Objects File Number
31 - Create a New Object File
32 - Change New Indexes File Number
33 - Create a New Index File
35 - Reorganize Database
50 - Closed Currently Opened Database
0 - Exit

Your Option:

Figure A.2: ODB menu for accessing read-write databases
APPENDIX A. MAN PAGES

Envision Object Database

In use Read-Only Database: envision

Options:
  12 - Get Class Info
  14 - List Classes
  22 - Get Object Info
  25 - List Objects
  29 - Get Object Info from Index String
  50 - Closed Currently Opened Database
  0 - Exit

Your Option:

Figure A.3: ODB menu for accessing read-only databases

where objects are stored may get too large which may cause, for example, that it does not
fit in a file system. That is why the ODB system allows the user to partition a database
into different files. This partition is completely transparent to the user. The database is
still an atomic unit, even though it may be composed of several object storage files. Object
databases are always created with only one object storage file (whose name ends with the
extension .objs.1). At any point the user may create a new object storage file, by selecting
option 31 (Create a New Object File) of the menu. Even after a second object storage file is
created, new objects will continue to be stored in the original file, unless option 30 (Change
New Objects File Number) is selected and another object file is chosen as the one where
new objects are to be stored. Note that at the top of the read-write menu (shown in Figure
A.2) there is an indication that informs on which file are new objects going to be stored.

Indexes are stored in the same way as objects are, in the files whose names end with
.index. followed by a number. Again, all indexes are stored in index file number 1 until a
new index files is created and selected as the recipient of new index entries. This can be
accomplished through options 32 and 33 of the menu.
APPENDIX A. MAN PAGES

Option 35 (Reorganize database) physically eliminates previously deleted objects (objects are not physically eliminated from a database when they are deleted) and unused storage. For object databases whose objects are frequently updated, reorganizing the database may save a considerable amount of disk space.

Every time the user is prompted for information, except in the menus, hitting only the return key returns the program to the previous menu.

A.3 Dictionary Interface

This is also a text interface that provides access to all the functions defined in the dictionary library that we developed. It is not as simple to use as the ODB interface, especially with respect to adding new entries to a dictionary. We believe that the dictionary library functions that we developed are very powerful, but this interface could be improved significantly.

To invoke the dictionary interface type mdict at the UNIX prompt. No command line arguments are expected. The menu of Figure A.4 appears. If option number 1 (Open a Dictionary) is selected, and the user has read-write access to the specified dictionary, then the menu shown in Figure A.5 is displayed. If the user has read-only access to the specified dictionary, then the system displays a reduced menu, as shown in Figure A.6.

Option 3 of the read-write menu (Add Entry) must be selected to add an entry to the opened dictionary. An entry can be added for a single word or for a sequence of words. An entry must include a category for the word or sequence of words, and it may contain one or more attributes that additionally describe the word or sequence of words. More than one entry may be entered in a dictionary for the same word or sequence of words.

Dictionary entries are stored in a LISP-like format. The greatest weakness of this interface is that it requires the user to provide entries to be added to the dictionary in this format. After selecting option 3 of the menu, the system prompts the user for a word. To add an entry for a word the user must type that word in response to the system prompt.
APPENDIX A. MAN PAGES

DELTOS Dictionary Maintenance Program

---

No Dictionary in use

Options:

1 - Open a Dictionary
2 - Create and Open a Dictionary
0 - Exit

Your Option:

Figure A.4: First menu displayed by the dictionary interface

To add an entry for a sequence of words the user must type the first word of the sequence. The system prompts then for the definition of that word. The definition must be entered in LISP-like format. The first character of a definition is always a left parenthesis. If the entry being added is for a sequence of words, then all the words except for the first one must be typed after the opening parenthesis, each one separated by a blank. Following this, another opening parenthesis must be included. Note that if the entry is for a single word the two opening parenthesis must be together. For example, if we want to add an entry for "Virginia" we must enter (up to the point that has been explained):

Word: Virginia
Definition: ((

If we want to add an entry for "Virginia Polytechnic Institute" we must enter:

Word: Virginia
Definition: (Polytechnic Institute

Note that the dictionary interface is case insensitive, which means that entering "Virginia" is the same as entering "virginia". Following the second left parenthesis, the category of the entry must be included. For example, we may want to categorize "Virginia" as a given.name and "Virginia Polytechnic Institute" as a university.name. We
DELTO Dictionary Maintenance Program

----- ----------- ----------- -----

In use Read Write Dictionary delto

Options:

3 - Add Entry
4 - Fetch Entry as Stored
5 - Look Up Single Word
6 - Look Up Word Sequence
7 - Add Complete Dictionary Data File
8 - Delete Dictionary Entry
9 - Add Word Substitute Entry
10 - Look Up Word Substitute
11 - List All Substitutions
12 - Add Complete Substitutions Data File
13 - Delete Word Substitute Entry
14 - Process Word Suggestion File
15 - Process Substitution Suggestion File
19 - Reorganize Dictionary
20 - Close Currently Opened Dictionary
0 - Exit

Your Option:

Figure A.5: Dictionary interface menu for read-write dictionaries
APPENDIX A. MAN PAGES

DELTO Dictionary Maintenance Program

-----  ---------  --------  ---------

In Use Read Only Dictionary delto

Options:

  4 - Fetch Entry as Stored
  5 - Look Up Single Word
  6 - Look Up Word Sequence
  10 - Look Up Word Substitute
  11 - List All Substitutions
  20 - Close Currently Opened Dictionary
  0 - Exit

Your Option:

Figure A.6: Dictionary interface menu for read-only dictionaries

may then include any number of attribute-value pairs, each of which must be enclosed within parentheses. The attribute names must start with a letter and may continue with letters, digits or underscores (even though the system does not check this). Attribute values may be non-negative integers, strings enclosed within double quotes, or identifiers following the same rule as the one for attribute names. For example, we may want to include an attribute for "Virginia" that specifies that it is a female given name. Following the attribute pairs, if any, the two previously opened parentheses must be closed, which completes the definition. Even though the system does not check for the correctness of the attribute names or values, it does check for the syntactic correctness of the overall definition and does not allow an invalid definition to be entered into a dictionary. The two examples that we have described in this paragraph can be visualized in Figure A.7.

Options 4 through 6 retrieve entries stored in the opened dictionary. Option 4 (Fetch Entry as Stored) does not perform any analysis of the entries that it finds for a given word. It just displays every entry that it encounters in its internal format. Options 5 and 6, for retrieving words and sequences of words respectively, perform a little analysis of the
APPENDIX A. MAN PAGES

Word: Virginia
Definition: ((given_name (gender f)))

Word: Virginia
Definition: (Polytechnic Institute (university_name))

Figure A.7: Entries added into a dictionary

retrieved entries and produce an easier to read output.

Option 7 (Add Complete Dictionary Data File) is designed to enter into a dictionary all the entries stored in a text file. The names of these files must end with .ddt. Each line in such files is considered an entry. The first element of a line must be a word. Following the word, a definition for it conforming to the syntax described above must be included. In Figure A.8 we present a short dictionary data file. This feature is useful for loading into a dictionary large lists of words of the same category. For example, we may want to load any of the large lists of person names that are available on the Internet. For this purpose we developed a short lex program that reads in a list of words and generates a dictionary data file containing entries for each of the words in the original file. This program can be invoked in the following way:

\texttt{wl2dic input\_file output\_file [lexical\_definition]}

where the command line argument enclosed within square brackets is optional. If a lexical definition is given as a command line argument, it is used for every word in the input file.

Every time an entry is added interactively, through option 3 of the menu (Add Entry), a line is added to a dictionary data file associated with the opened dictionary. In this way, all the entries added interactively to a dictionary can be automatically added to another dictionary, or to the same dictionary if for any reason information contained in it is lost.

Option 8 (Delete Dictionary Entry) allows to selectively delete individual entries associated with a given word.

Options 9 through 13 allow the user to work with word substitutions. Word substitu-
Journal ((journal_indicator))
ACM ((organization))
Inc ((company_indicator))
UCLA ((university_name))
University ((university_indicator))
Frand ((last_name))
SCHOOL ((university_indicator))
IBM ((company_name))
XTs ((computer_type))
ATs ((computer_type))
PS ((computer_type))
Zenith ((company_name))
Apple ((company_name))

Figure A.8: A dictionary data file

tions should be specified for abbreviations. For example, we could define "Virginia" as a substitution for "VA". In this way, we do not have to define "VA Polytechnic Institute" as a university name. The dictionary system is able to recognize it as such because it can replace "VA" with "Virginia". The drawback of this approach is that "VA" can now be also interpreted as a given name.

In the same way as with entries for words or sequences of words, substitutions can be entered interactively (through option 9) or automatically from the information contained in a text file (through option 12). The names of these files must end with the extension .sbst. Each line must contain a single substitution entry. The abbreviation or substitute must be followed by the unabbreviated word, as shown in Figure A.9. Again, each time a substitution entry is added interactively, a line is added to a substitutions data file associated with the opened dictionary. This file then can be used to automatically add those substitutions to another dictionary or to the same one, if needed. Only one substitution entry is allowed for each substitute or abbreviation. This limitation is imposed by the dictionary library itself. It would be useful to remove this constraint and allow an abbreviation to be the substitute of more than one word.
APPENDIX A. MAN PAGES

Assn. association
Symp. symposium
Jan january
Co Company
AcM Association for Computing Machinery
Fyi For your information

Figure A.9: A substitutions data file

Options 14 and 15 of the menu allow the user to process suggestion files generated by any of the DELTO lexical analyzers, as described in Section 3.4.1. Option 14 is for processing word suggestions, while option 15 is for processing substitution suggestions. For word suggestions the user is prompted with a suggested word. The user may then enter a definition for the word following the syntax described above for word definitions, in which case the entry is added to the dictionary. The user may, alternatively, skip the word. The next suggested word is then presented to the user. This process continues until the suggestion file is exhausted or until the user quits. Suggestion files are updated so that the next time that they are processed, words that have already been presented to the user are not suggested again. Substitutions suggestions work in the same way, with the difference that instead of a definition, the user may enter an unabbreviated word for the suggested substitution.

Option 19 (Reorganize Dictionary) may significantly reduce the size of the dictionary if entries are frequently deleted, or if entries for words that already exist in the dictionary are frequently added.

A.4 DELTO Compiler

The DELTO compiler must be invoked in the following way:

```
dlc format_description[.dlt] [-dd DELTO_directory] [-s]
```
APPENDIX A. MAN PAGES

Format description file names usually end with the extension .dlt. This extension needs not be specified to the compiler if the name of the description to be compiled does not include any other dot, except for the one of the mentioned extension. The DELTO directory must be specified, either through the environment variable or through the -dd command line argument. The compiler needs access to certain files that are located in the DELTO directory (the .dlc files).

For a given format description, the compiler generates an ODB database that mirrors the data structures that it generates as a result of parsing and analyzing the format description. This database, named in the same way as the format description, can be viewed through the ODB interface. However, it should not be modified. Unpredictable results may occur when executing the DELTO analyzer over a compiled format description that has been manually updated. This ODB database is not generated if the -s argument is included in the command line. In this case the compiler just checks the syntactic and semantic correctness of the given format description.

The compiler also generates, even when the -s option is active, three other files: a .lst file that contains a human readable representation of the data structures that it generated; a .ci file that contains a list of the classes whose objects must be indexed within the object repositories of documents processed with this format description; and a .wi file that contains the words that must be indexed within the object repositories of documents processed with this format description. These last two files are needed by the object repository generator as described in Section A.6.

A.5 Lexical Analyzers

We developed three lexical analyzers, one for processing general SGML documents, one for processing SGML bibliographic documents conforming to the Envision DTD, and a default lexical analyzer for processing any other ASCII documents. Their names are lex.sgm, lex.bibs, and lex.default, respectively. The three lexical analyzers are based on
APPENDIX A. MAN PAGES

the skeleton defined by the lxl.all.h file. Creating a new lexical analyzer is a simple task: the lxl.all.h must be included and the lex rules for the tokens that want to be identified must be specified. The lxl.all.h handles the interfacing with the dictionary system and with the lexical functions library, and also the command line processing. For this reason, the usage of the three lexical analyzers that we developed is identical. We use lxl.sgml in our examples, but the other two lexical analyzers can be used interchangeably.

A lexical analyzer should be invoked in the following way:

    lxl.sgml [-i input_file] [-o output_file] [-d dictionary] [-help]

If the input file is not specified, then standard input is used. If both the input file and the output file are left unspecified, then both standard input and standard output are used. If only the input file is specified, then a file named in the same way as the input with the extension .lx is used as output. If the dictionary is not specified, then the default dictionary is used. The default dictionary is named delto and must be located in the DELTO directory. There is no command line argument to specify the DELTO directory. It must be specified through the environment variable (a command line argument for this purpose should be added to the lexical analyzers). If a lexical analyzer is invoked with the -help command line argument, then a message explaining its usage is displayed.

The names of all the lexical tokens that can be generated by a lexical analyzer cannot be known because this depends on the dictionary that it is used. Adding new categories to a dictionary results in new types of tokens that can be generated by a lexical analyzer using that dictionary. In addition, the SGML lexical analyzer not only generates a sgml.start.tag token for every SGML start tag that it encounters, but it also generates a token whose category depends on the name of the tag. For example, for a tag such as <document> the lexical analyzer generates both a token of category sgml.start.tag and a token of category sgml.document.tag. If a dot occurs within the name of an encountered SGML tag, like in <b.document>, it is replaced by an underscore within the category of the generated token (sgml.b.document.tag). The SGML lexical analyzer has no way to know
APPENDIX A. MAN PAGES

the category of all the tokens of that kind that it may generate.

Some possible categories of tokens that a lexical analyzer may generate are known in advance. These are listed in .toks files associated with each lexical analyzer that must be located in the DELTO directory. A format description that uses a particular lexical analyzer does not need to define as classes the categories listed in the .toks file of the lexical analyzer that it uses. They are automatically defined and instances of them can be referenced freely throughout the entire format description. For example, objects of class word can be referenced within format descriptions that use either the lx.default or the lx.sgml lexical analyzers, without having to declare word as a class within the format description.

However, if any of the attributes generated by the lexical analyzer are referenced, then these must be defined as attributes of the class within the format description. If for example, a regular expression unit such as <<word>> is included within a format description, then there is no need to define the word class. If a unit such as <<word,case=1>> is included, where case is one of the attributes that the lexical analyzers generate for word tokens, then the following class definition must be added to the format description:

```plaintext
CLASS DEFINITION: Word
    ATTRIBUTES
    {
        Case : NUMERIC;
    }
ENDDEF;
```

If a reference to an object of a class generated by the lexical analyzer is made within a format description, and that class is not listed in the .toks file of the lexical analyzer, then the class must be defined within the format description. For example, a format description referencing the class sgml.document.tag generated by the SGML lexical analyzer, must include the following class definition:

```plaintext
CLASS DEFINITION: Sgml_document_tag
ENDDEF;
```
APPENDIX A. MAN PAGES

A.6 Object Repository Generator

The object repository generator must be invoked in the following way:

```bash
dlctcrt lx_file [-dd delto_directory] [-o repository_name]
            [-ci classes_to_index_file_name]
            [-wi words_to_index_file_name]
```

The name of a file generated by a lexical analyzer must be specified. The path to the
DELTO directory must be available to the object repository generator, either through the
environment variable or through the command line argument. If the name of the object
repository to be generated is omitted, then an object repository with the same name as the
one of the input file is created. The -ci command line argument specifies a file that contains
the names of all the classes whose objects must be indexed. The file that should be given as
this argument is the .ci file generated by the DELTO compiler for the format description
being used. The -wi argument indicates a file that contains all the words that must be
indexed. In this case, the .wi file generated by the compiler for the format description
being used is the one that should be given to this argument.

A.7 Object Repository Analyzer

The object repository analyzer must be invoked in the following way:

```bash
dla repository_name -d description_name [-o object_database_name]
            [-dd delto_directory] [-l object_database_log_file_name]
            [-dc dictionary] [-of object_file_name]
```

The name of an object repository and of a DELTO format description are required. A
path to the DELTO directory must be available, either through the environment variable
or through the command line argument. If the format description cannot be located in the
current directory, then the DELTO directory is searched for it (as long as a path has not
been included before the name of the description). If an object database name is given by
means of the -o directive, then all the specifications contained in ODB Interface declarations
APPENDIX A. MAN PAGES

within the used format description are executed and the object database is updated with the information extracted from the processed object repository. Every action applied to the object database is recorded in the log file specified with the -l directive. A program for undoing those actions, given the log file, could be easily written. The name that the final document will have, after the DELTO translator is executed, may be specified through the -of directive. This is for the sole purpose of recording that file name in the object associated with the entire document. A dictionary may be specified with the -dc directive. If it is not specified, then the default dictionary located in the DELTO directory is used. The dictionary is used only for executing the subst function that is described in Section 4.4.6.

During its execution the object repository analyzer sends to the standard output information about the class of objects that it is working on. For each class, it first outputs a message saying that it is trying to recognize instances of that class and then, if any such instances have been recognized, it writes a message saying that it is verifying each of those instances. It also outputs a number which represents the number of processes that it will fork to verify those instances. The number of forked processes is in direct relation with the number of potential objects that have been recognized.

The resulting object repository may be browsed with the use of the ODB interface, though this may be of little usefulness without the knowledge of the internal structure of the repository. A very useful tool that could be constructed is an object repository browser. Until such a tool is constructed, it will be complicated to verify the results produced by the analysis of a document before a translation is generated.

A.8 DELTO Translator

The DELTO translator must be invoked in the following way:

```
rtg trans_specification -r rep_name [-o output_file_name]
[-l log_file_name] [-rp replacement_file_name]
[-dd delto_directory]
```
APPENDIX A. MAN PAGES

A Retarget translation specification and an object repository must be provided to the translator. In addition there must be a path to the DELTO directory available, either through the environment variable or through the -dd command line directive. If the translation specification is not found in the current directory, then the DELTO directory is searched. An output file may be provided through the -o directive. If it is not provided, then the output is routed to the standard output. A log file where error messages may be stored can be specified through the -l directive. If no log file is given, then error messages are sent to the standard error.

In Section 3.5 we described the replacement file that can be used by the DELTO translator to isolate the Retarget specification and the translation process from certain complexities of the target format. The replacement table maps strings that may be generated by the translation process and that cannot be copied literally to the output, into proper strings that can be safely written to the output. Characters that need to be escaped in the target format are an example of the utility that this table may have. The name of such a table may be given to the translator through the -rp command line directive. A string that cannot be copied to the output must be included at the beginning of a line within this table. The string located at the beginning of the following line is the one that must be used to replace it. An empty line must be left before another two lines, like the ones described above, can be included. For example, in Figure A.10 we present a replacement table that maps some characters into their escaped equivalents.

A.9 DELTO Execution Utility

The DELTO execution utility invokes all the software components described in this Appendix that are needed for a particular analysis and translation task. This utility must be invoked in the way described in Figure A.11.

The name of the input document must be provided followed by the name of the DELTO format description to be used. The system looks for the format description in the current
APPENDIX A. MAN PAGES

\[
\]
\[
\}
\[
\}
\]

Figure A.10: A replacement table for escaping some characters

dlx input_file delto_description number_output_files
 {output_file translation_description
  [-rp replacement_file] [-log translation_log_file]
} /*repeated number_output_files times */
[-dc dictionary] [-cs class_stop_file] [-vd work_directory]
[-odb object_database] [-dlog dla_log_file] [-noc]

Figure A.11: Usage of the DELTO execution utility
directory first and, if it cannot find it in that directory, in the DELTO directory. The
DELTO directory is only specifiable through the \texttt{DELTO\_DIRECTORY} environment variable.

Following the name of the input file and of the format description, the user must specify
the number of output documents that must be generated for the input document. This could
be any non-negative number, including 0 if no output documents are needed. Following this
number, the user must specify information about how to construct each of the requested
output documents. This information consists of the name of the output file, the name
of the Retarget specification to be used in the translation and, optionally, the name of a
replacement file to be used for the translation, and the name of the log file that will contain
the translation process message errors, if there is any. The Retarget specification is searched
in the current directory and, if it is not found there, in the DELTO directory.

The user may specify, following the information concerning the output documents, a diction-
ary name (the default one is used if left unspecified), a working directory (the current
directory is used if left unspecified), an object database that is used to store the persis-
tent objects extracted from the processed document, the name of a log file for recording
operations applied over the object database (standard error is used if left unspecified), and
the \texttt{-noc} directive. If the \texttt{-noc} directive is specified, then the object repository generated
for the input document is not deleted from the working directory. By default, the object
repository is deleted once the process is complete. For example, we may invoke the DELTO
execution utility in the following way to process a document and generate two translations
of it in two different formats:

\begin{verbatim}
dlx doc.x desc.dlt 2 doc.sgml x.2sgml doc.tex x.2tex -odb database
\end{verbatim}

The above invocation processes document \texttt{doc.x} with the DELTO format description
\texttt{desc.dlt}. It generates two output documents. The first one named \texttt{doc.sgml} which is
generated using the \texttt{x.2sgml} Retarget specification. The second output document is named
\texttt{doc.tex} and is generated using the \texttt{x.2tex} Retarget specifications. The information about
persistent objects extracted from the input document is stored into the permanent object
APPENDIX A. MAN PAGES

database named database. The object repository for the input document is created in the current directory and is deleted once the process is completed.
Appendix B

HINTS FOR WRITING DELTO FORMAT DESCRIPTIONS

In this appendix, we provide a few suggestions that may make the process of writing a DELTO format description easier. Most of the difficulties that may appear when writing a DELTO description are related to the current implementation and are not inherent to DELTO. In particular, some knowledge of the way the DELTO System works may be useful when writing format descriptions.

B.1 Recognizing Objects

For each class that is defined in a format description, the DELTO System executes a recognition process that identifies all the potential objects of the class within a document and a verification process that filters those potential objects that are not valid ones. This verification process is also responsible for assigning a weight to each valid object.

The recognition process is crucial. Whatever is not identified by this process is not identified at all by the system. For the recognition process, the DELTO Compiler selects the best predicates within a class definition that can be used to recognize objects. One or more predicates are selected by the compiler. This selection is related to the kinds of predicates, the frequency indicators associated with them, and the way predicates are connected to each other by means of logical operators (OR or AND).
APPENDIX B. HINTS FOR WRITING DELTO FORMAT DESCRIPTIONS

B.1.1 Recognizers

There are only three kinds of predicates that can be used as recognizers for a class: is.a predicates, pattern predicates, and begins predicates combined with ends predicates. The best recognizer is the is.a predicate. If an is.a predicate associated with an ALWAYS frequency indicator is defined for a class, then this predicate is the only recognizer that is selected by the DELTO Compiler. If an is.a predicate and a pattern predicate are defined, and they are not associated with the ALWAYS frequency indicator, then they are both selected as recognizers. The process of selecting recognizers is complex, and it need not be fully explained here. The important thing to remember is that for every class of textual objects there must be, at least, one (or two, in the case of a begins-ends combination) predicate that can be used as a recognizer. Otherwise, the compiler signals an error. Even for the document class the current implementation of the compiler requires a recognizer (though it should not, because it is never used by the DELTO Analyzer). The simplest recognizer that may be given for the document class is the following begins-ends combination.

\begin{verbatim}
  begins( DOC_BEG BOUNDARY );
  ends( BOUNDARY DOC_ENDS );
\end{verbatim}

The is.a recognizer

The is.a predicate is the best recognizer because it is the one that can recognize potential objects fastest. A few precautions should be taken when using it. If, for example, a predicate such as is.a("<<A>>") is specified within a class definition, then all the objects of class A are going to be indexed on their class within the object repository, as described in Section 3.4.2. This means that if A is a class of very frequent objects, then the indexing entry for class A is going to be large and, most important, the time required to index every object of class A will be large. For this reason, it is not convenient to define such predicates. For example, a predicate such as is.a("<<word>>") triggers an extremely long process (during the object repository generation) that indexes every single word occurring in the input document. The argument to an is.a predicate should always be a class whose instances do
APPENDIX B. HINTS FOR WRITING DELTO FORMAT DESCRIPTIONS

not occur with high frequency.

Note also that expressions included within the regular expression argument given to an is_a predicate are ignored by the current implementation. For example, the following two predicates are considered equivalent.

\[
is_a( \langle\langle\text{paragraph, \#words > 30}\rangle\rangle);
\]
\[
is_a( \langle\langle\text{paragraph}\rangle\rangle);
\]

Other recognizers

Similar considerations to those related to the is_a predicates must be taken for pattern and begins-end predicates. For a regular expression given to a pattern, begins, or ends predicate, one or more classes or strings are selected by the compiler as the triggers of the predicate (when used as a recognizer). For example, given the following expression:

\[
\text{pattern}( \langle\langle\text{A}\rangle\rangle* \langle\langle\text{B}\rangle\rangle* \langle\langle\text{C}\rangle\rangle \langle\langle\text{D}\rangle\rangle);
\]

objects of classes A, B or C can be triggers of the recognizer, because an instance of any of those objects can start a piece of text that satisfies the expression. Note that objects of class D cannot start a piece of text satisfying the expression. For that reason, objects of class D are not triggers of this recognizer. The objects that can act as triggers of recognizers are also indexed on their class. For the previous example, all objects of classes A, B, and C are indexed within the object repository. Again, it is not convenient to use objects of very frequent classes as triggers to pattern, begins, or ends predicates. For example, it would not be wise to define the following expression:

\[
\text{begins}( \langle\langle\text{word}\rangle\rangle \langle\langle\text{given_name}\rangle\rangle \langle\langle\text{last_name}\rangle\rangle);
\]

because it would imply that all words in the analyzed documents must be indexed.

The same considerations apply for ends predicates but the triggers of these predicates are the objects that may end a piece of text that satisfies the predicate. For example, for the predicate
APPENDIX B. HINTS FOR WRITING DELTO FORMAT DESCRIPTIONS

ends( <<C>> <<B>> <<A>>? );

the triggers are the objects of classes A and B (this is because ends predicates are matched from right to left).

The same considerations must be taken when using strings as regular expression operands. A string can also be a trigger, as for example in the following expression:

    pattern( "Introduction" <<word>>+ );

In this case the string "Introduction" is a trigger of the expression, which means that all occurrences of that string are going to be indexed in the object repository. Again, care should be taken to avoid using very frequent strings as triggers. For example, it is not recommended to specify an expression such as:

    pattern( "the introduction" <<word>>+ );

Given the above expression, all the occurrences of the word "the" (the first substring of the entire string) are indexed within the object repository.

The wildcard symbol, ANYTHING cannot be used as a trigger. The DELTO Analyzer signals an error during execution in this case (invalid regular expression entry point), and the expression is ignored. Note that this is not detected by the compiler, though it probably should be. For example, an expression such as the following is not valid.

    pattern( ANYTHING <<given_name>> );

A more sophisticated implementation of DELTO could allow expressions such as the one shown above.

Embedded expressions

Expressions included within regular expression units are ignored for the purpose of recognizing potential objects with pattern, begins, or ends predicates. For example, given the following expression:
APPENDIX B. HINTS FOR WRITING DELTO FORMAT DESCRIPTIONS

    pattern( <<paragraph, #word >20>> <<title>> );

all paragraph objects are used as triggers for the expression, not just those that satisfy the embedded expression. During the verification process the ones that do not satisfy the embedded expression are eliminated as potential candidates. For this reason, the second of the following two predicates is always preferred when working with SGML input.

    pattern( <<sgml_start_tag, tag = "title">> <<person_name>> );
    pattern( <<sgml_tag_title>> <<person_name>> );

Note that the first expression triggers a process that recognizes as potential objects every SGML tag (of any kind) followed by a person name, leaving up to the verification process the task of eliminating those potential objects that do not begin with a title tag. On the other hand, the second expression recognizes exactly those objects that are needed, speeding up the process significantly.

Begins-ends combination

Sometimes it may be better to specify a begins-ends combination than a pattern predicate. For example, to define objects that may represent very large pieces of text, the following expression may result in very inefficient recognition and verification time.

    pattern( <<nl>> <<nl>> BOUNDARY ANYTHING* BOUNDARY <<nl>> <<nl>> );

On the other hand, the following two expressions can recognize the same objects in much less time, given the fact that they do not have to scan the entire recognized objects, but only their boundaries.

    begins( <<nl>> <<nl>> BOUNDARY );
    ends( BOUNDARY <<nl>> <<nl>> );

B.2 Bindings

Bindings are not executed in the most natural way by the DELTO System. For example, the following two bindings are not executed in the way which we may consider most natural.
APPENDIX B. HINTS FOR WRITING DELTO FORMAT DESCRIPTIONS

pattern( <<A>><x[1] := y[1],
         x[2] := x[1]] );

We may think that the value that is bound to x[2] is the one that is bound to x[1] as a result of the first binding in the sequence, but this is not true. The reason is that the DELTO Analyzer first evaluates the left-hand side of a binding, then the right-hand side of it, and then repeats this process for all the other bindings that must be applied to an object, before actually executing any of the bindings. Only after all the left-hand sides and right-hand sides of all bindings associated with an object are evaluated, are the bindings executed. In our previous example, the value that is bound to x[2] is the value that x[1] had before it is bound to y[1].

However, if a binding left-hand side creates a new object, then this new object can be referenced in successive bindings because the object is created at the time of the left-hand side evaluation, and before the binding is executed. The following expression is an example of this characteristic.

pattern( <<A>><!.
         !.VV[NEW].A[NEW] := ØO,
         !.VV[LAST].S[NEW] := "A string"] );

In this case, an object of class VV is created by the first binding, and a value is bound to component A of that new object. The second binding creates a new component for the last referenced object of class VV (in this case, the newly created one), and binds a value to it.

Note, however, that the third binding in the following sequence does not result in what we may expect.

pattern( <<A>><!.
         !.VV[NEW].A[NEW] := ØO,
         !.VV[LAST].S[NEW] := "A string",
         !.XX[NEW].S[NEW] := !.VV[LAST].S[LAST] );

The last binding of the previous sequence creates an object of class XX. This object is created when evaluating the left-hand side of the binding. The right-hand side of the
APPENDIX B. HINTS FOR WRITING DELTO FORMAT DESCRIPTIONS

binding references the last value bound to component S of the last referenced object of class W. Note that the last referenced object of class W is the one that is created by the first binding in the sequence, but because both sides of all the bindings for an object are evaluated before any of the bindings are executed, there is no value bound to component S of the newly created W object at the time when the right-hand side of the last binding in the sequence is evaluated. Therefore, the right-hand side of the last binding in the sequence evaluates to a null value and the binding is not executed.

Again, objects created by bindings can be referenced again by successive bindings, but not values bound to new or existent objects. These values can only be referenced by other class definitions, which is the same as saying that the flow of component and attribute values is strictly bottom-up.

A future implementation could execute bindings in a more natural way. The way in which the current implementation executes bindings is a result of the need to delay the execution of bindings (especially those affecting objects other than the one being recognized) until the system knows for sure that the object being recognized is a valid object. A different approach would be to execute all bindings immediately and undo them if the system discovers that the object being recognized is not a valid one.
Appendix C

DE尔TO FORMAT DESCRIPTIONS USED FOR ENVISION

In this Appendix we present the three DELTO format descriptions that we developed for our evaluation process, and that are actually used to process Envision documents. In Section C.1 we present the general format description that we developed. This description is used by the other two format descriptions and should be used by any future format descriptions to be developed for processing Envision data. In Section C.2 we show the format description that we use to process SGML bibliographic documents, and in Section C.3 we introduce the format description that we are using to process pseudo-SGML full-text articles.

C.1 General Format Description

 Ancient comments:

/* File: general.dlt */
/* Descr.: General DELTO description, to be used for all specific */
/* format descriptions needed by Envision. Contains */
/* definitions for all persistent classes and for */
/* all the classes that may be needed by most specific format */
/* descriptions. */
/* Author: Guillermo Andres Averboch */
/* Dates: 05/16/95 (v 1.0) */
/* Compile: dlc general */
/*-----------------------------------------------*/
LEX: "lx.default";

CLASS DEFINITION: Address
COMPONENTS
{
    street  : String 0..1;
    city    : String 0..1;
}
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

```
sbd : String 0..1;
pc  : String 0..1;
country : String 0..1;
}
ENDDEF;

CLASS DEFINITION: Person_Name
COMPONENTS
{
  fname : String 0..1;
mname : String 0..N;
sname : String 0..1;
person : Person 0..1;
}
ENDDEF;

CLASS DEFINITION: Corporate_Name
COMPONENTS
{
  cname : String 0..1;
  inst_object : Institution 0..1;
}
ENDDEF;

CLASS DEFINITION: Publisher_Name
COMPONENTS
{
  pname : String 0..1;
  inst_object : Institution 0..1;
}
ENDDEF;

CLASS DEFINITION: Publication_Name
COMPONENTS
{
  pname : String 0..1;
publ_object : Publication 0..1;
}
ENDDEF;

CLASS DEFINITION: Date
COMPONENTS
{
  year : String 1..1;
  month : String 0..1;
  day : String 0..1;
}
```
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

ENDDEF;

CLASS DEFINITION: Document
PERSISTENT INSTANCES
COMPONENTS
{
  Title : String 1..1;
  Item_type : String 1..1;
  Author : Person 0..N;
  Institution : Institution 0..N;
  Editor : Person 0..N;
  Published_in : Publication 0..1;
  Publisher : String 0..1;
  Edition : String 0..1;
  Series : String 0..1;
  Date : Date 0..1;
  Volume : String 0..1;
  Number : String 0..1;
  Chapter : String 0..1;
  PP : String 0..1;
  Abstract : String 0..1;
  Issn : String 0..1;
  Isbn : String 0..1;
  Coden : String 0..1;
  Libcon : String 0..1;
  Crt : String 0..1;
  Crdat : Indexing_Term 0..N;
  Term : Indexing_Term 0..N;
  Keyword : Indexing_Term 0..N;
  Bib_Document : Document 0..N;
  Bib_Entry : Document 0..N;
  Cites : Document 0..N;
  Cited_in : Document 0..N;
}
ODB-INTERFACE
{
  ACCESS-KEYS
  {
    Clear-String( Title ) very-strong;
    Concat( Author[EACH].Name.Sname ) moderate;
  }
  MATCH-DESC
  {
    ALWAYS
    {
      Similar-to( Title, ODB.Title, 75 );
      #Author = ODB.#Author;
    }
  }
}
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

} } 

ACTIONS
{
  if( not Found() )
  {
    odb_add_value( ODB.Title, Title );
    odb_add_link( ODB.Author, Author[EACH].ODB.Author_of );
    odb_index_on( Clear-String(Title) );
    odb_index_on( Concat(Author[EACH].Name.Sname) );
  }

  if( ODB.#Item_type = 0 )
  {
    odb_add_value( ODB.Item_type, Item_type );
  }

  if( ODB.#Institution = 0 )
  {
    odb_add_link( ODB.Institution, 
                  Institution[EACH].ODB.Document );
  }

  if( ODB.#Published_in = 0 )
  {
    odb_add_link( ODB.Published_in, 
                  Published_in[EACH].ODB.Document );
  }

  if( ODB.#Editor = 0 )
  {
    odb_add_link( ODB.Editor, Editor[EACH].ODB.Editor_of );
  }

  if( ODB.#Publisher = 0 )
  {
    odb_add_value( ODB.Publisher, Publisher );
  }

  if( ODB.#Edition = 0 )
  {
    odb_add_value( ODB.Edition, Edition );
  }

  if( ODB.#Series = 0 )
  {
    odb_add_value( ODB.Series, Series );
  }

  if( ODB.#Date = 0 and #Date > 0 )
  {
    if( Date.#day > 0 and Date.#month > 0 and Date.#year > 0 )
    {
      ...
    }
  }
}
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

odb_add_value( ODB.Date, Date.month + " "+
        Date.day + "," +
        Date.year );

} else if( Date.#day == 0 and Date.#month > 0 and Date.#year > 0 )
{
    odb_add_value( ODB.Date, Date.month + "," +
        Date.year );
}

if( Date.#day == 0 and Date.#month == 0 and Date.#year > 0 )
{
    odb_add_value( ODB.Date, Date.year );
}

if( ODB.#Volume == 0 )
{
    odb_add_value( ODB.Volume, Volume );
}

if( ODB.#Number == 0 )
{
    odb_add_value( ODB.Number, Number );
}

if( ODB.#Chapter == 0 )
{
    odb_add_value( ODB.Chapter, Chapter );
}

if( ODB.#PP == 0 )
{
    odb_add_value( ODB.PP, Clear-String(P) );
}

if( ODB.#Abstract == 0 )
{
    odb_add_value( ODB.Abstract, Clear-String(Abstract) );
}

if( ODB.#Issn == 0 )
{
    odb_add_value( ODB.Issn, Issn );
}

if( ODB.#Isbn == 0 )
{
    odb_add_value( ODB.Isbn, Isbn );
}

if( ODB.#Coden == 0 )
{
    odb_add_value( ODB.Coden, Coden );
}

if( ODB.#Libcon == 0 )
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

```c
{  
    odb_add_value( ODB.Libcon, Libcon );
}
if( ODB.#Crt = 0 )
{
    odb_add_value( ODB.Crt, Crt );
}
if( ODB.#Crct = 0 )
{
    odb_add_link( ODB.Crct, Crct[EACH].ODB.document );
}
if( ODB.#Term = 0 )
{
    odb_add_link( ODB.Term, Term[EACH].ODB.document );
}
if( ODB.#Keyword = 0 )
{
    odb_add_link( ODB.Keyword, Keyword[EACH].ODB.document );
}
if( ODB.#Cites = 0 )
{
    odb_add_link( ODB.Cites, Cites[EACH].ODB.Cited_in );
}
odb_add_link( ODB.Bib_Document, Bib_Document[EACH].ODB.Bib_Entry );
}

DESCRIPTION
{
    ALWAYS
    {
        doc_1:
            begins( DOC_BEG BOUNDARY );
        doc_2:
            ends( BOUNDARY DOC_END );
    }
}
ENDDEF;

CLASS DEFINITION: Person
PERSISTENT INSTANCES
COMPONENTS
{
    Name : Person_Name 1..1;
    Other_Name : String 0..N;
    Institution : Institution 0..1;
    Address : Address 0..N;
    Phone : String 0..N;
}
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

```plaintext
Email : String 0..N;
Author_of : Document 0..N;
Editor_of : Document 0..N;

DB-INTERFACE
{
    ACCESS-KEYS
    {
        Name.sname + "", " + Initial( Name.fname ) + " " +
        Concat( Initial(Name.sname[EACH]) )
            very-strong;
        Name.sname + "", " + Initial( Name.fname )
            strong;
    }
    ACTIONS
    {
        if( not Found() )
        {
            odb_add_value( ODB.Name, Name.sname + "", " + Name.fname + " " +
                Name.mname[ALL] );

            odb_index_on( Name.sname + "", " + Name.fname + " " +
                Name.mname[ALL] );
            odb_index_on( Name.sname + "", " + Name.fname );
            odb_index_on( Name.sname + "", " + Initial(Name.fname) + " " +
                Concat(Initial(Name.mname[EACH])) );
            odb_index_on( Name.sname + "", " + Initial(Name.fname) );
            odb_index_on( Name.sname);
        }
        if( Found() and
            not similar-to(ODB.Name, Name.sname + "", " + Name.fname + " " + Name.mname[ALL], 90)
            and
            not similar-to(ODB.Other_Name[ANY],
                Name.sname + "", " + Name.fname + " " +
                Name.mname[ALL], 90)
        )
        {
            odb_add_value( ODB.Other_Name, Name.sname + "", " + Name.fname + " " +
                Name.mname[ALL] );
            odb_index_on( Name.sname + "", " + Name.fname + " " +
                Name.mname[ALL] );
        }
        if( ODB.#Institution = 0 )
        {
            odb_add_link( ODB.Institution, Institution[EACH].ODB.Person );
        }
        if( ODB.#Address = 0 and #Address > 0 )
```

170
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

```c
{
    odb_add_value( ODB.Address,
                   Address.street + "
" +
                   Address.city + ", " + Address.sbd + ", " +
                   Address.pc + "
" +
                   Address.country);
}
if( ODB.#Phone = 0 )
{
    odb_add_value( ODB.Phone, Phone );
}
if( ODB.#Email = 0 )
{
    odb_add_value( ODB.Email, Email );
}
}
}
ENDDEF;

CLASS DEFINITION: Institution
PERSISTENT INSTANCES
ATTRIBUTES
{
    Type : String;
}
COMPONENTS
{
    Name : String 1..1;
    Other_Name : String 0..N;
    Address : Address 0..N;
    Phone : String 0..N;
    Email : String 0..N;
    Person : Person 0..N;
    Document : Document 0..N;
    Publication : Publication 0..N;
}
ODB-INTERFACE
{
    ACCESS-KEYS
    {
        Subst( Name ) very-strong;
        Name very-strong;
    }
}

171
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

ACTIONS
{
    if( not found() )
    {
        odb_add_value( ODB.Name, Name );
        odb_index_on( Subst(Name) );
        odb_index_on( Name );
    }
    if( found() and not similar-to(ODB.Name, Name, 90) )
    {
        odb_add_value( ODB.Other_Name, Name );
        odb_index_on( Name );
    }
    if( ODB.#Type = 0 )
    {
        odb_add_value( ODB.Type, Type );
    }
    if( ODB.#Address = 0 and #Address > 0 )
    {
        odb_add_value( ODB.Address,
                        Address.street + "
                        Address.city + "," + Address.sbd + " 
                        Address.pc + "
                        Address.country );
    }
    if( ODB.#Phone = 0 )
    {
        odb_add_value( ODB.Phone, Phone );
    }
    if( ODB.#Email = 0 )
    {
        odb_add_value( ODB.Email, Email );
    }
    odb_add_link( ODB.Publication, Publication[EACH].ODB.Publisher );
}
ENDDEF;

CLASS DEFINITION: Publication
PERSISTENT INSTANCES
COMPONENTS
{
    Name : String 1..1;
    Other_Name : String 0..N;
    Publisher : Institution 0..1;
}
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

Frequency : String 0..1;
Document : Document 0..N;
}

ODB-INTERFACE
{
ACCESS-KEYS
{
    Subst( Name ) very-strong;
    Name very-strong;
}

ACTIONS
{
    if( not found() )
    {
        odb_add_value( ODB.Name, Name );
        odb_index_on( Subst(Name) );
        odb_index_on( Name );
    }
    if( found() and not similar-to(ODB.Name, Name, 90) )
    {
        odb_add_value( ODB.Other_Name, Name );
        odb_index_on( Name );
    }
    if( ODB.#Frequency = 0 )
    {
        odb_add_value( ODB.Frequency, Frequency );
    }
}
}

ENDDEF;

CLASS DEFINITION: Indexing_Term
PERSISTENT INSTANCES
COMPONENTS
{
    Term : String 1..1;
    Document : Document 0..N;
}

ODB-INTERFACE
{
    ACCESS-KEYS
    {
        Term very-strong;
    }
    ACTIONS
    {
        if( not found() )
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

{ 
    odb_add_value( ODB.Term, Term );
    odb_index_on( Term );
}

}

ENDDEF;

ENDFMTD;

C.2 Format Description for SGML Bibliographic Documents

/*******************************************************************/
/* File: bibs.dtd */
/* Descr.: DELTO description to be used for Envision SGML Bibs */
/* (previously converted to SGML with Denis Brueni's */
/* translators). */
/* Author: Guillermo Andres Averboch */
/* Dates: 05/17/95 (v 1.0) */
/* Compile: dll bibs */
/*******************************************************************/

USE: "general";
LEX: "lx.bibs";

CLASS DEFINITION: Document
    PERSISTENT INSTANCES
    DESCRIPTION
    {
        ALWAYS
        {
            contains( <<bib>>[@1.Item_type := "Bib"] );
        }
    }
    ENDFDEF;

CLASS DEFINITION: Person_Name
    COMPONENTS
    {
        affil : String 0..1;
    }
    DESCRIPTION
    {
        ALWAYS
        {
            pattern("$<au>$" BOUNDARY
                    "$<name>$"?

174
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

("$<role>$" ANYTHING* "$</role>$")?
("$<xnm>$" ANYTHING* "$</xnm>$" )*
("$<fnm>$" fname[] "$</fnm>$")?
("$<mnm>$" mname[] "$</mnm>$" )*
("$<snm>$" sname[] "$</snm>$")
("$<xnm>$" ANYTHING* "$</xnm>$" )*
("$<affil>$" affil[] "$</affil>$")?
"$/<name>$"?
BOUNDARY "$</au>$"

OR

pattern( "$<au>$" BOUNDARY
 "$<name>$" )?
("$<role>$" ANYTHING* "$</role>$")?
"$<tnm>$" sname[] "$</tnm>$" 
("$<affil>$" affil[] "$</affil>$")?
"$/<name>$"?
BOUNDARY "$</au>$"

OR

pattern( "$<editor>$" BOUNDARY
 "$<name>$" )?
("$<role>$" ANYTHING* "$</role>$")?
("$<xnm>$" ANYTHING* "$</xnm>$" )*
("$<fnm>$" fname[] "$</fnm>$")?
("$<mnm>$" mname[] "$</mnm>$" )*
("$<snm>$" sname[] "$</snm>$")
("$<xnm>$" ANYTHING* "$</xnm>$" )*
("$<affil>$" affil[] "$</affil>$")?
"$/<name>$"?
BOUNDARY "$</editor>$"

OR

pattern( "$<editor>$" BOUNDARY
 "$<name>$" )?
("$<role>$" ANYTHING* "$</role>$")?
"$<tnm>$" sname[] "$</tnm>$" 
("$<affil>$" affil[] "$</affil>$")?
"$/<name>$"?
BOUNDARY "$</editor>$" );
}
ENDDEF;
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

CLASS DEFINITION: Corporate_Name
DESCRIPTION
{
  ALWAYS
  {
    pattern( "<$au>$" BOUNDARY
      "<$name>$"?
      ("<$role>$" ANYTHING* "</role>$")?
      "<$cnn>$" cnme[] "</cnn>$"
      ("<$affil>$" ANYTHING* "</affil>$" )?
      "$</name>$"?
      BOUNDARY "$</au>$" ) ) VERY-STRONG INDICATOR;
  }
}
ENDDEF;

CLASS DEFINITION: Title
COMPONENTS
{
  title_text : String 1..N;
}
DESCRIPTION
{
  ALWAYS
  {
    pattern( "<$title>$"
      ( title_text[]
        ("<$subtitle>$" title_text[] "</subtitle>$")*
      )
      "$</title>$" );
  }
}
ENDDEF;

CLASS DEFINITION: Date
COMPONENTS
{
  weekday : String 0..1;
}
DESCRIPTION
{
  ALWAYS
  {
    pattern( "<$date>$"
      ( "<$weekday>$" weekday[] "</weekday>$" )?
      ( "<$day>$" day[] "</day>$" )?
      ( "<$month>$" month[] "</month>$" )?

176
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

```
( "$<year>$"  year[]  "$</year>$" )?
"$</date>$" );
}
ENDDEF;

CLASS DEFINITION: Address
COMPONENTS
{
    odv       : String 0..N;
    onm       : String 0..1;
}
DESCRIPTION
{
    ALWAYS
    {
        pattern( "$<address>$"
            ( "$<name>$"       ANYTHING* "$</name>$" )*
            ( "$<odv>$"  odv[]  "$</odv>$" )*
            ( "$<onm>$"  onm[]  "$</onm>$" )?
            ( "$<street>$" street[]  "$</street>$" )?
            ( "$<city>$"  city[]  "$</city>$" )?
            ( "$<sbd>$"  sbd[]  "$</sbd>$" )?
            ( "$<pc>$"  pc[]  "$</pc>$" )?
            ( "$<country>$" country[]  "$</country>$" )?
        "$</address>$" );
    }
ENDDEF;

CLASS DEFINITION: Publisher_Name
DESCRIPTION
{
    ALWAYS
    {
        pattern( "$<pub>$"  pname "$</pub>$" );
    }
ENDDEF;

CLASS DEFINITION: Publication_Name
DESCRIPTION
{
    ALWAYS
    {
        pattern( <<sgml_tag_in>>
            ( <<Title>>[pname:=title_text[ALL]] | pname )
```

177
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

"$</in>$" ;

} ENDDEF;

Class Definition: CrCat
Components
{
    Term : String 1..1;
    Indexing.Term : Indexing.Term 1..1;
}
Description
{
    always
    {
        pattern( "$<crcat>$" term[] "$</crcat>$" );
    }
}
EndDef;

Class Definition: Term
Components
{
    Term : String 1..1;
    Indexing.Term : Indexing.Term 1..1;
}
Description
{
    always
    {
        pattern( "$<term>$" term[] "$</term>$" );
    }
}
EndDef;

Class Definition: Keyword
Components
{
    Term : String 1..1;
    Indexing.Term : Indexing.Term 1..1;
}
Description
{
    always
    {
        pattern( "$<keyword>$" term[] "$</keyword>$" );
    }
}
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

CLASS DEFINITION: Bib_Entry
COMPONENTS
{
  item_type : String 1..1;
  icode : Numeric 1..1;
  author : Person 0..N;
  institution : Institution 0..N;
  title : String 1..1;
  edition : String 0..1;
  series : String 0..1;
  publication : Publication 0..1;
  editor : Person 0..N;
  publisher : Institution 0..1;
  date : Date 0..1;
  freq : String 0..1;
  volume : String 0..1;
  number : String 0..1;
  chapter : String 0..1;
  pp : String 0..1;
  issn : String 0..1;
  isbn : String 0..1;
  coden : String 0..1;
  libcon : String 0..1;
  cft : String 0..N;
  ccrat : Indexing_Term 0..N;
  term : Indexing_Term 0..N;
  keyword : Indexing_Term 0..N;
  abstract : String 0..1;
  Document : Document 0..1;
}

DESCRIPTION
{
  ALWAYS
  {
    pattern( ( <<sgml_tag_b_art>> [item_type := "JrnArt", icode:= 1] |  
               <<sgml_tag_b_jrnart>>[item_type := "JrnArt", icode:= 1] |  
               <<sgml_tag_b_inbook>>[item_type := "InBook", icode:= 2] |  
               <<sgml_tag_b_inproc>>[item_type := "InProc", icode:= 3] |  
               <<sgml_tag_b_bklet>> [item_type := "Bklet", icode:= 5] |  
               <<sgml_tag_b_bcast>> [item_type := "Bcast", icode:= 6] |  
               <<sgml_tag_b_chart>> [item_type := "Chart", icode:= 7] |  
               <<sgml_tag_b_softw>>[item_type := "Softwr", icode:= 8] |  
               <<sgml_tag_b_editor>>[item_type := "Editor", icode:= 9] |  
    )}
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

<<sgml_tag_b_film>> [item_type := "Film", icode:=10] |
<<sgml_tag_b_legal>> [item_type := "Legal", icode:=11] |
<<sgml_tag_b_letter>> [item_type := "Letter", icode:=12] |
<<sgml_tag_b_map>> [item_type := "Map", icode:=14] |
<<sgml_tag_b_perf>> [item_type := "Perf", icode:=16] |
<<sgml_tag_b_comm>> [item_type := "Comm", icode:=17] |
<<sgml_tag_b_phd>> [item_type := "Phd", icode:=18] |
<<sgml_tag_b_rec>> [item_type := "Rec", icode:=19] |
<<sgml_tag_b_review>> [item_type := "Review", icode:=20] |
<<sgml_tag_b_tr>> [item_type := "TR", icode:=21] |
<<sgml_tag_b_unpub>> [item_type := "Unpub", icode:=22] |
<<sgml_tag_b_inprog>> [item_type := "InProg", icode:=23] |

("$<no>$" ANYTHING+ "$</no>$")?
("$<au>$" ( <<Person_Name>>

[! Person[similar-to(name.sname,sname,90) and
   Initial(name.fname)=Initial(fname)].
   name[] := $0,
   Author[] := !.Person[LAST],
   person[] := !.Person[LAST]]
  |
  <<Corporate_name>>
  [! Institution[similar-to(name,cname,90)
   or
   similar-to(subst(name),
   subst(cname),90)].
   name[] := cname,
   Institution[] := !.Institution[LAST],
   inst_object[] := !.Institution[LAST]])

"$</au>$")*
<<Title>>[title := title_text[ALL]]?
("$<edition>$" edition[] "$</edition>$")?
("$<series>$"
 ( <<Title>>[series[] := title_text[ALL]] |
   series[] )
"$</series>$")?

<<publication_name>>
[! publication[similar-to(name,pname,90)
   or
   similar-to(subst(name),subst(pname),90)].
   name[] := pname,
   publication[] := !.publication[LAST],

180
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

publ_object[] := !.publication[LAST]

("$<editor>$"
  <<Person_Name>>
    [!.Person[similar-to(name.sname,sname,90) and
        Initial(name.fname)=Initial(fname)].name:=0,
        Editor[] := !.Person[LAST],
        Person[] := !.Person[LAST]]
  "$</editor>$")*
<<publisher_name>>
  [!.Institution[similar-to(name,pname,90)
      or
        similar-to(subst(name),subst(pname),90)]
    .name[] := pname,
    Publisher[] := !.Institution[LAST],
    inst_object[] := !.Institution[LAST]]?
<<Address>>[!.Institution[LAST].address := 4]?

( <<Date>>[date[] := 00] )?

("$<freq>$"  freq[]  "$</freq>$")?
("$<volume>$"  volume[]  "$</volume>$")?
("$<number>$"  number[]  "$</number>$")?
("$<inchap>$"  chapter[]  "$</inchap>$")?
("$<pp>$"  pp[]  "$</pp>$")?
("$<issn>$"  issn[]  "$</issn>$")?
("$<isbn>$"  isbn[]  "$</isbn>$")?
("$<coden>$"  coden[]  "$</coden>$")?
("$<libcon>$"  libcon[]  "$</libcon>$")?

ANYTHING*  /* s.obi */

("$<crt>$"  crt[]  "$</crt>$")?

("$<cat>$"
  <<rcat>>[!..Indexing_Term[term=01.term].term:=00.term,
      01.rcat[] := !..Indexing_Term[LAST],
      00.indexing_term[] := !..Indexing_Term[LAST]]*
  "$</cat>$")?

("$<terms>$"
  <<term>>[!..Indexing_Term[term=01.term].term:=00.term,
      01.term[] := !..Indexing_Term[LAST],
      00.indexing_term[] := !..Indexing_Term[LAST]]*

181
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

"$</terms>$"
)

("$<keywords>$" <<keyword>> ![.Indexing_Term[term=01.term].term:=01.tern,
Q1.keyword[] := !.Indexing_Term[LAST],
Q0.indexing_term[] := !.Indexing_Term[LAST]]*
"$</keywords>$"
)

("$<abstract>$" abstract[] "$</abstract>$")?

("$<b.art>$" | "$</b.jrnart>$" | "$</b.inbook>$" | "$</b.proc>$" | "$</b.book>$" | "$</b.bklet>$" | "$</b.chart>$" | "$</b.softwr>$" | "$</b.editr1>$" | "$</b.film>$" | "$</b.legal>$" | "$</b.letter>$" | "$</b.manual>$" | "$</b.map>$" | "$</b.perf>$" | "$</b.phd>$" | "$</b.rec>$" | "$</b.review>$" | "$</b.tr>$" | "$</b.unpub>$" | "$</b.inprog>$" | "$</b.artvkrk>$")

)

ENDDEF;

CLASS DEFINITION: Bib
DESCRIPTION
{
  ALWAYS
  {
    irrelevant( <nl> )
  }
  pattern ( "$<bib>$"
  ( <<Bib_Entry>>
    [ !.Document[].Title      := title,
      !.Document[LAST].Item_type := item_type,
      !.Document[LAST].Published_in := publication,
      !.Document[LAST].Publisher := publisher_name,
      !.Document[LAST].Edition := edition,
      !.Document[LAST].Series := series,
      !.Document[LAST].Date := date,
      !.Document[LAST].Volume := volume,
      !.Document[LAST].Number := number,
      !.Document[LAST].PP := pp,
      !.Document[LAST].Issn := issn,
      !.Document[LAST].Isbn := isbn,
      !.Document[LAST].Coden := coden,
      !.Document[LAST].Libcon := libcon,
      !.Document[LAST].Crt := crt,
  )
  )
  
182
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

! .Document[LAST].Crcat := crcat[EACH],
! .Document[LAST].Term := term[EACH],
! .Document[LAST].Keyword := keyword[EACH],
! .Document[LAST].Abstract := abstract,

! .Document[LAST].Author := author[EACH],
author[EACH].Author_of := ! .Document[LAST],
! .Document[LAST].Editor := editor[EACH],
editor[EACH].Editor_of := ! .Document[LAST],
publisher[1].Frequency := freq,
document := ! .Document[LAST]]

ANYTHING*
)*
"$</bib>$" );
}
}

ENDDEF;

class definition: sgml_tag_b_jrnart
define;
class definition: sgml_tag_b_art
define;
class definition: sgml_tag_b_inbook
define;
class definition: sgml_tag_b_inproc
define;
class definition: sgml_tag_b_book
define;
class definition: sgml_tag_b_bklet
define;
class definition: sgml_tag_b_bcast
define;
class definition: sgml_tag_b_chart
define;
class definition: sgml_tag_b_softwr
define;
class definition: sgml_tag_b_liter
define;
class definition: sgml_tag_b_films
define;
class definition: sgml_tag_b_legal
define;
class definition: sgml_tag_b_letter
define;
class definition: sgml_tag_b_manual
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

enddef;
class definition: sgml_tag_b_map
enddef;
class definition: sgml_tag_b_thesis
enddef;
class definition: sgml_tag_b_perf
enddef;
class definition: sgml_tag_b_comm
enddef;
class definition: sgml_tag_b_phd
enddef;
class definition: sgml_tag_b_rec
enddef;
class definition: sgml_tag_b_review
enddef;
class definition: sgml_tag_b_tr
enddef;
class definition: sgml_tag_b_unpub
enddef;
class definition: sgml_tag_b_inprog
enddef;
class definition: sgml_tag_b_artwrk
enddef;

class definition: sgml_tag_in
enddef;
class definition: sgml_tag_crcat
enddef;

ENDfmtd

C.3 Format Description for Psuedo-SGML Full-Text Articles

/-------------------------------------------------------------------*/
/* File: wav.gral.dlt */
/* Date: 05/04/95 */
/* Description: DELTO general description for Waverly SML documents */
/* Uses: general.dlt */
/-------------------------------------------------------------------*/

Use: "general";
Lex: "lx.sgml";

/-------------------------------------------------------------------*/
/* Class Definitions for needed lexical analyzer tokens */
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

---

Class Definition: sgml_tag_tu
EndDef;

Class Definition: sgml_tag_vsp
  Components
  {
    sp : Numeric 1..1;
  }
EndDef;

Class Definition: sgml_tag_p
EndDef;

Class Definition: sgml_tagssf
EndDef;

Class Definition: sgml_tagscp
EndDef;

Class Definition: sgml_tag_it
EndDef;

Class Definition: sgml_tag_b
EndDef;

Class Definition: sgml_tag_hsp
EndDef;

Class Definition: sgml_tag_hmk
EndDef;

Class Definition: sgml_tag_hmk
EndDef;

Class Definition: sgml_tag_tbl
EndDef;

Class Definition: sgml_start_tag
EndDef;

Class Definition: sgml_end_tag
EndDef;

Class Definition: Numbering
  Components
  {

185
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

```
level : Numeric 1..1;
numbers : String 1..1;
}
EndDef;

Class Definition: Text_Structure_Indicator
  Components
  {
    level : Numeric 1..1;
  }
EndDef;

Class Definition: Word
  Components
  {
    case : Numeric 1..1;
  }
EndDef;

Class Definition: University_Name
EndDef;

Class Definition: Journal_Name
EndDef;

Class Definition: Month
EndDef;

Class Definition: Number
  Attributes
  {
    Value : Numeric;
  }
EndDef;

/*---------------------------------------------------------------*/
/* Class Definitions for objects to be recognized by the */
/* Analyzer (and not by the lexical analyzer) */
/*---------------------------------------------------------------*/
Class Definition: Parag_Sep
  Description
  {
    always
    {
      pattern( BOUNDRY <<nl>>2..N BOUNDARY ANYTHING:[nl] )
      or
      pattern( <<nl>>* <<docum_end>> );
```
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

```plaintext
never
{
    begins( <<nl>> BOUNDARY );
}
}
EndDef;

Class Definition: Font_Tag
Description
{
    always
    {
        font_tag_1:
        is_a( <<sgml_tag_sff>> )
        or
        is_a( <<sgml_tag_scp>> )
        or
        is_a( <<sgml_tag_it>> )
        or
        is_a( <<sgml_tag_b>> );
    }
}
EndDef;

Class Definition: Font_End_Tag
Description
{
    always
    {
        font_end_tag_1:
        pattern( "$</sff>$" | "$</scp>$" | "$</it>$" | "$</b>$" );
    }
}
EndDef;

Class Definition: Title_Unit
Components
{
    sp : Numeric 0..N;
}
Description
{
    always
    {
        title_unit_1:
        pattern( <<sgml_tag_tu>> <<sgml_tag_vsp>>[@1.sp[] := sp]*
```
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

\[
\langle\text{n1}\rangle \langle\text{sgml_tag\_tu}\rangle \langle\text{sgml_tag\_vsp}\rangle
\]
\[
[\text{t1.sp[]} := \text{sp}^*]?
\]
\[
\langle\text{n1}\rangle? \text{OBJECT (}\langle\text{parag\_sep}\rangle \mid \langle\text{sgml_tag\_tu}\rangle)\rangle;
\]
\]
\]
EndDef;

Class Definition: Title
Components
{
  title_text : String 1..2;
  level : Numeric 0..1;
}
Description
{
  always
  {
    title_1:
      is_a( \langle\text{title\_unit}\rangle );
    title_2:
      contains( \langle\text{sgml\_tag\_p}\rangle ANYTHING 0..2
        ( \langle\text{numbering}\rangle[\text{t1.level} := \text{level},
          \text{t1.title\_text} := \text{text}] \mid
          \langle\text{text\_structure\_indicator}\rangle[\text{t1.level} := \text{level}]
        ) );
  }
  frequently
  {
    title_3:
      ends( \langle\text{sgml\_tag\_hsp}\rangle
        \text{title\_text[]} \text{BOUNDARY } \langle\text{parag\_sep}\rangle );
    title_4:
      pattern( \langle\text{title\_unit}, \text{sp}[\text{ANY}] \geq 12\rangle );
  }
  never
  {
    title_5:
      contains( \langle\text{open\_clb}\rangle \mid \langle\text{close\_clb}\rangle );
  }
}
EndDef;

Class Definition: Person_Name
Description
{
  always
  {


188
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

pname_1:
  irrelevant( <<sgml_start_tag>> );
}
frequently
{
  pname_2:
  pattern( (<<given_name>>[01.fname := text] | 
    <<initial>>[01.fname := text]) 
  (<<given_name>>[01.mname := text] | 
    <<initial>>[01.mname := text][1..2] 
    <<last_name>>[01.sname := text][1..2] ) 
  very-strong indicator;
  pname_3:
  not ends( <<given_name>> ) weak indicator;
}
sometimes
{
  pname_4:
  pattern( (<<given_name>>[01.fname := text] | 
    <<initial>>[01.fname := text]) 
  (<<last_name>>[01.sname := text][1..2] ) 
  very-strong indicator;
  pname_5:
  pattern( (<<given_name>>[01.fname := text] | 
    <<initial>>[01.fname := text]) 
  (<<given_name>>[01.mname := text] | 
    <<initial>>[01.mname := text][1..2] 
    <<word, case = 1 or case = 2, 
    not intersects(<<last_name>>)>> 
    [01.sname := text][1..2] ) 
  very-strong indicator;
}
EndDef;

Class Definition: Author
Components
{
  pname : Person_Name 1..N;
}
Description
{
  always
  {
    author_1:
    pattern( <<font_tag>> ANYTHING 0..2 
      <<person_name>>[01.pname := 0] 
      ... 
  }
}
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

(ANYTHING 0..2 <<person_name>>[01.fname := o])*
 ANYTHING 0..2 <<font_end_tag>> ;
}
}
EndDef;

Class Definition: University_Name_90
Description
{
 always
{
 is_a( <<university_name>> );
 pattern( <<university_name, weight := 90>> );
}
}
EndDef;

Class Definition: Author_Inst
Components
{
 inst_name : String 1..1;
 inst_type : String 1..1;
}
Description
{
 always
{
 Author_Inst_1:
 pattern( <<font_tag>> ANYTHING 0..5
 <<university_name_90>>
 [01.inst_name := e.text,
 01.inst_type := "univ"] );
}
}
EndDef;

Class Definition: Abstract
Description
{
 always
{
 abstract_1:
 begins( <<sgml_tag_tu>><<sgml_tag_vsp, sp=4>><<n1>>
 <<sgml_tag_tu>><<sgml_tag_vsp, sp=10>> );
 abstract_2:
 ends( BOUNDARY <<keyword_section>> );
}
}
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

}  
EndDef;

Class Definition: Keyword_Section

Components
{
  crcat : Indexing_Term 0..N;
  term : Indexing_Term 0..N;
  phrase : Indexing_Term 0..N;
}

Description
{
  always
  {
    keyword_sec_1:
      irrelevant( <<nl>> );
    keyword_sec_2:
      pattern( "Categories and Subject Descriptors:" 
        (ANYTHING* <<font_tag>> crcat[] . term[] <<font_end_tag>>)* 
        <<parag_sep>>
        <<sgml_start_tag>>0..4
        "General Terms:" term[] . term[] (<<comma>> term[] . term[])* 
        <<parag_sep>>
        <<sgml_start_tag>>0..4
        "Additional Key Words and Phrases:" phrase[] . term[] (<<comma>> phrase[] . term[])* 
        <<parag_sep>>
      );
  }
}
EndDef;

Class Definition: Date

Description
{
  always
  {
    pattern( <<month>>[month[] := 0.text] 
      <<number, value > 1950>>[year[] := 0.text] );
  }
}
EndDef;

Class Definition: Front_Matter

Components
{
  title : String 1..1;
  author : Person 0..N;

  191
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

abstract : Abstract 0..1;
keywords : Keyword_Section 0..1;

Description
{
  always
  {
    fm_1:
      pattern( DOC_BEGIN OBJECT <<title>> );
  }
  frequently
  {
    fm_2:
      contains( <<font_tag>> title[] <<font_end_tag>>,

        <<author, pos = !.author[EACH].pos>>
          [!.Person[].name := pname,
           !.Person[LAST].author_of := !.Document[1],
           !.author := !.Person[LAST] ]

          ( ANYTHING*
            ( <<author_inst>>
              [!.Institution[].name := !.inst_name,
               !.Institution[LAST].type := !.inst_type,
               !.Institution[LAST].person := !.Person[LAST],
               !.Institution[LAST].document := !.Document[1],
               !.Person[LAST].institution := !.Institution[LAST] ]
            )
        )
      )
  }
}

ANYTHING*
( <<abstract>> | <<keyword_section>> )

fm_3:
  contains( <<abstract>>[01.abstract := 0] );
fm_4:
  contains( <<keyword_section>>[01.keywords := 0] );
}

EndDef;

Class Definition: Back_Matter

192
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

Components
{
    journal_name : String 0..1;
    volume        : String 0..1;
    number        : String 0..1;
    date          : Date 0..1;
    pp            : String 0..1;
    table         : Table 0..N;
    figure_caption: Figure_Caption 0..N;
}

Description
{
    always
    {
        bm_1:
        
        pattern( BOUNDARY "REFERENCES" <<font_end_tag>>
                ANYTHING* BOUNDARY <<docum_end>> );
    }
    frequently
    {
        bm_2:
        
        contains( <<title_unit,
                  begins( <<journal_name>>[.2.journal_name := text]
                  <<comma>> "Vol." .1.volume
                  <<comma>> "No." .1.number
                  <<comma>> <<date>>[.2.date := #]
                  <<comma>> "Pages" .1.pp <<period>> ) >> );
    }
    sometimes
    {
        bm_3:
        
        contains( table[] (ANYTHING* table[])* ANYTHING* "Vol." );
        bm_4:
        
        contains( figure_caption[] (ANYTHING* figure_caption[])*
                  ANYTHING* "Vol." );
    }
}

EndDef;

Class Definition: Bib

Description
{
    always
    {
        bib_1:
        
        begins( "REFERENCES" <<font_end_tag>> );
        bib_2:
        
        ends( "REFERENCES" <<font_end_tag>> );
    }
}

193
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

    ends( BOUNDARY
         <<sgml_tag_hmrk>> <<sgml_tag_tu>> <<sgml_tag_vsp>> );
    bib_3:
        in( <<back_matter>> );
    }
EndDef:

Class Definition: Body
    Description
    {
        always
        {
            body_1:
                pattern( <<front_matter>> OBJECT <<back_matter>>? <<docum_end>> );
        }
    }
EndDef;

Class Definition: Level_0_Title
    Description
    {
        always
        {
            level0_1:
                is_a( <<title>> );
            level0_2:
                pattern( <<title, level = 0>> );
        }
    }
EndDef;

Class Definition: Section
    Description
    {
        always
        {
            section_1:
                in( <<body>> );
            section_2:
                pattern( BOUNDARY <<level_0_title>> ANYTHING* BOUNDARY
                         (<<level_0_title>> | <<back_matter>> | <<docum_end>>));
        }
    }
EndDef;

194
Class Definition: Level_1_Title
  Description
  { always
    { level1_1:
      is_a( <<title>> );
      level1_2:
        pattern( <<title, level = 1>> );
    }
  }
EndDef;

Class Definition: Sub_Section
  Description
  { always
    { sub_section_1:
      in( <<body>> );
      sub_section_2:
        pattern( BOUNDARY <<level_1_title>> ANYTHING* BOUNDARY
          ( <<level_0_title>> | <<level_1_title>> !
            <<back_matter>> | <<docum_end>>
          )
        );
    }
  }
EndDef;

Class Definition: Paragraph
  Description
  { always
    { par_1:
      in( <<body>> )
      or
      in( <<back_matter>> );
      par_2:
        begins( <<parag_sep>> BOUNDARY );
      par_3:
        ends( BOUNDARY <<parag_sep>> );
    }
  frequently
  { par_4:
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

    not intersects( <<title>> );
  }
}
EndDef;

Class Definition: Bib_Entry
Components
{
  label : String 1..1;
  title : String 1..1;
  document : Document 1..1;
}
Description
{
  always
  {
    bib_entry_1:
      is_a( <<paragraph>> );
    bib_entry_2:
      in( <<bib>> );
    bib_entry_3:
      irrelevant( <<nl>> );

    bib_entry_4:
      begins ( <<sgml_tag_p>> ANYTHING 0..3
        ( <<number>>[@1.label := text] |
          <<numbering, level = 0>>[@1.label := numbers] )
        ANYTHING 0..2
        ( <<sgml_tag_hmk>> | <<sgml_tag_hmkr>> )

        <<word, case=1 or case=2>>
          [@1.Person_Name[].sname := text,
            .Person[name.sname[@1.text].name :=
                .!Person[Last],
            .Document[].author := .!Person[LAST],
            Document := .!Document[LAST]]
        } <<comma>>

      <<word, case=1 or case=2>>
        [@1.Person_Name[Last].fname := text]
      <<word, case=1 or case=2>>
        [@1.Person_Name[Last].mname := text]0..2

      ( <<comma>> (<<nl>>? "and"? <<nl>>?)
        <<word, case=1 or case=2>>
          [@1.Person_Name[].sname := text,
          ...}
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

! .Person[sname=S1.text].name :=
    ! .Person[Name][LAST],
]
<<comma>>
<<word, case=1 or case=2>>
    [!.Person_Name[Last].fname := text]
<<word, case=1 or case=2>>
    [!.Person_Name[Last].mname := text] 0..2
)*

ANYTHING 0..2
<<sgml_start_tag>>+
title
( <<sgml_end_tag>> | <<period>> )
);

EndDef;

Class Definition: Bib_Ref
Components
{
    bib_entry : Bib_Entry 1..N;
}
Description
{
    always
    {
        bib_ref_1:
        in( <<body>> );
        bib_ref_2:
        irrelevant( <<nl>> );
        bib_ref_3:
        irrelevant( <<sgml_start_tag>> );
        bib_ref_4:
        pattern( <<open_sqbr>>
            <<number>>
            [S1.bib_entry := !.Bib_Entry[label = S1.text]]
        ( <<comma>>
            <<number>>
            [S1.bib_entry := !.Bib_Entry[label = S1.text]]
        )*  
        <<close_sqbr>> );
    }

EndDef;

197
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

Class Definition: Table_End_Mark
    Description
    {
    always
    {
        tbl_end_1:
        pattern( <<ltthan>> <<inv_bar>> "tbl" <<gthan>> );
    }
    }
EndDef;

Class Definition: Column_Mark
    Description
    {
    always
    {
        col_mark_1:
        pattern( "$\langle\rangle$" );
    }
    }
EndDef;

Class Definition: Table
    Components
    {
    title : String 0..N;
    header : String 0..N;
    item : String 0..N;
    }
    Description
    {
    always
    {
        table_1:
        begins( <<sgml_tag_tbl>> );
        table_2:
        ends( <<table_end_mark>> );
    }
    frequently
    {
        table_3:
        begins( <<sgml_tag_tbl>> title[] <<sgml_tag_hsp>> title[] <<sgml_tag_vsp>> <<parag_sep>>[header[] := "<row>", header[] := "<c>"
        &Mark1 ]
        table_4:
        contains( *Mark1

198
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

header[] (<<column_mark>>
    [header[] := "</c>",
     header[] := "<c>"]
    header[])*
)

<parag_sep>
    [header[] := "</c>",
     header[] := "</row>",
     item[] := "<row>",
     item[] := "<c>"
        &Mark2);

    contains(*Mark2
        item[] (<<column_mark>>
            [item[] := "</c>",
             item[] := "<c>"]
            item[])*
        ( <<n1>[item[] := "</c>",
             item[] := "</row>",
             item[] := "<row>",
             item[] := "<c>"
                <<n1>>*
         item[] (<<column_mark>>
                [item[] := "</c>",
                 item[] := "<c>"
                    item[])*
        )*
     )<table_end_mark>[item[] := "</c>",
         item[] := "</row>"]

EndDef;

Class Definition: Figure_Caption
    Description
    {
        always
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

{  
  fig_caption_i:  
  pattern( BOUNDARY  
           "Fig." <<numbering>> <<gml_tag_hsp>> ANYTHING*  
           BOUNDARY <<parag_sep>> );  
}

EndDef;

Class Definition: Document  
Components  
{
  front_matter : front_matter 1..1;  
  body : body 1..1;  
  back_matter : back_matter 0..1;  
  table : table 0..N;  
  figure_caption : figure_caption 0..N;  
  bib : bib 0..1;  
}

Description  
{
  always
  {
    doc_1:  
      begins( DOC_BEG BOUNDARY );  
    doc_2:  
      ends( BOUNDARY DOC_END );  
    doc_3:  
      contains( <<front_matter>>  
        [01.front_matter := 00,  
         01.title := Clear-String(title),  
         01.item_type := "JrnArt",  
         01.author := author[Each],  
         01.abstract := Clear-String(abstract.text),  
         01.crcat[] := keywords.crcat[Each],  
         01.term[] := keywords.term[Each],  
         01.keyword[] := keywords.phrase[Each]]  
      <<body>>  
        [01.body := 00],  
      <<back_matter>>  
        [01.back_matter := 00,  
         !.Publication[].name := journal_name,  
         !.Publication[Last].name := "ACK",  
         01.published_in := !.Publication[Last],  
         01.date := date,  
         01.volume := volume,  
         01.number := number,}  

200
APPENDIX C. DELTO FORMAT DESCRIPTIONS USED FOR ENVISION

01.pp := Clear-String(pp),
01.table := table[EACH],
01.figure_caption := figure_caption[EACH]?
);
}
frequently {

doc_4:
    contains( <<bib_entry, pos <= bib_entry[EACH].pos>>
        [00.document[1].title := Clear-String(00.title),
        00.document[1].cited_in := 01,
        01.cites := 00.document]
        ( ANYTHING* <<bib_entry>>
            [00.document[1].title :=
            Clear-String(00.title),
            00.document[1].cited_in := 01,
            01.cites := 00.document]
        )* );

doc_5:
    contains( <<bib>>[01.bib:=0] );
}
}
EndDef;

EndFmtd:
Appendix D

RETARGET SPECIFICATIONS USED FOR ENVISION

We wrote two Retarget specifications for our evaluation process. These two specifications are used for generating Envision data. We present these two Retarget specifications in this Appendix. The first one of them, presented in Section D.1 generates SGML bibliographic documents with information about links, from the results of processing SGML bibliographic documents with the DELTO format description shown in Appendix C.2. The second Retarget specification generates SGML full-text articles conforming to the Envision DTD from the results of processing pseudo-SGML full-text articles with the DELTO format description presented in Appendix C.3. The second Retarget specification is presented in Section D.2.

D.1 Retarget Specification for SGML Bibliographic Documents

{begin document}
{output_width := 78;}
<!doctype document SYSTEM "envision.dtd">
<DOCUMENT>
<ObjectID>{odbid}</ObjectID>

{process( bib_entry > 1 with bib_entry );
  ignore_text( sgm1_markup_decl );}

</DOCUMENT>
{end document}

{begin bib_entry}

{if icode = 1}
APPENDIX D. RETARGET SPECIFICATIONS USED FOR ENVISION

```xml
<b.jrnart id="{document.obid}">
{endif}
{if icode = 2}
<b.inbook id="{document.obid}">
{endif}
{if icode = 3}
<b.inproc id="{document.obid}"/>
{endif}
{if icode = 4}
<b.book id="{document.obid}"/>
{endif}
{if icode = 5}
<b.bklet id="{document.obid}"/>
{endif}
{if icode = 6}
<b.bcast id="{document.obid}"/>
{endif}
{if icode = 7}
<b.chart id="{document.obid}"/>
{endif}
{if icode = 8}
<b.softwr id="{document.obid}"/>
{endif}
{if icode = 9}
<b.editrl id="{document.obid}"/>
{endif}
{if icode = 10}
<b.film id="{document.obid}"/>
{endif}
{if icode = 11}
<b.legal id="{document.obid}"/>
{endif}
{if icode = 12}
<b.letter id="{document.obid}"/>
{endif}
{if icode = 13}
<b.manual id="{document.obid}"/>
{endif}
{if icode = 14}
<b.map id="{document.obid}"/>
{endif}
{if icode = 15}
<b.thesis id="{document.obid}"/>
{endif}
{if icode = 16}
<b.perf id="{document.obid}"/>
{endif}
```
APPENDIX D. RETARGET SPECIFICATIONS USED FOR ENVISION

{if icode = 17}
  <b:comm id="{document.odbid}"/>
{endif}

{if icode = 18}
  <b:phd id="{document.odbid}"/>
{endif}

{if icode = 19}
  <b:rec id="{document.odbid}"/>
{endif}

{if icode = 20}
  <b:review id="{document.odbid}"/>
{endif}

{if icode = 21}
  <b:tr id="{document.odbid}"/>
{endif}

{if icode = 22}
  <b:unpub id="{document.odbid}"/>
{endif}

{if icode = 23}
  <b:inprog id="{document.odbid}"/>
{endif}

{if icode = 24}
  <b:artwrk id="{document.odbid}"/>
{endif}

{process( person_name with person_name,
corporate_name with corporate_name,
publisher_name with publisher_name,
publication_name with publication_name,
crcat with indexing_term_template,
term with indexing_term_template,
keyword with indexing_term_template );

ignore_text( sgml_tag_b_jrnart, sgml_tag_b_art, sgml_tag_b_inbook,
sgml_tag_b_inproc, sgml_tag_b_book, sgml_tag_b_bklet,
sgml_tag_b_bcast, sgml_tag_b_graph, sgml_tag_b_softwr,
sgml_tag_b_editr, sgml_tag_b_film, sgml_tag_b_legal,
sgml_tag_b_letter, sgml_tag_b_manual, sgml_tag_b_map,
sgml_tag_b_thesis, sgml_tag_b_perf, sgml_tag_b_comm,
sgml_tag_b_phd, sgml_tag_b_rec, sgml_tag_b_review,
sgml_tag_b_tr, sgml_tag_b_unpub, sgml_tag_b_inprog,
sgml_tag_b_artwrk );
}
{end}

{begin person_name}
<ANCHOR objectid="{person.odbid}">{text}</ANCHOR>
{end}
D.2 Retarget Specification for Pseudo-SGML Full-text Articles

{begin document}
{output_width := 78}
{ignore_input_sgml := TRUE}
<!DOCTYPE ARTICLE PUBLIC "-//EY Article//EN" "Article.Rules"
[
<!ENTITY Beta "[Beta]" >
<!ENTITY Delta "[Delta]" >
<!ENTITY Gamma "[Gamma]" >
<!ENTITY Lambda "[Lambda]" >
<!ENTITY Omega "[Omega]" >
<!ENTITY Phi "[Phi]" >
<!ENTITY Psi "[Psi]" >
<!ENTITY Sigma "[Sigma]" >
<!ENTITY Theta "[Theta]" >
<!ENTITY alpha "[alpha]" >
<!ENTITY amp "&" >
<!ENTITY beta "[beta]" >
<!ENTITY chi "[chi]" >
<!ENTITY delta "[delta]" >
<!ENTITY epsi "[epsi]" >
<!ENTITY eta "[eta]" >
<!ENTITY gamma "[gamma]" >
<!ENTITY gt ">" >
<!ENTITY lambda "[lambda]" >
<!ENTITY larr "<--" >
<!ENTITY lt "<" >
<!ENTITY mu "[mu]" >

205
APPENDIX D. RETARGET SPECIFICATIONS USED FOR ENVISION

<!ENTITY ndash "--" >
<!ENTITY nq "\/" >
<!ENTITY omega "[omega]" >
<!ENTITY percnt "[percnt]" >
<!ENTITY phi "[phi]" >
<!ENTITY pi "[pi]" >
<!ENTITY psi "[psi]" >
<!ENTITY rarr "\rightarrow" >
<!ENTITY sigma "[sigma]" >
<!ENTITY tau "[tau]" >
<!ENTITY theta "[theta]" >
<!ENTITY xi "[xi]" >

<Document>
<ObjectID>{obid}</ObjectID>
<Article>
<MyBib><B.JRNART id="{obid}"/>

{for i:=1 to #author}
<Au><Anchor id="{author[i].obid}"/>
{ignore_line_breaks := TRUE}
<_fnm>{author[i].name.fname}</fnm>
{for j:=1 to author[i].name.#mname}
<Mnm>{author[i].name.mname[j]}</mnm>
{endfor}
{for j:=1 to author[i].name.#sname}
<Snm>{author[i].name.sname[j]}</snm>
{endfor}
</Anchor></Au>
{ignore_line_breaks := FALSE}
{endfor}
<Title>{title}</title>
<br><Anchor objectid="{published_in.obid}">{published_in.name}</Anchor></br>
<Date><Month>{date.month}</month><Year>{date.year}</year></Date>
<Vol>{volume}</Vol>
<Number>{number}</Number>
<Pp>{pp}</Pp>

{if #crcat > 0}
<Cat>
{for i:=1 to #crcat}
{ignore_line_breaks := TRUE}
<Crcat><Anchor objectid="{crcat[i].obid}">{crcat[i].term}</Anchor></Crcat>
{endfor}

206
APPENDIX D. RETARGET SPECIFICATIONS USED FOR ENVISION

</CAT>

{ignore_line_breaks := FALSE}
{endif}

{if #term > 0}
<TERMS>
{for i:=1 to #term}
{ignore_line_breaks := TRUE}
<TERM><ANCHOR objectid="{term[i].odbid}"{term[i].term}</ANCHOR></TERM>
{endfor}
</TERMS>

{ignore_line_breaks := FALSE}
{endif}

{if #keyword > 0}
<KEYWORDS>
{for i:=1 to #keyword}
{ignore_line_breaks := TRUE}
<KEYWORD>
<ANCHOR objectid="{keyword[i].odbid}"{keyword[i].term}</ANCHOR>
</KEYWORD>
{endfor}
</KEYWORDS>

{ignore_line_breaks := FALSE}
{endif}

{ignore_line_breaks := TRUE}
<ABSTRACT>{front_matter.abstract}</ABSTRACT>
</MYBIB>

{ignore_line_breaks := FALSE}
</FM>
<TITLE>{title}</TITLE>

{for i:=1 to #author}
<AU><ANCHOR objectid="{author[i].odbid}"{ignore_line_breaks := TRUE}
<FM>{author[i].name.fname}</FM>
{for j:=1 to author[i].name.#name}
<MNM>{author[i].name.mname[j]}</MNM>
{endfor}
{for j:=1 to author[i].name.#name}

207
APPENDIX D. RETARGET SPECIFICATIONS USED FOR ENVISION

<SNM>{author[i].name.sname[j]}</SNM>
{endfor}
</ANCHOR></AV>
{ignore_line_breaks := FALSE}

{endfor}
<Date><Month>{date.month}</Month><Year>{date.year}</Year></Date>

{ignore_line_breaks := TRUE}
<Abstract>{front_matter.abstract}</Abstract>

{if #rcat > 0}
<Cat>
{for i:=1 to #rcat}
{ignore_line_breaks := TRUE}
<CRCat><Anchor objectid="{rcat[i].odbid}">{rcat[i].term}</Anchor></CRCat>
{endfor}
</Cat>

{ignore_line_breaks := FALSE}
{endif}

{if #term > 0}
<Terms>
{for i:=1 to #term}
{ignore_line_breaks := TRUE}
<Term>{term[i].term}</Term>
{endfor}
</Terms>

{ignore_line_breaks := FALSE}
{endif}

{if #keyword > 0}
<Keywords>
{for i:=1 to #keyword}
{ignore_line_breaks := TRUE}
<Keyword>{keyword[i].term}</Keyword>
{endfor}
</Keywords>

{ignore_line_breaks := FALSE}
{endif}
APPENDIX D. RETARGET SPECIFICATIONS USED FOR ENVISION

{ignore_line_breaks := TRUE}
<BODY>
{call body( body )}

{for i:= 1 to #table}
{call table( table[i] )}
{endfor}

{for i:=1 to #figure_caption}
<P>{figure_caption[i].text}</P>
{endfor}
</BODY>

{if #bib = 1}
{call bib( bib )}
{endif}
</DOCUMENT>
{end document}

{begin body}
{process( section with section, subsection with subsection, paragraph with paragraph )
  ignore_text( sgml_start_tag, sgml_end_tag )}
{end body}

{begin section}
<SEC>
{process( subsection with subsection, title with title, paragraph with paragraph )
  ignore_text( sgml_start_tag, sgml_end_tag )}
</SEC>
{end section}
APPENDIX D. RETARGET SPECIFICATIONS USED FOR ENVISION

{begin subsection}
<SS>
{process( title with title, paragraph with paragraph )
    ignore_text( sgm1_start_tag, sgm1_end_tag )
}
</SS>
{end subsection}

{begin title}
<TITLE>
{process()
    ignore_text( sgm1_start_tag, sgm1_end_tag )
}
</TITLE>
{end title}

{begin bib}
<BIB>
{process( bib_entry with bib_entry, paragraph with paragraph )
    ignore_text( sgm1_start_tag, sgm1_end_tag )
}
</BIB>
</BM>
{end bib}

{begin bib_entry}
<B.OTHER id="{id}"/>
<P><ANCHOR objectid="{document.odbid}"
{process( sgm1_tag_b with copy, sgm1_tag_it with copy )
    ignore_text( sgm1_start_tag, sgm1_end_tag )
}
</ANCHOR></P>
</B.OTHER>
{end bib_entry}

{begin table}
<TABLE>
<TH>
{for i:=1 to #title}
{title[ii]}
{endfor}
</TH>
APPENDIX D. RETARGET SPECIFICATIONS USED FOR ENVISION

{for i:=1 to #header}
 {header[i]}
 {endfor}

{for i:=1 to #item}
 {item[i]}
 {endfor}

</TABLE>
 {end table}

{begin paragraph}
 <P>
 {process( bib_ref with bib_ref )
  ignore_text( sgml_start_tag, sgml_end_tag )
 }
 </P>
 {end paragraph}

{begin bib_ref}
 {if #bib_entry > 0}
   [<ANCHOR indocid="{bib_entry[i].id}">{bib_entry[i].label}</ANCHOR>
   {for i:=2 to #bib_entry}
   ,<ANCHOR indocid="{bib_entry[i].id}">{bib_entry[i].label}</ANCHOR>
   {endfor}
  ]
 {endif}
 {end bib_ref}

{begin copy}
 {text}
 {end copy}
Appendix E

ODB CLASSES DEFINED FOR ENVISION

We defined five classes of objects to be stored in the Envision permanent database. These are document, person, institution, publication, and indexing_term. In this Appendix we present the ODB definitions for those classes, as shown by the ODB interface.

E.1 Definition for the Document Class

Class ID: dc        Class Name: document       Alt. ID: 00001
Other Class Info:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th># Values</th>
<th>Internal ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>STRING</td>
<td>0..1</td>
<td>0</td>
</tr>
<tr>
<td>item_type</td>
<td>STRING</td>
<td>0..1</td>
<td>1</td>
</tr>
<tr>
<td>author</td>
<td>LINK</td>
<td>0..N</td>
<td>2</td>
</tr>
<tr>
<td>institution</td>
<td>STRING</td>
<td>0..N</td>
<td>3</td>
</tr>
<tr>
<td>published_in</td>
<td>LINK</td>
<td>0..N</td>
<td>4</td>
</tr>
<tr>
<td>edition</td>
<td>STRING</td>
<td>0..1</td>
<td>5</td>
</tr>
<tr>
<td>series</td>
<td>STRING</td>
<td>0..1</td>
<td>6</td>
</tr>
<tr>
<td>date</td>
<td>STRING</td>
<td>0..1</td>
<td>7</td>
</tr>
<tr>
<td>volume</td>
<td>STRING</td>
<td>0..1</td>
<td>8</td>
</tr>
<tr>
<td>number</td>
<td>STRING</td>
<td>0..1</td>
<td>9</td>
</tr>
<tr>
<td>chapter</td>
<td>STRING</td>
<td>0..1</td>
<td>10</td>
</tr>
<tr>
<td>pp</td>
<td>STRING</td>
<td>0..1</td>
<td>11</td>
</tr>
<tr>
<td>issn</td>
<td>STRING</td>
<td>0..1</td>
<td>12</td>
</tr>
<tr>
<td>isbn</td>
<td>STRING</td>
<td>0..1</td>
<td>13</td>
</tr>
<tr>
<td>coden</td>
<td>STRING</td>
<td>0..1</td>
<td>14</td>
</tr>
<tr>
<td>libcon</td>
<td>STRING</td>
<td>0..1</td>
<td>15</td>
</tr>
<tr>
<td>crt</td>
<td>STRING</td>
<td>0..1</td>
<td>16</td>
</tr>
<tr>
<td>crcat</td>
<td>LINK</td>
<td>0..N</td>
<td>17</td>
</tr>
<tr>
<td>term</td>
<td>LINK</td>
<td>0..N</td>
<td>18</td>
</tr>
<tr>
<td>keyword</td>
<td>LINK</td>
<td>0..N</td>
<td>19</td>
</tr>
<tr>
<td>bib_document</td>
<td>LINK</td>
<td>0..N</td>
<td>20</td>
</tr>
<tr>
<td>weights</td>
<td>STRING</td>
<td>0..N</td>
<td>21</td>
</tr>
<tr>
<td>_sgml_file</td>
<td>STRING</td>
<td>0..N</td>
<td>22</td>
</tr>
</tbody>
</table>
### APPENDIX E. ODB CLASSES DEFINED FOR ENVISION

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th># Values</th>
<th>Internal ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>_sgml_file_depto_d</td>
<td>STRING</td>
<td>0..N</td>
<td>23</td>
</tr>
<tr>
<td>_sgml_file_date</td>
<td>STRING</td>
<td>0..N</td>
<td>24</td>
</tr>
<tr>
<td>bib_entry</td>
<td>LINK</td>
<td>0..N</td>
<td>25</td>
</tr>
<tr>
<td>editor</td>
<td>LINK</td>
<td>0..N</td>
<td>26</td>
</tr>
<tr>
<td>publisher</td>
<td>STRING</td>
<td>0..1</td>
<td>27</td>
</tr>
<tr>
<td>abstract</td>
<td>STRING</td>
<td>0..1</td>
<td>28</td>
</tr>
<tr>
<td>cites</td>
<td>LINK</td>
<td>0..N</td>
<td>29</td>
</tr>
<tr>
<td>cited_in</td>
<td>LINK</td>
<td>0..N</td>
<td>30</td>
</tr>
</tbody>
</table>

---

#### E.2 Definition for the Person Class

**Class ID:** pr  
**Class Name:** person  
**Alt. ID:** 00002

**Other Class Info:**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th># Values</th>
<th>Internal ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>STRING</td>
<td>0..1</td>
<td>0</td>
</tr>
<tr>
<td>also_known_as</td>
<td>STRING</td>
<td>0..N</td>
<td>1</td>
</tr>
<tr>
<td>institution</td>
<td>LINK</td>
<td>0..N</td>
<td>2</td>
</tr>
<tr>
<td>address</td>
<td>STRING</td>
<td>0..N</td>
<td>3</td>
</tr>
<tr>
<td>phone</td>
<td>STRING</td>
<td>0..N</td>
<td>4</td>
</tr>
<tr>
<td>email</td>
<td>STRING</td>
<td>0..N</td>
<td>5</td>
</tr>
<tr>
<td>rsrch_interests</td>
<td>STRING</td>
<td>0..N</td>
<td>6</td>
</tr>
<tr>
<td>biography</td>
<td>STRING</td>
<td>0..N</td>
<td>7</td>
</tr>
<tr>
<td>author_of</td>
<td>LINK</td>
<td>0..N</td>
<td>8</td>
</tr>
<tr>
<td>current_as_of</td>
<td>DATE</td>
<td>0..1</td>
<td>9</td>
</tr>
<tr>
<td>editor_of</td>
<td>LINK</td>
<td>0..N</td>
<td>10</td>
</tr>
<tr>
<td>other_name</td>
<td>STRING</td>
<td>0..N</td>
<td>11</td>
</tr>
</tbody>
</table>

---

#### E.3 Definition for the Institution Class

**Class ID:** in  
**Class Name:** Institution  
**Alt. ID:** 00003

**Other Class Info:**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th># Values</th>
<th>Internal ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>STRING</td>
<td>0..1</td>
<td>0</td>
</tr>
<tr>
<td>also_known_as</td>
<td>STRING</td>
<td>0..N</td>
<td>1</td>
</tr>
<tr>
<td>type</td>
<td>STRING</td>
<td>0..1</td>
<td>2</td>
</tr>
<tr>
<td>address</td>
<td>STRING</td>
<td>0..1</td>
<td>3</td>
</tr>
<tr>
<td>phone</td>
<td>STRING</td>
<td>0..N</td>
<td>4</td>
</tr>
<tr>
<td>email</td>
<td>STRING</td>
<td>0..N</td>
<td>5</td>
</tr>
<tr>
<td>person</td>
<td>LINK</td>
<td>0..N</td>
<td>6</td>
</tr>
</tbody>
</table>
APPENDIX E. ODB CLASSES DEFINED FOR ENVISION

document       LINK       0..N       7
publication    LINK       0..N       8
current_as_of  STRING     0..1       9
other_name     STRING     0..N       10

E.4 Definition for the Publication Class

Class ID: pb    Class Name: Publication    Alt. ID: 00004
Other Class Info:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th># Values</th>
<th>Internal ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>STRING</td>
<td>0..1</td>
<td>0</td>
</tr>
<tr>
<td>also_known_as</td>
<td>STRING</td>
<td>0..N</td>
<td>1</td>
</tr>
<tr>
<td>publisher</td>
<td>LINK</td>
<td>0..1</td>
<td>2</td>
</tr>
<tr>
<td>frequency</td>
<td>STRING</td>
<td>0..1</td>
<td>3</td>
</tr>
<tr>
<td>document</td>
<td>LINK</td>
<td>0..N</td>
<td>4</td>
</tr>
<tr>
<td>other_name</td>
<td>STRING</td>
<td>0..N</td>
<td>5</td>
</tr>
</tbody>
</table>

E.5 Definition for the Indexing_term Class

Class ID: it    Class Name: indexing_term    Alt. ID: 00005
Other Class Info:

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
<th># Values</th>
<th>Internal ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>term</td>
<td>STRING</td>
<td>0..1</td>
<td>0</td>
</tr>
<tr>
<td>document</td>
<td>LINK</td>
<td>0..N</td>
<td>1</td>
</tr>
</tbody>
</table>
Appendix F
DELTO GRAMMAR

<frmt_description> ::= [USE : literal {, literal};]
[LEX : literal ;]
[UNDEFINE : <class_name> {, <class_name> } ] ;
[CONSTANTS: const_ident { integer | literal}
{, const_ident { integer | literal} } ;]
[<class_def> {; <class_def> }]
ENDFMTD

<class_name> ::= ident

<class_def> ::= CLASS ( DEFINITION | REDEFINITION ) : <class_name>
[PERSISTENT INSTANCES]
[UNDEFINE <label> {, <label> }]
[<attrs_block>]
[<components_block>]
[<odb_interface_block>]
[<description_block>]
{ <string_comp_descr> }
ENDDEF

<label> ::= ident
APPENDIX F. DELTO GRAMMAR

\[
\begin{align*}
\langle\text{attrs\_block}\rangle & \ ::= \ \text{ATTRIBUTES} \\
& \quad \{ \% \\
& \quad \quad \langle\text{attrib\_def}\rangle \ {;\ }\langle\text{attrib\_def}\rangle \} \\
& \quad \} \\
\langle\text{attrib\_def}\rangle & \ ::= \ \langle\text{attrib\_name}\rangle \ : \ \langle\text{attrib\_type}\rangle \\
\langle\text{attrib\_name}\rangle & \ ::= \ \text{ident} \\
\langle\text{attrib\_type}\rangle & \ ::= \ \text{NUMERIC} \ | \ \text{STRING} \ | \ \text{REFID} \\
\langle\text{components\_block}\rangle & \ ::= \ \text{COMPONENTS} \\
& \quad \{ \%
\quad \quad \langle\text{component\_def}\rangle \ {;\ }\langle\text{component\_def}\rangle \} \\
& \quad \} \\
\langle\text{component\_def}\rangle & \ ::= \ \langle\text{component\_name}\rangle \\
& \quad \quad \ : \ \langle\text{component\_type}\rangle \ [ \langle\text{range}\rangle \] \\
\langle\text{component\_name}\rangle & \ ::= \ \text{ident} \\
\langle\text{component\_type}\rangle & \ ::= \ \langle\text{class\_name}\rangle \ | \ \text{NUMERIC} \ | \ \text{STRING} \\
\langle\text{range}\rangle & \ ::= \ \text{integer} \ '...' \ \text{integer} \ |
\quad \quad \text{integer} \ '...' \ \mathbb{N}
\end{align*}
\]
APPENDIX F. DELTO GRAMMAR

<description_block> ::= DESCRIPTION
                     '{',
                     { <frequency_block> }
                     '}',

<frequency_block> ::= ( ALWAYS | FREQUENTLY | SOMETIMES | NEVER )
                     '{',
                     <strength_block>
                     '}',

<strength_block> ::= [ <toplv_exp> [ <strength_indic> ]
                     { ; <toplv_exp> [ <strength_indic> ]] ]

<strength_indic> ::= ( VERY-STRONG | STRONG | MODERATE | WEAK |
                     VERY-WEAK )
                    INDICATOR

<toplv_exp> ::= [ <label> : ] <expression>
               [ ']' <attrib_bind> { , <attrib_bind> } ']' ]

<attrib_bind> ::= <attribute> ' := '
                      ( <attribute> | <component> | <function> |
                        integer | literal )

<expression> ::= <predicate> { <log_op> <predicate> }

217
APPENDIX F. DELTO GRAMMAR

<predicate> ::= <argument> [ <rel_op> <argument> ] |
               <predic_name> ( ( [ <arg_list> ] ) ) |
               NOT <predicate> |
               ( ' <expression> ' )

<arg_list> ::= <argument> {, <argument> }

<argument> ::= <attribute> | <component> | <function> |
              <regular_exp> | integer | literal

<log_op> ::= OR | AND

<predic_name> ::= ident

<rel_op> ::= < | <= | = | <> | >= | >

<attribute> ::= [[ ( !. <class_name> [ <selector> ] ) | ( @ integer ) ) ]
               <attrib_name>

<component> ::= [[ ( !. <class_name> [ <selector> ] ) |
                  ( @ integer ) [.] ]]
               [ <component_name> [ <selector> ]
               [ , <component_name> [ <selector> ]]
               [ . <attribute> ]]

218
APPENDIX F. DELTO GRAMMAR

\[ <\text{selector}> ::= '[][ <\text{index\_binding}> ] <\text{integer}> ' | '[][ <\text{index\_binding}> ] <\text{sel\_keyword}> ' | '[][ <\text{index\_binding}> ] <\text{expression}> ' | '[][ <\text{index\_var}> ' | '[]'
\]

\[ <\text{index\_binding}> ::= <\text{index\_var}> .=.
\]

\[ <\text{index\_var}> ::= \text{ident}
\]

\[ <\text{sel\_keyword}> ::= \text{EACH} | \text{ALL} | \text{ANY} | \text{LAST} | \text{NEW}
\]

\[ <\text{function}> ::= ( <\text{funct\_name} > '(', [ <\text{funct\_arg} > {, <\text{funct\_arg} > } ] ')') | ( <\text{funct\_arg} > '+' <\text{funct\_arg} > )
\]

\[ <\text{funct\_arg} > ::= <\text{attribute}> | <\text{component}> | <\text{function}> | <\text{literal}> | <\text{integer}>
\]

\[ <\text{funct\_name}> ::= \text{ident}
\]
APPENDIX F. DELTO GRAMMAR

<regular_exp> ::= <regular_oprnd> { <regular_oprnd> } 
                { '?' <regular_oprnd> { <regular_oprnd> } }

<regular_oprnd> ::= <reg_exp_unit> [ <occur_indic> ] | 
                 <reg_exp_symbol> | 
                 [ <component> := ] <wildcard> 
                 [ ':' [ '{' { <class_name> } '}' ] ] 
                 [ <occur_indic> ] | 
                 literal [ <occur_indic> ] | 
                 similar_to (' literal integer ') 
                 [ <occur_indic> ] | 
                 '(' <regular_expr> ')' [ <occur_indic> ]

<reg_exp_unit> ::= '<?' <class_name> {, <expression> } '>'
                 [ '[' <comp_binding> {, <comp_binding> } ']' ]

<reg_exp_symbol> ::= BOUNDARY | OBJECT | DOC_BEG | DOC_END

<wildcard> ::= ANYTHING

<occur_indic> ::= '?' | '*' | '+' | <range>
<str_comp_desc> ::= STRING COMPONENT DESCRIPTION FOR ':
   <component> {, <component> }
   '{
   <frequency_block> { <frequency_block> }
   '}

<odb_interface> ::= ODB-INTERFACE
   '{
   [ <access_keys_block> ]
   [ <match_desc_block> ]
   [ <actions_block> ]
   '}

<access_keys_block> ::= ACCESS KEYS
   '{
   <key> {; <key> }
   '}

<key> ::= ( <component> | <attribute> | <string_funct> )
   [ <strength_indic> ]

<match_desc_block> ::= MATCH-DESC
   '{
   { <frequency_block> }
   '}

221
APPENDIX F. DELEO GRAMMAR

<action_block> ::= '{

    { <odb_action> | <if> }

'}

<odb_action> ::= <odb_action_name>

    '(', <odb_action_arg> {, <odb_action_arg>} ')'

<odb_action_arg> ::= <attribute> | <component> | <function> |

    literal | integer

<odb_action_name> ::= ODB_ADD_VALUE | ODB_UPDATE_VALUE | ODB_ADD_LINK |

    ODB_INDEX_ON

<if> ::= IF '(' <expression> ')' <action_block>
Appendix G

RETARGET GRAMMAR

<translation_spec> ::= { <template> }

<template> ::= '{' BEGIN <template_name> '}'
            { literal_text | <statement> }
            '{' END [ <template_name> ] '}'

<template_name> ::= ident

<statement> ::= <component> | <process_st> | <call_st> | <if_st> | <for_st> | <assignment_st>

<component> ::= '{' <component_name> [ <selector> ]
              { . <component_name> [ <selector> ] } '}'

<component_name> ::= [#] ident

<selector> ::= '[' ( integer | <index_var> ) ']

<index_var> ::= ident
APPENDIX G. RETARGET GRAMMAR

<process_st> ::= '{
PROCESS
'(' [ <template_sel> {, <template_sel> }] ')'
[IGNORE_TEXT
'(' '<class_name> {, '<class_name>' } ')'
'}'

$template_selc ::= '<class_name> [<rel_op> integer] WITH <template_name>

<call_st> ::= '{CALL <template_name> '(' '<component>' ')'

<if_st> ::= '{IF '<component> log_op integer'}'

<log_op> ::= < | <= | = | <> | >= | >

<for_st> ::= '{FOR <index_var> := integer TO '<component>' '}

<assign_st> ::= '{' '{<assignment>[;]}' '}

<assignment> ::= '<global_var> := ( integer | <boolean> )
APPENDIX G. RETARGET GRAMMAR

<global_var> ::= OUTPUT_WIDTH | IGNORE_LINE_BREAKS | IGNORE_SPACING |
                  DEFAULT_SPACING | IGNORE_SGML_INPUT

<boolean> ::= TRUE | FALSE
VITA

I am who I am, and that’s it. BTW, I was born and grew up in the most beautiful city of the entire world: Buenos Aires, Argentina, where I attended the “Escuela Superior de Comercio Carlos Pellegrini” and then the “Universidad de Buenos Aires”.

[Signature]

226