

# Principles of Biomedical Informatics

Ira J. Kalet, Ph.D.

University of Washington  
Seattle, Washington

Version of February 24, 2007

**DRAFT ONLY: NOT FOR DISTRIBUTION**

© 2006 by Ira J. Kalet

All rights reserved. No part of this manuscript may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the copyright holder.

# Contents

<b>Preface</b>	<b>ix</b>
<b>I Foundations of Biomedical Informatics</b>	<b>1</b>
<b>1 Biomedical Computing With Symbols and Lists</b>	<b>3</b>
1.1 What can be expressed symbolically? . . . . .	4
1.1.1 Biomedical examples . . . . .	6
1.1.2 Natural Language in Symbols and Lists . . . . .	7
1.2 The interactive Lisp environment . . . . .	9
1.2.1 LISP data types, values and expressions . . . . .	10
1.2.2 Evaluation of Arguments, Special Forms and Macros . . . . .	12
1.2.3 Side Effects . . . . .	13
1.3 Some Essential Programming Ideas . . . . .	13
1.3.1 Creating new functions from existing ones . . . . .	13
1.3.2 Formal Parameters and Local Bindings . . . . .	15
1.3.3 Predicates, truth and falsehood . . . . .	16
1.3.4 Copies and Identity . . . . .	16
1.3.5 Testing for equality . . . . .	17
1.3.6 Keywords . . . . .	18
1.3.7 Logic functions . . . . .	18
1.3.8 Conditional expressions . . . . .	18
1.4 Advanced topics . . . . .	19
1.4.1 Anonymous functions . . . . .	19
1.4.2 Functions can be used as data . . . . .	20
1.4.3 Declarations and Optimization . . . . .	20
1.4.4 Macros - Code That Writes Code . . . . .	20
1.4.5 Special Features of Lisp . . . . .	20
1.5 Object-oriented Programming in Lisp - CLOS . . . . .	21
1.5.1 Types, Structures and Classes . . . . .	21
1.5.2 Generic Functions and Methods . . . . .	22
1.6 More Notes . . . . .	24
1.7 The Meta-Object Protocol (MOP) . . . . .	24
1.8 Lisp Promo . . . . .	26

1.9	Some Programming Language History . . . . .	27
<b>2</b>	<b>Biomedical Data</b>	<b>31</b>
2.1	Data Description and Metadata . . . . .	34
2.1.1	Metadata . . . . .	35
2.1.2	Biomedical Encoding Examples . . . . .	37
2.2	Patient Data: An Illustration . . . . .	48
2.2.1	A Simple Solution Using Tags (Keywords) . . . . .	49
2.2.2	An Object-Oriented Design With Classes . . . . .	51
2.2.3	A Better Solution Using Metaobjects . . . . .	57
2.2.4	Medical Images: Incorporating Binary Data . . . . .	64
2.3	Database Systems and Ideas . . . . .	72
2.3.1	The Relational Model . . . . .	73
2.3.2	A Brief Introduction to SQL . . . . .	74
2.3.3	Constraints and Keys . . . . .	76
2.3.4	Relational Algebra . . . . .	77
2.3.5	Relational Calculus . . . . .	77
2.3.6	More on the Relational Model . . . . .	77
2.3.7	Other Data Models . . . . .	78
2.4	Data Quality . . . . .	79
2.5	Decoding and Parsing of Data Streams . . . . .	80
2.6	Data Interchange Standards . . . . .	81
2.6.1	HL7 . . . . .	82
2.6.2	DICOM . . . . .	82
2.6.3	XML, RDF and OWL . . . . .	82
2.7	Data, Information and Knowledge . . . . .	84
2.8	Summary and Further Reading . . . . .	86
<b>3</b>	<b>Symbolic Biomedical Knowledge</b>	<b>89</b>
3.1	Biomedical Theories and Computer Programs . . . . .	90
3.1.1	A World Class Reasoning Example . . . . .	91
3.1.2	Biological Examples . . . . .	92
3.1.3	Symbolic Theories . . . . .	94
3.2	Logic and Inference Systems . . . . .	96
3.2.1	First order logic . . . . .	101
3.2.2	Rule-based Programming and Proof Strategies . . . . .	104
3.2.3	Limitations of FOL . . . . .	105
3.3	Frames, Semantic Nets and Ontologies . . . . .	106
3.3.1	A Simple Frame System . . . . .	107
3.3.2	Frame System Implementations and Standards . . . . .	123
3.3.3	Frames and Object-Oriented Programming Languages . . . . .	123
3.3.4	Frames and the Semantic Web . . . . .	123
3.4	Description Logics . . . . .	123
3.5	Search . . . . .	124
3.6	Classification Trees . . . . .	128
3.7	Modeling Time . . . . .	128

3.8	Simulation Models . . . . .	129
3.8.1	Petri Nets . . . . .	129
3.8.2	Modeling of Organizations, Work Flow, and Scheduling . . .	129
3.9	Summary and Further Reading . . . . .	129
<b>4</b>	<b>Probabilistic Biomedical Knowledge</b>	<b>131</b>
4.1	Basic Probability and Statistics . . . . .	132
4.1.1	The Laws of Probability . . . . .	132
4.1.2	Bayes' Theorem and Applications . . . . .	135
4.2	Probabilistic Modeling Methods . . . . .	143
4.2.1	Bayes Nets and Influence Diagrams . . . . .	143
4.2.2	Probabilistic Decision Trees . . . . .	143
4.2.3	Multiattribute Decision Modeling . . . . .	144
4.2.4	Statistics and Statistical Modeling . . . . .	145
4.3	Stochastic Models . . . . .	145
4.4	Hybrid Models . . . . .	145
4.5	Information Theory . . . . .	146
<b>5</b>	<b>Biomedical Information Access</b>	<b>147</b>
5.1	Natural Language Processing . . . . .	147
5.1.1	Principles . . . . .	147
5.1.2	Applications . . . . .	147
5.2	Information Retrieval Systems . . . . .	148
5.3	IR System Design . . . . .	148
5.3.1	Indexing . . . . .	148
5.3.2	Processing Queries . . . . .	148
5.3.3	Searching and Matching . . . . .	148
5.3.4	Ranking the Results . . . . .	148
5.4	Content-based Image Retrieval . . . . .	148
5.5	Electronic Medical Records . . . . .	148
5.6	Querying Knowledge Bases . . . . .	149
<b>6</b>	<b>Information Use in Biomedical Contexts</b>	<b>151</b>
6.1	The Social and Political Context of Biomedical Computing . . . . .	151
<b>II</b>	<b>Biomedical Ideas and Computational Realizations</b>	<b>153</b>
<b>7</b>	<b>Classifying and Organizing Genes and Proteins</b>	<b>155</b>
7.1	Some Essential Molecular Biology . . . . .	155
7.2	The Gene Ontology . . . . .	156
7.2.1	Representations of GO . . . . .	156
7.3	Gene Expression . . . . .	158

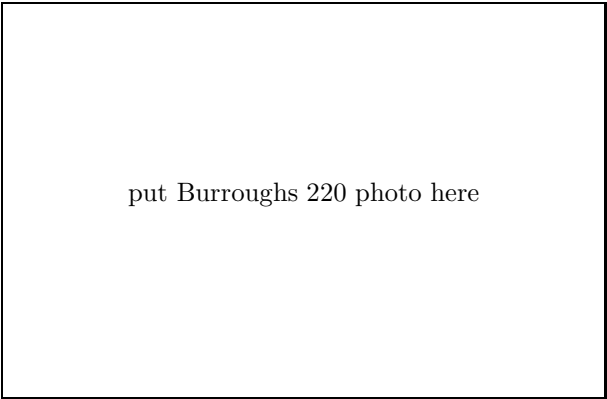
<b>8</b>	<b>Biological Networks</b>	<b>159</b>
8.1	EcoCyc - Using a Metabolic Pathway Model . . . . .	159
8.2	PathMiner - Computing Pathways From Reactions . . . . .	162
8.3	Petri Net Models . . . . .	162
<b>9</b>	<b>Modeling Biological Structure</b>	<b>163</b>
9.1	The UW Foundational Model of Anatomy . . . . .	163
9.1.1	The Components of the FMA . . . . .	163
9.1.2	FMA Projects . . . . .	163
9.1.3	A Simple Network Interface - the FMS . . . . .	164
9.2	Galen, Others . . . . .	169
9.3	Ontologies for Molecular Biology . . . . .	169
9.4	Protein Structure . . . . .	169
<b>10</b>	<b>Drug Interactions</b>	<b>171</b>
10.1	Background on Drug Interactions . . . . .	171
10.2	A Catalog of Drug Interactions . . . . .	172
10.3	Reasoning About Pharmacokinetics . . . . .	173
10.3.1	Using Knowledge About Enzymatic Metabolism . . . . .	175
10.3.2	A Rule-based Drug Interaction System . . . . .	177
10.3.3	Representing Levels of Evidence . . . . .	179
10.4	Reasoning About Pharmacodynamics . . . . .	179
10.5	Extensions . . . . .	182
<b>11</b>	<b>Using Medical Guidelines</b>	<b>185</b>
11.1	GLEE . . . . .	185
11.2	Other . . . . .	186
<b>12</b>	<b>A Nursing Expert System</b>	<b>187</b>
<b>13</b>	<b>DICOM: Medical Image Information Agents</b>	<b>189</b>
13.1	A Short History of DICOM . . . . .	189
13.2	Networks and Protocols . . . . .	190
13.3	About the DICOM Standard . . . . .	194
13.4	How to Implement DICOM . . . . .	200
13.4.1	The DICOM State Machine . . . . .	200
13.4.2	Action Functions . . . . .	206
13.4.3	Parsing and Generation of PDUs . . . . .	213
13.4.4	Parsing and Generation of Commands . . . . .	219
13.4.5	Parsing and Generation of Data . . . . .	221
13.5	Image Server Design . . . . .	224
13.5.1	Image Data Types Supported . . . . .	225
13.5.2	Storage of the Received Data . . . . .	225
13.6	DICOM Client Design . . . . .	226
13.7	Other . . . . .	227
13.7.1	What We Discovered About DICOM Itself . . . . .	227

13.7.2 Consistency With Other Implementations . . . . .	228
<b>14 Cancer Radiotherapy Planning</b>	<b>231</b>
14.1 Radiation Therapy . . . . .	232
14.2 Radiotherapy Planning Software . . . . .	234
14.3 Locating the Target . . . . .	237
14.3.1 Classification of Tumor Sites . . . . .	239
14.3.2 Computing the CTV . . . . .	242
14.3.3 Computing the PTV . . . . .	246
14.4 Influence Diagrams in RTP . . . . .	250
14.5 Automated Radiation Beam Placement . . . . .	250
<b>15 Public Health Applications</b>	<b>251</b>
<b>16 Intelligent Query Processing</b>	<b>253</b>
<b>III Engineering Practical Systems</b>	<b>255</b>
<b>17 Biomedical Data Integration</b>	<b>257</b>
17.1 Biowarehouse . . . . .	257
17.2 Biomediator . . . . .	257
<b>18 Building Safe Biomedical Software</b>	<b>259</b>
18.1 An Example: A Simple Radiotherapy Machine . . . . .	260
18.2 automated analysis . . . . .	263
<b>19 Using the World Wide Web</b>	<b>265</b>
19.1 Computed URLs: CL-HTTP . . . . .	265
19.2 Common Lisp as a CGI Language . . . . .	265
19.3 Other . . . . .	266
<b>A Software Resources and Current Research</b>	<b>267</b>
A.1 Programming Resources for the Reader . . . . .	267
A.2 Some Research Projects Using Lisp and Prolog . . . . .	268





# Preface



put Burroughs 220 photo here

One of the great challenges of biomedical informatics is to discover and/or invent ways to express biomedical data and knowledge in computable form, and to thus be able to automate the analysis and interpretation of a wide range of biomedical data, as well as facilitate access to biomedical data and knowledge. Medical informatics textbooks [122, 135] typically present a survey of such ideas and applications, but do not discuss how these ideas and applications are actually realized in computer software. Bioinformatics books typically have either narrowly focused on sequence analysis algorithms and other statistical applications, or on how to use a wide variety of existing software tools. Some of the bioinformatics books on closer inspection are really about the underlying biology with little or no reference to computerized data or knowledge modeling except as a means to arrive at biological results, reported as text and/or diagrams, rather than in computable form.

A recurring theme in discussions with biologists, medical researchers, health care providers, and computer scientists is the question, “What is biomedical informatics?” The practical answer, that we are going to provide the software tools to meet biomedical research and clinical practice computing needs, seems inadequate to define the intellectual content of the field. Thus, the main goal of this book is to provide an exposition of important fundamental ideas and methods in biomedical informatics, using actual working code that can be read and understood. The

object of the code is not only to provide software tools, but for the code to play the same formal role that equations and formulas play in theoretical physics. The application of the programs to specific problem instances is analogous to the solution of the equations of physics for specific practical problems, e.g., the design of airplane wings.

This is not a book about Artificial Intelligence, or at least it does not take the viewpoint typical of AI books. My viewpoint is that of a scientist interested in the creation and study of formal theories in biology and medicine. Whether such theories embody some notion of intelligence is not relevant here. Some of the topics covered strongly resemble those of an AI textbook, because the ideas and languages from that branch of computer science are the right ones for modeling a wide range of biological and medical phenomena, just as differential equations, vector spaces, abstract algebra and related areas of mathematics are the right ideas and languages for creating theories of physical phenomena.

While the predominant view in this book is that of computer programs as implementations of formal theories, it is not the only way that computable knowledge in biomedicine is used. In regard to formal theories, biology and medicine are relatively young (compared to the physical sciences, particularly physics). Much knowledge is in small chunks like jigsaw puzzle pieces. Independently of whether you believe the pieces can fit together to form a coherent theory, many people find it useful to organize, index and search collections of such chunks to solve important practical problems. This, too is an important part of biomedical informatics. Thus, Chapter 3 includes some discussion of knowledge as a collection of assertions that may or may not be a complete theory of a biomedical domain. It is developed more fully in Chapter 5, on biomedical information access, as well as in some example chapters in Part II.

There are precedents for using computer programs as renditions of subject matter, not just examples of programming techniques or programming language syntax and semantics. They are an inspiration for this work. Norvig's book, *Paradigms of Artificial Intelligence Programming* [94], explains fundamental ideas of artificial intelligence by showing how to build the programs that implement these ideas. Abelson and Sussman's well known *Structure and Interpretation of Computer Programs* [1] is really about computation ideas and principles, and not about how to do programming in Scheme. More recently, Sussman and Wisdom have written a remarkable new treatment of classical mechanics [132], not from the point of view of traditional numerical approximation of solutions of differential equations, but from the point of view of expressing the foundational concepts themselves directly in a computable form. My goal similarly is to write about biomedical informatics ideas by showing how to write programs that implement them.

## Structure of the Book

The chapters in Part I, Foundations of Biomedical Informatics, develop the principles, ideas and methods that constitute the academic discipline of Biomedical Informatics. These can be organized loosely into the areas of biomedical data and

knowledge representation, biomedical information access, and the use of biomedical data and information in context. Biomedical knowledge representation, as inherited or adapted from the field of Artificial Intelligence, can be loosely divided into categorical or symbolic methods and ideas, and probabilistic methods and ideas. Thus, the book begins with representation and transformation of biomedical data, then symbolic and probabilistic biomedical knowledge. Biomedical information access addresses both data and knowledge. This part concludes with some discussion of the context for creating and using biomedical data, knowledge and software.

\*\*\* expand the above paragraph?

Part II, Biomedical Ideas and Computational Realizations, provides a series of applications of the principles, illustrating how to express biomedical ideas, data, and knowledge easily in computable form. The chapters include examples from the three main application domains of biology, clinical practice (broadly conceived, including medicine, nursing and pharmacy) and public health, in roughly that order.

The Appendix provides reference material on the programming languages and environments used in the book. It also includes pointers to resources for pursuing some of the engineering aspects of building a practical system in more depth.

\*\*\* Part III? maybe we do need this - "Building Practical Systems

\*\*\* topics currently not given much coverage, perhaps should be amplified somewhere: maybe a chapter on security/privacy issues and de-identification strategies (hmm, EpiQMS?), also see next section - maybe a whole chapter on design of software.

\*\*\* topics deliberately not covered, and why: mainstream bioinformatics and computational biology that is covered in other books, e.g., sequence alignment, motifs, clustering, PCA, phylogenetic trees, etc.

\*\*\* what about data mining more generally, or machine learning?

## Software Design and Engineering

There is a crisis in biomedical software today. The NIH has recognized the importance of computing in biomedical research through the Biomedical Information Science and Technology Initiative (BISTI). The cost of building custom (one of a kind) programs of any scale, for each research team, is usually prohibitive. Yet, commercial software packages are often not adaptable to the specific needs of research labs, let alone the complexities of clinical and public health practice. It is certainly not simply a matter of taking advantage of existing technology that has been produced by computer scientists. The problems of data integration, design of extensible and flexible software systems and programming environments, matching

different data models and modeling systems, are all important research questions in computer and information science. The emerging partnership that BISTI promises to promote is critical for the future of biomedicine.

A major issue in biomedical informatics is that of code reuse and software engineering. While the enthusiasm for “open source” software development is growing rapidly, there is little discussion of best practices in software design, and the result is a large body of freely available code that is notoriously difficult to extend or modify. Many advocates of “open source” seem to believe that by itself, this approach will produce (by some evolutionary process) high quality software. This is insufficient for the crisis we face today, especially in safety critical applications such as clinical medicine. The examples I develop in Part II are intended to be exemplars of good software design as well as clear illustrations of the underlying principles.

The challenge is even greater in health care than in biomedical research. Software inadequacies, errors and failures have been shown to be directly or indirectly responsible for serious injury as well as death to patients [82]. Regulatory agencies such as the FDA face a daunting job to deal with this on the scale of very complex electronic medical record systems with huge numbers of hardware and software components. The use of electronic data resources and decision support to screen for drug interactions is now the standard of practice for many pharmacies, from small neighborhood operations to those operated by major health care providers. Unfortunately these systems fall far short of reasonable expectations, so much so that their results are often disregarded or overridden [48].

\*\*\* more to say about this?? comment on the relevance of software engineering methods and tools (or non-relevance in some cases)

\*\*\* this should probably be a chapter all by itself, to include why it is important to have a complete, consistent and unambiguous spec, how to verify, what is the difference between verification and validation, testing and its role, user centered design, what is robust and what is unreasonable - some of this will be in the Software Safety chapter, and some in the DICOM chapter, and can be referenced

## Programming Languages

The main programming language that is used in this book is Common Lisp. To help the reader who may be unfamiliar with Lisp, Chapter 1 provides a introduction to the essential ideas.

From the very beginning of informatics research in medicine, Lisp has been a prominent choice of programming language. In medicine and biology, much of the knowledge (as well as data) is non-numeric. Lisp is particularly well suited to this. However, Lisp is also fine for numeric computation, user interface programming, web applications and many other things as well. So, the importance of Lisp in biomedical informatics should not be underestimated by thinking of it as only “the language people used to use for Artificial Intelligence research.” Its power comes from the

also cite the EMR articles on deaths due to medical errors and the introduction of EMR systems

ability to easily create abstraction layers and programs that transform expressions before evaluating them. This makes it easy to define higher level languages in Lisp, that provide just the right tools for efficient problem solving. Also in Lisp, functions can be manipulated as data: new functions can be written by other functions, and then used. While this may seem mysterious and useless to the programmer new to Lisp, once you see how it works, you will wonder how people could write programs without it (cf. Graham [41, Chapter 1]).

Lisp is a fine general purpose programming language. It has been shown to be easier to use than C or java, and compilers are available to generate code that runs as fast or faster than equivalent code written in C. It certainly supports higher performance than java. A recent study by Ron Garrett (formerly Erann Gat) [37] demonstrated these and other important points.

\*\*\* develop this list and maybe move to Chapter 1, with forward pointer here

Other important things to know about Lisp:

1. Fewer lines of code than other languages to solve similar problems, which also means less development time, fewer programmers, lower cost, less bugs, and less management overhead,
2. Easy to debug, lots of built in tools and facilities, ease of finding errors, because of structure,
3. Lots of libraries and freely available code base, lots of support, and yes, there are Lisp programmers,
4. Myths about Lisp are untrue (list here),
5. and most important, Lisp provides a superb interactive programming and execution environment.

A goal of this book is to refocus thinking about biomedical informatics programming, not as a technical skill for manipulating bits and bytes, but as a natural, logical, and accessible language for expressing biomedical knowledge, much as mathematics is used as the universal language for expressing knowledge in the physical sciences. Lisp is used in this work, because it was designed for building these abstractions, higher level languages, and extensions. As bioinformatics in particular moves from the data-intensive stage of searching and correlating massive amounts of raw data to the knowledge-intensive stage of organizing, representing, and correlating massive numbers of biological concepts, the expressive power and efficacy of Lisp will become even more important.

As biomedical data and knowledge become increasingly complex, and the software to manage it grows in scale, it is almost a truism to emphasize how important it is to effectively use abstraction and layered design. Lisp is also important because it has proven to be the best suited to these goals. I chose Common Lisp (one of the two major contemporary dialects of Lisp) because it has all the technical support to write “industrial strength” programs that perform as well or better than

programs written in more popular programming languages such as `C++`, `java` and `perl`. Common Lisp has a very solid and powerful object oriented programming facility, the Common Lisp Object System (CLOS). This facility provides a flexible means to write extensible code; indeed, CLOS is built on its own ideas, and includes a meta-object protocol (MOP), which provides ways to extend and augment CLOS itself without needing or modifying any CLOS source code. Common Lisp also provides a macro facility that makes it easy to write code that transforms expressions, i.e., it supports writing interpreters for higher level (but embedded) languages for special purpose tasks.

Use of Lisp for solving problems in biomedical informatics has experienced slow but steady growth in recent times. There are many web sites and much code to perform specific tasks, but it remains daunting to the new informatics researcher and programmer to become familiar with the advanced programming ideas and the skills needed to utilize existing facilities and contribute to this field. Too often the well trodden but less powerful path of programming in the conventional languages and styles is adopted and taught. This is done despite the availability of several good introductions and working environments. Fortunately several good books are available. They are described in Appendix A.1.

Some examples also use the Prolog programming language. Prolog is a programming language designed to represent logic systems, principally first order logic restricted to Horn clauses. Simple Prolog-like interpreters have been written in Lisp, and indeed Lisp has also been used for logic programming and constraint programming, but the availability of excellent (and free) Prolog systems also makes it attractive to use Prolog directly. Prolog is important for the same reasons as Lisp, because it provides a more abstract symbolic language in which to directly express biologic concepts, rather than encoding them in more machine-like data structures.

Many people have asked, “Why use Lisp and Prolog; why not use Java or Perl (or Python or Ruby, etc.)?” The answer above should be sufficient, that the best language to represent abstract ideas is one that supports abstract symbols and expressions (lists) directly. It is possible to mimic some of this in Java, Perl, etc., but it is makeshift, at best. I would ask (and have asked) such a questioner the opposite question, “Why would anyone use Java or Perl, when Lisp seems more appropriate?” The responses always include exactly two reasons: first, everyone else is doing it, and second, there is a lot of support for those languages. This is not sufficient justification to make a technically inferior choice, especially in view of the fact that there are plenty of choices of robust Lisp and Prolog implementations (both free and commercial), and a critical mass of users of both.

Significant bioinformatics work is done with Lisp and Prolog. Appendix A.2 provides a brief summary of some of the more noteworthy projects with pointers to publications and web sites.

## Biology, Medicine and Public Health

What should the reader know about biology, medicine and population health?

Background on biology - references and readings for molecular biology, genetics,

anatomy, physiology, general medicine (ICM). The reader should be familiar with basic concepts of modern biology (genes, proteins, DNA, basic functions of living organisms), and have some experience with disease, treatment and the health care system.

\*\*\* background in math?

## Notes on Typography

In this book, code and data are distinguished from the rest of the text by using a **different font**. Blocks of code that define functions, classes, etc. are shown without a prompt. When the intent is to show a dialog, i.e., to illustrate what happens at the interactive **read-eval-print** loop, a prompt is shown. Different Common Lisp implementations each use their own style of prompt. Here, we follow the same convention as in Graham [41], the “greater-than” angle bracket, `>`.

## How To Use This Book

My main objective in writing the book is to provide a principled exposition of the foundational ideas of biomedical informatics. It provides a solid technical introduction to the field, and a way to gain deeper insight into familiar biomedical computing ideas and problems. I hope that experienced researchers and practitioners in biomedical and health informatics will find it a useful synthesis of our field.

The intended audience also includes:

- biologists, clinicians and others in health care, who have little previous exposure to programming or a technical approach to biomedical informatics, for whom Chapter 1 is provided as an introduction to symbolic computing and programming,
- computer scientists, mathematicians, engineers, and others with technical backgrounds, who have an interest in biomedical informatics but not much exposure to biology or medicine, for who some basic ideas in molecular and cellular biology, human anatomy, pharmacology, clinical medicine and other biomedical areas are introduced where needed, and
- students interested in biomedical informatics as a field of study and potential career.

The book can thus serve as a text for a course in several contexts:

\*\*\* fill in here example courses

## Acknowledgements

My interest in computing started during my freshman year in college, 1961, when I found that the Cornell University computer center provided free student access to its Burroughs 220 computer (the picture shows me in 1962 at the operator's console). I taught myself Burroughs 220 assembly language and began my first programming project. The goal of this project was to replicate some experiments in automatically composing music according to rules of harmony, voice leading, and also some more modern composition techniques. The experiments I wanted to replicate and build on were done by Lejaren Hiller and Leonard Isaacson [51], and resulted in a composition, the "Illiac Suite for String Quartet," named for the computer at the University of Illinois that they used. I soon realized that neither assembly language nor languages like ALGOL and FORTRAN were suitable for this task. I did not know at the time that a very powerful programming language called Lisp had just been invented to express exactly these kinds of symbolic, abstract ideas. I also did not know that this kind of programming was called "artificial intelligence." Nevertheless, it was my start. It is a pleasure to thank my high school classmate, Eric Marks, who taught me music theory and got me interested in that project.

Thanks to Alberto Riva for permission to use code from his CL-DNA project [107]. The gene sequence code in Chapter 2 is derived from this work, and additional examples in Chapter 7 elaborate on it.

Thanks to Richard Boyce, John Horn, Carol Collins and Tom Hazlet for permission to incorporate ideas from our drug interaction modeling project. This work began as a simple exercise for a class I teach on biomedical programming. Richard Boyce took it up, as a Biomedical and Health Informatics Ph.D. student, refined it and added the idea of managing evidence by categorizing it and using a Truth Maintenance System. John Horn, Carol Collins and Tom Hazlet contributed much pharmacokinetic knowledge as well as modeling advice.

The design ideas as well as almost all of the implementation of the Prism DICOM System at the University of Washington were the work of Robert Giansiracusa in collaboration with the author [67], as part of the Prism Radiation Therapy Planning System [68]. What I present in Chapter 13 is a simplified version with some variations, to explain the implementation. Our original DICOM source code is available at URL <http://www.radonc.washington.edu/medinfo/prism/>.

Thanks also to

\*\*\* huge list to be added later



## Part I

# Foundations of Biomedical Informatics



## Chapter 1

# Biomedical Computing With Symbols and Lists

For many centuries, biology and medicine have been organized principally as descriptive sciences, with much less emphasis on elaborate theories and deduction than on remembering and applying the right chunks of knowledge and experience to experimental or clinical problems. Now, that is changing, and a new era of biomedical science is here. This new era is not only one in which we are awash in a massive flood of data, but one in which deep computational power can be brought to bear on biomedical problems. It is the “coming of age” of a new intellectual discipline, *Biomedical and Health Informatics*.

Essential to this new discipline is the application of formal ideas such as logic, probability, and computational methods for organizing and manipulating large scale repositories of both data and knowledge. The data and knowledge take both numerical and symbolic form. The organizational structure of the data and knowledge is an essential part of the enterprise, not just a convenience. Biological data and knowledge are in fact sequences, tree structures, networks of relationships and so on. For the biologist, clinician, or public health practitioner to deal with this requires substantial familiarity with a language for expressing computational ideas, a *programming* language. For the computer scientist, mathematician, or statistician, it will require understanding biology, medicine and population health data and knowledge, and how it can be expressed in a formal computational language.

The goal of this book is to provide an exposition of these ideas and methods. To begin, this chapter introduces symbolic computing using the Common Lisp programming language. In the Preface (page xii) there is a brief explanation of the choice of Lisp over the more familiar programming languages. For the reader unfamiliar with Lisp, and for those who would enjoy a comfortable review, we develop in this chapter the essential ideas of Lisp, and the Common Lisp dialect in particular. I have also included some discussion of commonly used Lisp idioms, and some notes on my personal coding style and preferences. It is not a substitute for the excellent resources described in Appendix A.1, but should help the beginner

by providing some additional insights. For the expert, it may be interesting to see another programmer's view of this venerable and robust programming language.

The remaining chapters of Part I show how biomedical data and knowledge can be represented, organized and used in computational form. Chapter 2 focuses on data in various forms, from molecular sequences to medical images. It includes a brief introduction to database principles and techniques as well as how to represent data with tags and hierarchical structures. Chapter 3 presents ideas for representing and reasoning with knowledge in symbolic form, while Chapter 4 introduces elementary probability and statistical methods for representing and computing with uncertain biomedical knowledge. Chapter 5 describes how information retrieval systems work and how to deal with the challenges of indexing and retrieving biomedical documents and other kinds of biomedical information. In this chapter, you will also find a brief introduction to the basics of natural language processing and content-based image retrieval. While text and biomedical images may seem to be polar opposites in terms of content, some of the same techniques can be applied to both.

In Part II, a series of chapters develop more specific applications in biology, medicine, nursing, and public health. Finally, Part III takes up some important topics in building robust systems, including software and data engineering, software safety (life critical systems used in patient care), and data integration.

## 1.1 What can be expressed symbolically?

The most basic kinds of expressions we will use consist of *symbols* and *lists*. A symbol names some entity, perhaps something abstract or undefined, or a particular thing in the real world. In Lisp, a symbol is written without any quotes around it. It is a kind of *word*. Symbol names can include any of the normal alphanumeric characters and many other printing characters as well. Internally, symbols can have both upper case and lower case characters and symbols whose names differ only by case of some characters are different symbols. However, the usual way in which Common Lisp is configured is that the reader (when reading files as well as at the interpreter prompt) will convert all alphabetic characters to upper case. Moreover in the usual Common Lisp environment all the built in functions, constants and other operators are upper case, so it is OK to use lower case in files and when typing input.

Numbers also are used in Lisp, and they are represented as usual, written as integers or decimal numbers.

The string is also a data type. It is written by enclosing a series of characters in a pair of double-quote characters.<sup>1</sup>

The following are all symbols:

```
*
+
-
```

---

<sup>1</sup>Yes, there is a way to make the double-quote appear in a string, and I refer you to Guy Steele's book [127, page 34] for the details.

```

/
ira
FUBAR
house-57
a-big-long-symbol-with-$$-and-other-%+&*--characters
foo.bar.77

```

In accordance with the comments about case above, the symbol externally written as `house-57` is going to be the same symbol as `HOUSE-57` and `hoUSE-57`, and so on. All will be converted to the internal representation `HOUSE-57`.

Symbols, numbers, and strings are all examples of elemental data called “atoms”. Here are examples:

```

6734    45.9
6273647382509860711054623476
i-am-an-atom    ANOTHER
"a string atom"

```

A list begins with a left parenthesis and has symbols, numbers, lists or other Lisp data types in it, each separated by one or more *space* characters. Note that the `*` and `+` characters appearing below are actually symbols with names that are one character long.

```

(this is a list)

(this is (a (very nested) list))

(* principal
  (expt (+ 1.0 interest) years))

(G G A C C A C T T A A G)

```

A few symbols have special meanings. The symbol `t` is often used to represent the logical value, “true”, and `nil` is used to represent “false”. The symbol `nil` also is another name for the empty list, `()`. It is the only object in Lisp that is both a list and a symbol. As you will see in the discussion following, the symbols `*` and `+` and others like them are NOT ones with special meanings. However, like many other symbols, they have function definitions associated with them, as do symbols like `sqrt`, `expt`, and others. In Lisp, mathematical expressions are written with *prefix* notation, just like any other functional expressions, i.e., the function name comes first and then the arguments to the function follow.

Using parentheses to group together related data is extremely powerful. When writing mathematical expressions, it makes the expressions completely clear and unambiguous. With parentheses, you don’t need any precedence rules, i.e., in the expression  $a * b + c$  does it mean, “multiply  $a$  and  $b$  then add the result to  $c$ ”, or, “first do the addition, then the multiplication”? Writing the expression using parentheses and putting the operation symbols first leaves no room for doubt:

```
(* a (+ b c))
```

Structured descriptions are easy. Here is an address book entry, using nested lists:

```
(address-book-entry
  (name (John Quincy Jones))
  (street-address (1935 Underhill Blvd))
  (city andover) (state ma)
  (phones (home 617-254-6174)
          (cell 617-254-6175)))
```

Some of the components of the address book entry are labeled using a symbol as a tag, the first element of a sublist, and other components are simply organized according to a hopefully agreed-on convention. Both ideas are useful in organizing biomedical data.

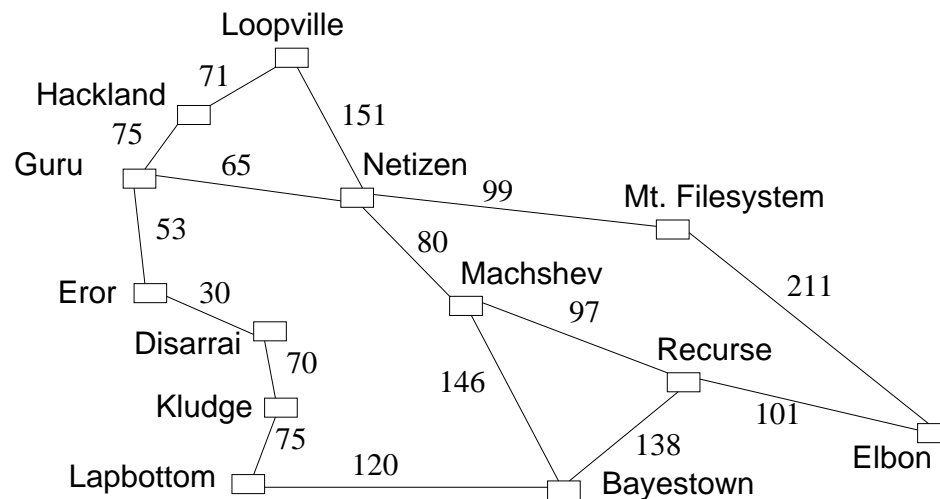
### 1.1.1 Biomedical examples

\*\*\* example from the human body, lymphatics or blood vessels.

\*\*\* Also, biochemical pathways

A road map is a kind of symbolic network:

\*\*\* replace the following with a map of the Seattle area, including interesting locations like UW, ISB, MS, FHCRC, ATL, Rosetta, etc.



```

((guru hackland netizen eror)
 (hackland guru loopville)
 (eror guru disarrai)
 (netizen guru loopville machshev
      mt-filesystem)
 (mt-filesystem netizen elbon)
 (machshev netizen recurse bayestown)
 ...)
```

Possibly could use roadmaps to analyse outbreak data?

### 1.1.2 Natural Language in Symbols and Lists

Text (articles, abstracts, reports, etc.) in ordinary human languages is the recording medium for much biomedical data and knowledge. Thus, searching computerized text is important for identifying documents or data records containing relevant information for example about disease and treatment, about protein interactions, or biological pathways, or about patterns of disease outbreaks. The text could be abstracts of publications, full text of articles, text based knowledge resources like GENBANK or other kinds of data and knowledge resources. As part of the process of building index information about such records, computer programs have to be able to decode the text stream into “words” (also called lexical analysis) and then organize the words in some useful way. After decoding, it is usually necessary to *parse* the sequences of words, i.e., to identify sentence structure and ultimately meaning.

Just looking at the parsing process for now, assuming that we have sentences as sequences of words, we can represent each word as a symbol, and each sentence as a list. If we are looking for sentence structure in terms of grammar, an effective way to represent knowledge about that is to create a *lexicon*, or dictionary of words in the language, along with their structural usage, i.e., is the word a verb, noun, adjective, etc. Grammatical structures can be represented by *rules*, that make a correspondence between a grammatical structure and its parts. Here is a small example, where in each expression the name of a structure or word type is the first symbol in the list, the = is a separator for human readability, and the remainder consists of all the possible words or structures that would match the structure or type.

```

(S = (NP VP))
(NP = (Art ADS Noun PPS)
      (Name) (Pronoun))
(VP = (Verb NP PPS))
(PPS = () (PP PPS))
(ADS = () (Adj ADS))
(PP = (Prep NP))
(Prep = to in by with on)
(Adj = big little blue green adiabatic)
```

```
(Art = the a)
(Name = Pat Kim Lee Terry Robin)
(Noun = man ball woman table)
(Verb = hit took saw liked)
(Pronoun = he she it these those that)))
```

In the above, the symbol **S** refers to “sentence”, **NP** is “noun phrase”, **VP** is “verb phrase”, **PP** is “prepositional phrase”, **PPS** is a sequence of prepositional phrases, and **ADS** is a sequence of adjectives. The lists following the individual parts of speech constitute the dictionary. The idea is that you take the words in the sentence in sequence and find them in the right hand side, one or more at a time, try to construct structures from them (using substitution by the symbols on the left hand side), and then continue with the next words, combining structures (which you can also look for on the right hand side).

Here are some sentence parsing examples using a small subset of the above rules. We imagine a function called **parse** that takes as input a list of words and returns the sentence structure with the words organized according to the rules and the dictionary.<sup>2</sup>

```
> (parse '(the table))
((NP (ART THE) (NOUN TABLE)))

> (parse '(the ball hit the table))
((S (NP (ART THE) (NOUN BALL))
    (VP (VERB HIT)
        (NP (ART THE) (NOUN TABLE))))))
```

Many sentences can be read several ways, all of which are grammatically correct. Sometimes the intended meaning is obvious to the reader, but for a computer program this presents a hard problem. This kind of ambiguity leads to more than one parse for a given sentence. The simplest approach is to initially generate all the possibly grammatically correct structures.

```
> (parse '(the man hit the table with the ball))
((S (NP (ART THE) (NOUN MAN))
    (VP (VP (VERB HIT) (NP (ART THE) (NOUN TABLE)))
        (PP (PREP WITH) (NP (ART THE) (NOUN BALL))))))
(S (NP (ART THE) (NOUN MAN))
    (VP (VERB HIT)
        (NP (NP (ART THE) (NOUN TABLE))
            (PP (PREP WITH)
                (NP (ART THE) (NOUN BALL))))))))
```

---

<sup>2</sup>Note that the symbols that are mixed case in the rules and dictionary have been converted to all upper case by the Lisp reader when they were read in. This is one of many details that are not critical to the idea but can be dealt with in a fully engineered natural language system.



Chapter 5 develops further the principles and methods of natural language processing, in the context of designing and using biomedical information retrieval systems.

## 1.2 The interactive Lisp environment

The theme of Lisp is expression evaluation using *function call*. In LISP, an expression is either a symbol or other single element (number, string, or more complex datum), or a list. A list that is an expression to be evaluated has as its first element a function or the name of a function. As an example, the expression `(sqrt 25)` is a two element list, whose first element is the symbol `sqrt`, which is the name of the square root function. It should give the result 5.0. In general, after the function, the remaining list elements are the inputs to the function. They are evaluated before being passed to the function itself, so they can also be expressions. Here is an example. The expression

```
(sqrt (+ (* 3 3) (* 4 4)))
```

means, compute the square root of the sum of 3 times 3 and 4 times 4. In this expression, the input to the square root function is the expression which is the list beginning with the `+` symbol, which is the name of the plus function. Its inputs are the two lists that begin with `*`, which is the symbol that names the multiply function. In the above, expressions are *nested*, i.e., functions can take arguments that are expressions.

In general, we use Lisp by interacting with the Lisp interpreter. The Lisp interpreter repeatedly

1. prints a *prompt*,
2. reads an expression typed by the user,
3. evaluates the expression, and
4. prints the result.

Where does the value go? It can be preserved as the value of a symbol (variable) by putting the expression inside a `setq` or `setf` expression. It can also be used inside any other expression, or as input to a function. The `setq` and `setf` operators have the side effect of attaching values to symbols (`setq`) or storing values in specified places (`setf`). This is analogous to assignment in FORTRAN-like programming languages. In the following, the angle bracket is the interpreter's prompt, and it is followed by the typed input of the programmer. Then the interpreter prints the result. Three such cycles of "read-eval-print" are shown.

```
> (sqrt 25)
5.0
> (setq var (sqrt 25))
```

```

5.0
> var
5.0
>

```

The result of the first evaluation is printed but not kept anywhere that we could refer to it. The `setq` expression returns the result, but also makes it the value of the symbol that is its first input, namely `var`. Now the symbol `var` has the value that is the result of the expression that followed it in the `setq` expression.

Although assignment is familiar to programmers exposed to most languages it is really unnecessary for many computations. Very large programs that do important calculations have no assignment expressions in them. They may have hundreds or thousands of functions defined, and when they are used, the user invokes one top level function, which uses the results of the other functions by nested calls and finally returns a result.

\*\*\* say more here? Refer to other functional programming languages, and related refs.?

### 1.2.1 LISP data types, values and expressions

So far, we have used symbols, lists, numbers and strings. Lisp supports subtypes of these general types, and also more complex structured types, such as arrays, functions, hash tables, streams (usually for input and output from/to files), built-in and user defined structures, and built-in and user defined classes. Lisp data types form a hierarchy. All Lisp objects are of type `t`, i.e., “Lisp object”. They subdivide into:

- numbers: integers (`fixnums`, `bignums`), ratios, floats (4 types), complex,
- characters,
- symbols (including `t`),
- lists (`conses` and `nil`),
- arrays (including strings),
- ... and more, to be introduced later.

The type of an object is associated with the object itself, and not with a symbol or location where the object may be referenced. A subtype or specialization of a type *inherits* all the properties of the more general type, and adds its own properties. A program can query the type of a datum, and take appropriate action accordingly. These ideas are the basis for object-oriented programming. The type system is extensible by the programmer, by defining additional types, structures or classes,

but the presence or absence of “classes” as such is not essential to the object-oriented programming idea.

The *value* associated with a symbol, a list element, or an array element, can be *any* kind of Lisp object. Here are some important aspects of values, expressions, and how the Lisp interpreter handles them.

- A symbol may have a value, and it also may have a function definition. These are separate.<sup>3</sup>
- The Lisp interpreter, when given a symbol, will look up its value and use that value in the expression in which it appears. If the symbol is the first thing in an expression, its function definition will be looked up and the function will be called with the remaining elements of the expression as inputs, or arguments, to the function.
- The `quote` operator is a special form, that prevents evaluation of its input.

`(quote foo)` means the *symbol* `foo`, not its value.

`(quote (+ 3 5))` means the *list* `(+ 3 5)`, not the *expression* that evaluates to the number 8.

- The `quote` operator is needed so often that it can be abbreviated by just putting the single quote character in front of a symbol or list, so

`'foo` is the same as `(quote foo)` and

`'(+ 3 5)` is the same as `(quote (+ 3 5))`.

In Lisp, type is associated with objects, not with the variables or places that may have these objects as the assigned values. Thus we don't usually specify symbols to have types. A symbol itself is of type `symbol`; its value can be any type (including another symbol). The following example session illustrates this:

```
> (setq ira 62.5)
62.5
> ira
62.5
> (setq ira '(the author of this book))
(THIS AUTHOR OF THIS BOOK)
> ira
(THIS AUTHOR OF THIS BOOK)
> (setq ira "stringing you along")
"stringing you along"
> ira
"stringing you along"
```

---

<sup>3</sup>This is one of the significant differences between Common Lisp and Scheme, the other major dialect of Lisp in use today. In Scheme, a symbol does not have a separate function definition; if the value is a function, the symbol can be used in an expression where function call is needed.

The symbol `ira` in this example first has as its value a number, 62.5. Then the value is changed to a list containing five symbols. Finally, the value is changed to a string of 19 characters.

Since lists are so useful and fundamental, the Lisp language provides primitives for constructing lists, and accessing their elements. A list is actually constructed using a more basic structure called a `cons`. A `cons` is a two-element structure, where each element is a pointer to something else, some Lisp object. These two elements can be obtained by the two operators, `first` and `rest`.<sup>4</sup> Just as their names suggest, the `first` function returns as its result the first element of a list, and the `rest` function returns the remaining elements of a list (in the form of a list). The result of calling the `rest` function is the list you would get with the first element left off.

### 1.2.2 Evaluation of Arguments, Special Forms and Macros

In general, before the interpreter passes arguments to functions, the arguments are *evaluated*. However, certain kinds of procedures *do not* evaluate their arguments. They are:

**special forms:** built-in procedures that do something out of the ordinary. The Common Lisp standard specifies these.

**macros:** procedures that transform expressions into other expressions before they are evaluated. You can write your own, in addition to the ones provided as part of standard Common Lisp.

If a procedure is neither a special form nor a macro, it is an ordinary function or a generic function, and its arguments are evaluated before it is called.

The ANSI standard for Common Lisp specifies that the following are the only special forms in the language:

<code>block</code>	<code>labels</code>	<code>quote</code>
<code>catch</code>	<code>let</code>	<code>return-from</code>
<code>declare</code>	<code>let*</code>	<code>setq</code>
<code>eval-when</code>	<code>load-time-eval</code>	<code>symbol-macrolet</code>
<code>flet</code>	<code>locally</code>	<code>tagbody</code>
<code>function</code>	<code>macrolet</code>	<code>the</code>
<code>generic-flet</code>	<code>multiple-value-call</code>	<code>throw</code>
<code>generic-labels</code>	<code>multiple-value-prog1</code>	<code>unwind-protect</code>
<code>go</code>	<code>progn</code>	<code>with-added-methods</code>
<code>if</code>	<code>progv</code>	

---

<sup>4</sup>These used to be called `car` and `cdr`, respectively, and traditional Lisp programmers still use the older names, but in Common Lisp, they are no longer preferred. The old names come from the first Lisp implementation on an old IBM mainframe computer.

### 1.2.3 Side Effects

We have already seen an example of a side effect, when using `setq`. The value returned also becomes the value associated with a symbol. In general, to cause side effects you use a macro, a special form, or a function that includes a macro or special form.

The `setf` operator causes side effects. It is similar to (and a generalization of) the `setq` operator. The `setf` operator can assign values to symbols, and it can also attach values to more general names of places, e.g., places within a list, array, or structure or a slot within a class.

The following functions do *not* cause side effects:

- `first`, `rest`, `last`, `butlast`, `nthcdr`, `assoc`
- `cons`, `append`, `list`, `reverse`
- `length` and the arithmetic functions.

The benefit of using such functions is that they are safer than ones causing side effects. When side effects happen that are unintended, one's data can be altered in a way that causes calculations to break. On the other hand, many of the above functions allocate additional memory, as they make copies of list structures. This temporary storage is eventually discarded and needs to be reclaimed. In Lisp this is automatic. It is called "garbage collection". Having automatic garbage collection is a huge convenience, but can have also a huge negative impact on the speed with which things get done.

It may be tempting to the beginner to write highly optimized code that uses the faster operators that cause side effects and modify structures in place, but it is an extremely bad idea. My advice is to write programs that are clear and verifiably correct first, even if they are horribly inefficient. Once you are sure you understand the solution of a computing problem, and have correct code, you can then transform it so that it runs faster, while maintaining correctness. It is much harder to write fast broken code and then fix it later. Finally, it is often more effective to understand the problem in a way that allows for a fundamentally faster algorithm, rather than depending on such low level optimizations as using destructive functions instead of non-destructive ones. You can still make the low level optimizations once you have a really clever algorithm.

## 1.3 Some Essential Programming Ideas

In this section we briefly describe some of the most common functions and macros used in programming. Most of these elements are common to all programming languages. Here we describe the form they take in Common Lisp.

### 1.3.1 Creating new functions from existing ones

Some of what we need to do with biomedical data may only involve writing some short expression and evaluating it. As an example, many useful queries on the

EcoCyc and other pathway-genome knowledge bases [74] are just short expressions that can be typed into the interpreter. However, to build large scale resources like EcoCyc requires writing really large programs. An effective way to build a large program is to write many smaller functional units and combine them to make bigger functional units, and so on. This is the general approach of *layered modular design*. To do this in Lisp, we need a way to define new named functions using existing ones. Then we can build further functions using the ones we have defined.

The `defun` macro associates names with function definitions:

```
(defun square      (x)      (*      x      x) )
  ^          ^          ^          ^          ^
```

To "square" something multiply it by itself

Then we can use the new function, by name:

```
> (square 6)
36
```

A function defining expression starts with the symbol `defun`. The first argument (not quoted) is the symbol to be associated with the new function. In the example above, it is the symbol `square`. The next argument is a list containing symbols naming formal parameters that are inputs to the function. These symbols can be used in the expressions in the body of the function. They will have values assigned to them when the function is called. In the example above, there is only one input, denoted by the symbol `x`, so the argument list has just that one symbol. The remaining arguments are expressions to be evaluated as the *body* of the function. The value of the last expression is the value returned by the function. So, as you can infer, if there is more than one expression, all except the last are there to produce temporary or permanent *side effects*. In the example above, there is only one expression, `(* x x)`, and its value is the value returned by the function when it is used.

If the first of these expressions is a string (and there are one or more expressions following), it becomes the *documentation* associated with the function. This is *not* the same as a "comment" as used in other programming languages (Common Lisp provides for comments, too). The documentation string, if provided, can be retrieved by code in a running program. Thus it is possible to provide interactive local documentation. The `describe` function, a built-in Common Lisp function, returns the documentation associated with its argument. Here is an example.

```
> (defun pythagoras (x y)
  "returns the hypotenuse of the triangle
  whose sides are x and y"
  (sqrt (+ (* x x) (* y y))))
PYTHAGORAS
> (describe 'pythagoras)
PYTHAGORAS is a SYMBOL.
It is unbound.
```

```

Its function binding is #<Interpreted Function PYTHAGORAS>
  which function takes arguments (X Y)
Its property list has these indicator/value pairs:
DOCUMENTATION  "returns the hypotenuse of the triangle
                whose sides are x and y"

```

Just as `quote` gives the symbol or list itself, `function` gives the function object associated with the symbol or function expression. However, be careful: `quote` is a special form that does not evaluate its argument, but `function` is an ordinary function and *does* evaluate its argument. In the expression `(function plugh)`, the symbol `plugh` will be evaluated, so if `plugh` does not have a value or the value is not a function, an error will result. Usually we want to look up the function definition associated with a symbol, not its value. For this purpose, the argument to `function` should be the symbol `plugh`. Thus the right way to do this is to *quote* it, i.e., `(function 'plugh)`. This idiom is used so often that it also has a special abbreviation, just as `quote` does: `(function 'plugh)` can be abbreviated as `#'plugh`.

### 1.3.2 Formal Parameters and Local Bindings

\*\*\* also include here some intro to the ideas of scope and extent

A function that has formal parameters creates local bindings for those parameters. For example, in

```

(defun pythagoras (x y)
  (sqrt (+ (* x x) (* y y))))

```

the symbols `x` and `y` inside the function body refer to the values passed in as arguments, not to values outside the function.

```

> (setq y 12)
12
> (setq x 5)
5
> (pythagoras 3 4)
5.0
> (pythagoras x y)
13.0

```

Inside the first call to `pythagoras` the bindings of `x` and `y` are 3 and 4. Inside the second, the bindings are 5 and 12, because the (external) values of `x` and `y` are used as the inputs, and the symbols `x` and `y` are *rebound* to these values.

Inside the body of a function, the local binding of a formal parameter can be changed, but it does *not* change the binding of the same symbol outside the function.

Suppose we have a function that attempts to do this:

```
(defun bad-pythagoras (x y)
  "bad-pythagoras attempts to double
   x and y after using them"
  (let ((tmp-x x)
        (tmp-y y))
    (setq x (* 2 x))
    (setq y (* 2 y))
    (sqrt (+ (* tmp-x tmp-x)
              (* tmp-y tmp-y)))))
```

We set `x` and `y` and then use `bad-pythagoras` with `x` and `y` as inputs.

```
> (setq x 3)
3
> (setq y 4)
4
> (bad-pythagoras x y)
5.0
```

Now, what are the values of `x` and `y`?

### 1.3.3 Predicates, truth and falsehood

All languages have a way to specify the values, *true* and *false*. A function whose return values are either of these is called a *predicate* function. With such values, one can then construct conditional expressions, which evaluate different expressions depending on the result of some test that returns true or false.

- `t` and `nil` are special (constant) symbols
- For use in logical expressions, anything not `nil` is effectively the same as `t`
- Predicate names often end with the letter “p” but *not always*

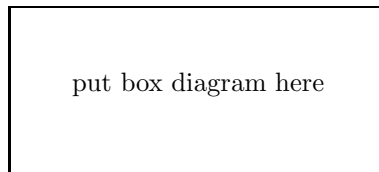
Here are examples using the `listp` function.

```
> (listp '(g c g c a a t))
T
> (listp 6.0)
NIL
```

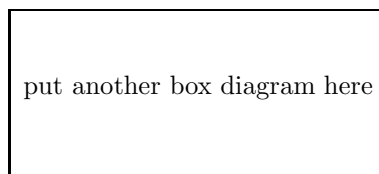
### 1.3.4 Copies and Identity

In the Lisp environment, every symbol is unique. Lists, however, can *look alike* but be copies of each other. A list can be depicted as a sequence of two compartment boxes, in which the first compartment contains a pointer to some thing, and the second compartment contains a pointer to the next box in the list.



Figure 1.1: Lists are connected `cons` cells

`cons`, `append`, `list` and `reverse` each make *new* lists. So, `(cons (first foo) (rest foo))` gives a list that looks like `foo` but is not exactly the same list. Instead, the call to `cons` creates a new box, in which the first compartment points to the first element of the original list, and the second compartment points to the rest of the original list. So, the original and the new one share some list structure, but they are not identical; only their “contents” are identical.

Figure 1.2: `cons` creates a new box

There are other list functions that actually modify the lists that are input to them. Such functions are called *destructive*, and should be used with caution.

### 1.3.5 Testing for equality

There are many tests that compare two objects:

- `eq` tests for strict object identity (this may or may not apply to strings or numbers)
- `eq1` is like `eq` but also returns true if the two objects are numbers of the same type and value or are character objects that represent the same character.
- `equal` returns true if the two arguments are structurally isomorphic (usually this means they print the same way).

The following all operate on numbers, and perform type coercion as necessary.

- `=` returns true if the arguments are numerically equal
- `/=` returns true if the arguments are numerically unequal
- `<` returns true if the arguments are monotonically increasing

- `>` returns true if the arguments are monotonically decreasing
- `<=` and `>=` test for monotonically nondecreasing, and nonincreasing, respectively.

### 1.3.6 Keywords

Keywords are symbols beginning with the `:` character.

Many functions that take sequences (e.g. lists) as inputs have optional keyword arguments. Keyword arguments follow required and optional arguments, and consist of keyword value pairs.

Parameter	Purpose	Default
<code>:test</code>	function for comparison	<code>eql</code>
<code>:key</code>	“lookup” function	<code>identity</code>
<code>:start</code>	starting position	<code>0</code>
<code>:end</code>	ending position	<code>nil</code>

Examples:

```
> (member 'a '((c d) (a b)) :key #'first)
((A B))
> (remove '(peter "gene clinics")
          '((ira "radiotherapy planning")
            (peter "gene clinics")
            (brent "ngi mania")))
:test #'equal
((IRA "radiotherapy planning") (BRENT "ngi mania"))
```

### 1.3.7 Logic functions

The following logic functions help form complex expressions:

- `and` evaluates arguments from left to right until one gives `nil` or all are non-`nil`.
- `or` evaluates arguments from left to right until one gives non-`nil`, or all are `nil`.
- `not` returns `nil` if its input is non-`nil`, and `t` if its input is `nil`.

### 1.3.8 Conditional expressions

A conditional expression can return different results according to a test (using a predicate expression).

```
(defun spy-check (password)
  (if (equal password
            "The moon is blue")
      "Here's the goods"
      "Foo, you are a phoney"))
```

The two conditionals, `cond` and `if` are equivalent, in that any expression written with one can be rewritten using the other.

(`if a b c`) is the same as (`cond (a b) (t c)`)  
and

```
(cond (test1 result1)
      (test2 result2)
      (test3 result3))
```

can be written as

```
(if test1 result1
    (if test2 result2
        (if test3 result3)))
```

Why have both? For convenience. When to use one or the other? A rule of thumb is to use `if` for situations where there are just two alternatives and a single test, and to use `cond` instead of nested `if` expressions when there are a series of tests and for each a single procedure to follow if a test is successful.

More convenience - short forms for special cases:

- (`when a b`) is the same as (`if a b nil`)
- (`unless a b`) is the same as (`if a nil b`)
- A `case` expression can also be rewritten in terms of an `if` or `cond` expression.

## 1.4 Advanced topics

\*\*\* elaboration of the idea of functions as data, use of pound-quote, `function`, `funcall` and `apply`.

\*\*\* some kind of simple discussion here about the lambda calculus?

### 1.4.1 Anonymous functions

`lambda` creates an anonymous function:

```
(lambda (x) (* x x))
```

This function of something multiplies it by itself

(`quote (lambda (x) (* x x))`) is a list.

(`function (lambda (x) (* x x))`) is the function defined by the “lambda expression”, (`lambda (x) (* x x)`).

Note that

```
(function (quote (lambda (x) (* x x))))
```

will result in an error, since a list is not a function.

### 1.4.2 Functions can be used as data

\*\*\* rewrite this to use lambdas and put them in a table by apartment number, e.g.,  
(name number fn)

```
(defun spy-check (greet password spy)
  (if (equal password
              (funcall spy greet))
      "Here's the goods"
      "Foo, you are a phoney"))
```

```
(defun goldberg-12 (greeting)
  (if (equal greeting
              "The moon is blue")
      "The Golem lives"
      "I beg your pardon?"))
```

### 1.4.3 Declarations and Optimization

Often in code you will see *declarations*, e.g., (`declare fixnum start`). Except for **special** declarations, no declaration should have any effect on the correctness or meaning of a program. It can be *useful*, even though unnecessary, to declare the type of object that will always be the value of a symbol, if you have made sure your code will work this way. Such declarations can be used by a good compiler to generate very fast compiled code. This is important when writing code for numerically intensive computations, when using arrays and vectors (where all the elements are of the same type), and in other circumstances as well.

For example, if you (the programmer) can guarantee that a certain array's elements are always exactly of type (**unsigned-byte 16**) and that the array is a **simple** array, the compiler can generate very fast array accesses. If a body of arithmetic expressions have elements that are of a guaranteed type and don't require type conversions, similarly the compiler can generate code that uses registers effectively and does not waste time or space on managing generic types.

There are tradeoffs. Providing declarations and compiling with a high optimization setting can eliminate important information that would otherwise be available when debugging. Error messages become much more obscure, and often entire sections of code may be restructured by the compiler so it is harder to determine where the error occurred. The best strategy is to design your code to be correct first and foremost, and to optimize it after it is determined to be correct.

### 1.4.4 Macros - Code That Writes Code

### 1.4.5 Special Features of Lisp

- Data, not variables, have types.

- Lisp supports lists.
- Functions are also data.
- Lisp is extensible (macros).

## 1.5 Object-oriented Programming in Lisp - CLOS

Implementation of an object-oriented design can be done with most any programming language [61, 62], but the work is facilitated by using a programming language with the right abstractions already built in. The essential abstractions are: the generalization of types to include abstract (or structured) data types (including user defined types) that have attributes, the idea of inheritance or specialization of types, and the idea of type-based dispatch in implementing functions or procedures. Usually the idea of having different blocks of code to execute for different types of inputs is described by saying that functions are implemented as collections of *methods*, where a particular method is associated with a particular type of input (or inputs).

Here we provide a brief introduction to these ideas as implemented in the Common Lisp Object System (CLOS). A concise introduction to the basic ideas of CLOS may also be found in Graham [41, Chapter 11]. A more complete and detailed discussion is in Steele [127, Chapter 28].

### 1.5.1 Types, Structures and Classes

Every programming language provides some way to represent *types*. A type definition specifies what kinds of values a variable or entity of that type may have. The types that are supported by most all programming languages include: numbers of various sorts, characters and strings (sequences of characters). More complex types, such as arrays, enumerated types and user-defined structures are supported by many programming languages. Lisp is distinguished by supporting symbols, lists and functions as built-in types, in addition. Numbers are usually subdivided into *subtypes*, such as integer, rational, float, complex and others.

There is no need in Lisp for what are called “enumerated types” in other languages, where one defines a sequence of names to be allowed values of a type. An example of this would be to define the names of colors as forming an enumerated type called “color”. In Lisp, we would just use symbols naming the colors, and there would be no need for a statement defining them.

In Lisp, the types are arranged in a type hierarchy, so that any datum belongs to the overall type, *Lisp object*, sometimes referred to as type `t`, and may belong to a series of successively more specialized types. For example, the number whose text representation is 3.14159 is of type `single-float`, which is a subclass of `float`, which is a subclass of `real`, which is a subclass of `number`, which (finally) is a subclass of `t`. Similarly, in Lisp, strings are implemented as character vectors, thus the `string` type is a subclass of `vector`, which is a subclass of `array`, which is a

subclass of `t`. The hierarchy of types is important for defining *inheritance*, and for dealing with *function dispatch*.

User defined types are created using either `defstruct` or `defclass`. `Defstruct` creates *structure* types, and they become a part of the type hierarchy just as if they were built in. Similarly, `defclass` creates user defined *classes*, which also become part of the type hierarchy. Each of these facilities provides a way to create a “record-like” structure, in which a datum contains named components, called *slots*, each of which can be referred to by name, rather than relying on the position of the component in an array. A structured type may have different kinds of values in different slots. The programmer need not be concerned with how the data are actually stored in memory.

In most discussions of object-oriented programming, the focus is on user defined types, so one might get the impression “Object-oriented programming = classes”, but this is most certainly *not* true. In the Smalltalk programming language, everything is an instance of a class of some sort, and the built-in classes are treated the same as the additional user-defined classes. This is also true in Lisp. On the other hand, although in java it is claimed that everything is a class, in reality, much of java’s syntax and semantics are about managing namespaces.

this paragraph needs some work, not sure I will leave it in, as it is a bit too polemic

\*\*\* add note on `defstruct` here

User defined types (defined either by `defstruct` or `defclass`) share the possibility of utilizing *inheritance*.<sup>5</sup>

The type system and the idea of a class are closely integrated in Common Lisp. Every type corresponds to a class, and every class definition results in a type definition as well. There are three kinds of classes in Common Lisp, *built-in* classes, *structure* classes and *standard* classes. The built-in classes have restricted capabilities and cannot have user defined subclasses. Many of the standard Common Lisp types correspond to built-in classes, e.g., `number` and its subtypes, `array`, `list`, and others. Each structure type defined by `defstruct` corresponds to a class that is an instance of `structure-class`. Similarly, classes defined by `defclass` are instances of `standard-class`. We will usually just refer to an instance of `standard-class` as a class (without qualification), where the meaning is clear. When using “class” to refer to a built-in or structure class, the context should make the meaning clear also.

### 1.5.2 Generic Functions and Methods

The idea of a generic function is that the function’s body (the code that executes when the function is called) is made up of methods (defined by `defmethod`). Each method is identified with a particular combination of the types of the arguments to the function. So, the same function call with different types of inputs will result in different methods being used.

---

<sup>5</sup>Built-in types also exhibit inheritance, where there is a type-subtype relation, as for numbers and sequences, for example.

A program can add methods to any generic function, though one should not do this unless the generic function's author has specified how this should be done. If a function has been defined using `defun`, it will be an ordinary function, and one cannot subsequently define methods for it.

\*\*\* put example here, as well as cite to Graham or others

In CLOS, methods are associated with functions, not with classes. This is different from the more common object-oriented programming languages (C++, java, others??) in which methods are associated with classes. The other object-oriented languages implement the capabilities of operations with instances of the corresponding classes. However, that makes it very awkward to have functions that dispatch based on the type of more than one argument. In CLOS, dispatch can be based on any number of function arguments. You just write a method for each combination of argument types.

Since classes are just an extension of the Common Lisp type system, one can write generic functions with methods for *any* types, whether those types are built in, defined by `defstruct` or defined by `defclass`. Thus one could define a generic function that performed an abstract operation on a variety of data types, implementing that abstract idea with different methods according to each data type. It is not necessary for the classes that specialize methods to have a common "base class" in CLOS, since everything already is a subclass of the class `t`.

\*\*\* need an example here, maybe print-object

Generic functions support inheritance, just as classes and slot definitions do. So, if a generic function has a method for some combination of input types, that method will *also* be applicable for any combination of input types in which each argument is a *subtype* of the type for which the method was defined.

Here is an example. The Prism Radiation Treatment Planning system (described more fully in Chapter 14) provides a wide variety of control panels for drawing anatomy, positioning radiation beams, viewing the simulation geometry, etc. There is a `generic-panel` class, with a method for the generic function `destroy`. Each panel is a subtype of `generic-panel` so this method is applicable to each of them. This is a simple consequence of the fact that if an entity belongs to a certain type, it also belongs to all the parent types, so an instance of a `beam-panel` is also an instance of a `generic-panel`.

This can be seen by the results returned by the two built-in functions, `type-of`, which returns the most specialized type an entity belongs to, and `typep`, which returns `t` if an entity is of a specified type. The following example also illustrates that the ideas of type and class apply to the entire type hierarchy, not just the complex types defined by `defstruct` and `defclass`.

```
> (type-of 3.14159)
SINGLE-FLOAT
> (typep 3.14159 'number)
T
```

Finally, an important element of Lisp that is not found in most other programming languages is that functions are also data, and the type `function` is also part of the type hierarchy. Thus, one can have variables whose values are function objects, and your program can store function objects anywhere you might store anything else. Furthermore one can write functions that create new function objects, which can then be used just as if they had been there all along. This will be particularly important as we move from data to knowledge in Chapter 3.

## 1.6 More Notes

\*\*\* clean up this section...

In Lisp, one can attach values to named constants, for use anywhere in a program, using the `defconstant` function. Similarly, `defvar` establishes global variables that can be referenced anywhere in your program. To associate a body of code with a name, as a closed subroutine, `defun` is used. In Lisp, all expressions are lists, beginning with the name of a function (or a macro or special form). The remaining elements of the list are the arguments to the function. In the case of defining forms like `defun`, the name of the function, variable or other comes next, and the remaining arguments depend on the particular form. Most defining forms allow for documentation strings, which are ignored when the function is called, but which are available to a running program.

\*\*\* Here put some discussion of the more usual idioms, etc.

Also, some discussion of the differences among functional, procedural and declarative programming, rule-based programming, OOP.

If a function or macro (or other symbol usage) appears in code and has not been defined previously, it is likely to be a built-in, part of the Common Lisp language specification. To check this and find out more about such functions, macros or other entities, the reader should look them up, e.g., in Appendix D of Graham [41], which has a complete and concise summary of all functions, macros, special forms, variables and parameters defined in the ANSI Common Lisp specification. More detailed explanation can be found in Steele [127]. Both books have excellent and thorough indexing, with respect to these definitions. For other symbols, which may not be part of the ANSI standard but are defined elsewhere, a reference is included where they first appear. Examples of such additional sources for predefined code are: The Common Lisp Object System Metaobject Protocol [78], which is not yet part of the ANSI standard, and the CLX library [116], which implements a Common Lisp binding to the X window system protocol.

## 1.7 The Meta-Object Protocol (MOP)

See Kiczales, et. al. [78].



This is not yet part of the ANSI standard, but supported in most Common Lisp implementations.

\*\*\* write some kind of quick and simple intro here. Also, include stuff on Java, CORBA, interfaces, etc.

The idea that a running program can gain access to information about its own data structure definitions has been present in Lisp for a long time, and is just now attracting attention in newer programming languages and environments. It is sometimes called “reflection”. It has only recently become available in Java programming environments. The way it works in Lisp is that the Common Lisp Object System (CLOS) itself is implemented using the same machinery that the application programmer uses. The way classes are implemented is that they are themselves objects, instances of a “higher level” class (a *metaclass*) called `standard-class`. The class `standard-class` is called a metaclass since its instances are “class” objects. It and the classes created by our code are examples of “metaobjects”. CLOS has other metaobjects as well, for example generic functions, which are instances of the `standard-generic-function` class.

In Lisp your code can check on the type of an object, retrieve its class, and retrieve properties of classes and objects. How this all works is specified in the CLOS Metaobject Protocol, or MOP [78]. Chapter 1.7 gives a brief introduction to the basic ideas of the Metaobject Protocol, and a little background about related ideas in other programming languages and environments.

In Chapter 2, Section 2.2.3 we show how the MOP can help solve the problem of storing complex medical data in the style of keyword-value pairs, without writing a lot of extra code. Our plan is that the code will use the class definition itself as metadata, to decide what to read and write.

One of the MOP ideas is that it should be possible for a running program to find out what the names are of the slots for a class. The MOP function `class-slots` gives us a list of the slots (actually the `slot-definition` metaobjects, since slots are described by `slot-definition` metaobjects), including inherited ones from superclasses, that are available in any particular class. Their properties can be examined using accessors, just as for other objects. From the `slot-definition` objects we can get their names (as symbols), which we can then use with `slot-value` to read the value stored in each slot, and with `(setf slot-value)` to write a value in a slot (as when restoring an object from the serialized data in a file. The MOP function `class-slots` requires as input a class, which is easy to obtain. For any object, the MOP function `class-of` returns its class (the actual class metaobject, not the name of the class). Since slots are implemented with metaobjects, we can use the MOP function `slot-definition-name` to obtain the name of a slot, given the slot itself. To get a list of all the slot names in a class, we iterate over the list of slot definitions using `mapcar`.

```
(defun slot-names (obj)
  (mapcar #'slot-definition-name
    (class-slots (class-of obj))))
```

The functions `slot-definition-name` and `class-slots` may be in a different package than the `common-lisp` package, but these are the names defined in the MOP documentation.<sup>6</sup>

\*\*\* replace the following with an application to the example class to be provided above.

Applying this to the `patient` class instance on page 55, we get

```
> (slot-names temp)
(NAME HOSPITAL-ID AGE ADDRESS DIAGNOSIS ...)
```

Thus, the MOP gives a way to find out about a class's slots, iterate over them, writing the contents of each, and recursively doing the same for anything that is itself a class instance rather than a Lisp type that already has a standard printed representation. This can be done in a way that is human readable and allows a symmetric function for reading the data back in and constructing an instance equivalent (`equal`) to the original. These slot name symbols can be used as input to the standard CLOS functions `slot-value` and (`setf slot-value`) to read and set data in the slots of an object. The standard constructor function in CLOS for creating instances of classes, `make-instance`, also can take a symbol as input that is the value of a variable, so one can write code that creates and manipulates class instances without having any class names or details about the class explicitly written in the code.

\*\*\* an aside or footnote on CORBA, "interfaces", reflection programming, perhaps in an appendix? How much of this can and can't be done easily or at all in C++ or java?

## 1.8 Lisp Promo

\*\*\* develop the list in the Preface in more detail here

Lisp is a fine general purpose programming language. It has been shown to be easier to use than C or java, and compilers are available to generate code that runs as fast or faster than equivalent code written in C. It certainly supports higher performance than java. A recent study by Ron Garrett (formerly Erann Gat) [37] demonstrated these and other important points.

\*\*\* develop this list here

Important things to know about Lisp:

---

<sup>6</sup>In Allegro Common Lisp<sup>TM</sup> (Franz, Inc.), these functions are external in the CLOS package. In CMUCL a more complex accessor is needed in place of `slot-definition-name` and `class-slots` is in the PCL package. In CLISP, the name corresponding to `slot-definition-name` is `slotdef-name`; this and `class-slots` are internal in the CLOS package. Other implementations may have other versions, since the MOP is not yet standardized.

1. Fewer lines of code than other languages to solve similar problems, which also means less development time, fewer programmers, lower cost, less bugs, and less management overhead,
2. Easy to debug, lots of built in tools and facilities, ease of finding errors, because of structure,
3. Lots of libraries and freely available code base, lots of support, and yes, there are Lisp programmers,
4. Myths about Lisp are untrue (list here),
5. and most important, Lisp provides a superb interactive programming and execution environment.

## 1.9 Some Programming Language History

When you look closely, a computer program is a sequence of binary numbers, stored in a computer as settings of electrical circuits, either “on” or “off”. The binary numbers can represent data or operation codes for the computer processing unit. Text data could be represented by assigning a numeric code to each character in the alphabet, along with punctuation marks as well. This idea had a precedent in the early teletype machines and in telegraphy (Morse code). Early computers were programmed by toggling switches to set these binary numbers in the computer’s memory. Later, more convenient input methods were invented, such as paper tape and punched cards.

Soon it was apparent that not much could be done if programming were so difficult. Assembler programs were invented, so that the operation codes could be named in a program rather than writing numbers, e.g., ADD for the addition operation, SUB for subtract, JMP for a change in the sequence of executing the operations, and so on. Assembler programs also introduced the idea of a symbolic address, so that you could add more instructions, changing the numeric location of data and other instructions, but the assembler could still resolve the symbols and translate to the right addresses (locations, or places in the program code or data). Assembly languages also included the idea of a closed procedure, or *subroutine*, which performs some operations on its inputs, returns some result, and can be used anywhere in a program that includes its definition. This idea of a closed procedure provides a way for the programmer to define complex operations that can be used just like built-in operators. This is a powerful means to write very complex programs but keep them manageable. It is also a fundamental idea in mathematics and in many areas of design and engineering. You can design an assembly or unit of some sort, and then use it as an element of a larger assembly or unit, without concern for the internal details. Examples in ordinary life are everywhere. The doors and windows of a house are often self-contained standard pre-built units. (more examples?)

In 1957, two engineers from IBM invented a “high level” language, FORTRAN, in which programmers could write programs that utilized textual renditions of mathematical formulas, as well as more abstract forms of control instead of machine instructions such as JMP. It was such a huge success that it is still in use today, and is the oldest programming language currently in use. In FORTRAN, one uses named variables and mathematical symbols, and a compiler translates the text into the appropriate (equivalent) binary numbers, in the machine language of the computer you are using. FORTRAN also supports the definition and use of subroutines.

One year later, in 1958, John McCarthy invented a yet more abstract language for writing programs that could be used to represent and study higher level mathematical ideas. It was based on the idea of abstract symbols (again represented as text using a character code) and *lists*, which were sequences of symbols. Although the symbols could be used as variables in the same way as in FORTRAN programs, they could also be manipulated as abstract data, without assigning any values. Thus one could create lists of symbols, lists of lists, and by extension, arbitrarily complex structures, and perform operations on them as if they were themselves simple entities like numbers. This is similar to mathematica abstractions like algebra (factoring polynomials, for example, is symbol manipulation, not arithmetic).

McCarthy also incorporated the idea that closed procedures, corresponding to mathematical functions, should also be elements of the language, not only to define and use, but as actual data, i.e., you could write a program that created and manipulated functions in the same way as numbers, symbols and lists. He did this by making functions take the form of lists beginning with a special symbol, `lambda`. Since they are lists, they can be constructed, modified, analyzed, stored as the value of a symbol (variable), as well as executed.

This could not be done in FORTRAN, because the text representing FORTRAN code did not correspond in any way to the internal representation of the compiled code. While you could write a FORTRAN program that wrote FORTRAN programs or subroutines, the same program could not then directly use (execute) what it wrote, in the context of itself.

\*\*\* need better explanation of compiled vs. interpreted languages and difference between code as text input to the compiler and code as data to be manipulated

McCarthy’s new language, which he called “LISP” (for LISP Processing), is (like FORTRAN) also still in use today, making it the second oldest actively used programming language. Unlike FORTRAN, it can be used in an interactive mode, where the programmer types expressions, they are executed (evaluated) in the current environment, and they can have the effect of modifying the environment, e.g., by adding more function definitions. It can also be used in the compile-load-execute style of FORTRAN, but that is less common.

To summarize, LISP is:

- Functional programming - expressions are evaluated and the results are returned,
- Symbol processing - in addition to numbers, characters, arrays etc., LISP

need some sentences here on the Lambda calculus

manipulates symbols and lists,

- Interactive - the environment can be built incrementally, parts inspected and tested, big functions built on the fly from smaller ones.



## Chapter 2

# Biomedical Data

*Just the facts, ma'am...*

– Sergeant Joe Friday  
Dragnet TV series, circa 1956

It's a truism to say that biology, medicine and public health are flooded with facts. The complete genetic sequences of many organisms from bacteria to human beings are now available. We also have many terabytes of data on the structures of the proteins in these organisms. The biomedical literature is rich with laboratory results, case studies, and clinical trials reports. Repositories exist that contain large amounts of public health surveillance and epidemiologic data. Electronic medical record systems are being deployed in many hospitals. It is a wonderful time for research and for the potential improvement of health care practice, except for one big problem: there is no coherent system for relating all or even part of the data. Worse yet, every repository and source of information seems to have its own arcane syntax and semantics for organizing and representing the data. It makes the Tower of Babel, with only 70 different languages,<sup>1</sup> look like a simple problem.

Biomedical data come in many forms. They include data about the fundamental molecules of life, the small molecules that are used by living organisms for energy and as building blocks for reproduction and growth, and the large molecules that encode all the information about molecular processes such as metabolism and regulation of cellular activity. At a higher level in complex organisms like humans, we have a large amount of data about physiological processes and the behavior of organs such as the heart, data about individual patients in hospital electronic medical record systems, and public health surveillance data, to mention just a few.

Cells, the basic unit of structure of living organisms, contain large molecules such as DNA and proteins. DNA is a long double stranded helix, where each strand

---

<sup>1</sup>The story, in Genesis, Chapter 11, names the 70 nations descendant from Noah, initially all speaking the same language, but one could infer that the “confounding of language” that happened gave each their own language

is a sequence of units called nucleotides. Four different nucleotides are found in varying quantities and in various places along the DNA strand. Each nucleotide is made up of a sugar molecule, a phosphate group, and one of the four bases, “guanine”, “adenine”, “cytosine” and “thymine” (symbolized by the letters G, A, C and T). Nucleotides are labeled by the names of the bases that are attached to the sugar-phosphate “backbone” units. The pattern of nucleotide sequences is different in different kinds of organisms and even between different organisms in the same species.

Sections of this huge molecule (DNA) consist of *genes*, which, when activated, are transcribed (copied into a similar kind of molecule, RNA, which differs only in that the base Thymine is replaced by another base, uracil, denoted by the letter “U”). The RNA strand, called “messenger RNA”, is then translated into a *protein*, a sequence of amino acid molecules, of which there are twenty different kinds. The proteins in turn regulate many of the activities of the cell, and thus the living organism, and also serve as building blocks for cellular material, thus also for tissues, and organs. The complete DNA sequence for a given organism is called its *genome*. The genomes of many organisms contain over a billion such nucleotides. Biologists have determined the genomes (partially or completely) for hundreds of organisms. Moreover, much information is now available for hundreds of thousands of individual genes and proteins, including how they interact with each other, and how activity levels of genes and proteins vary with different experimental conditions.

More background on molecular and cellular biology can be found in an article by Larry Hunter in the AI Magazine [55] and the opening chapter in an earlier book [54] on which the article is based. A deeper treatment of this subject will be available in a forthcoming text by the same author [56]. For the ambitious reader, standard textbooks such as [2] go into even more comprehensive detail.

\*\*\* add in a little here about the Physiome project and the Virtual Human, clinical laboratory data and the EMR/clinical picture, as well as public health - reportable diseases, epidemic tracking, etc.

All of these have some biological and technical aspects in common. Some of the technical aspects include: how to encode something that is not a numeric measurement (gene and protein sequences, organ names and relationships, descriptions of clinical findings, names of diseases, etc.), how to organize both numeric and non-numeric data for easy retrieval and interpretation, how to represent hierarchical relationships or groupings of data that have an *a priori* relationship (e.g. a collection of genes whose expression levels are being measured, or a set of related blood tests). Some of the biological aspects include: how to represent higher level phenomena (e.g. phenotypes, diseases and therapies) in relation to lower level ones (e.g. genotypes, cellular and organ level processes, and body structure), and how to systematically and consistently assign names and classifications to all the entities, processes and relationships [111].

The representation and organization of biomedical data require consideration of three dimensions of ideas.

**Context** Data will need to be represented in the context of running programs,



where the data are volatile (when the program terminates, the data disappear). In addition, it is necessary to have a representation of data in files or other forms of non-volatile storage, so that they have long term persistence, and can be searched, retrieved and updated. Finally a third context is that of network communication, where the data need to be represented inside network messages. Many standards have been (and are being) developed for storing biomedical information in files, and transmitting it over a network. Standards are needed for application programming interfaces (API) for access to biomedical information from within running programs.

**Encoding** The choice, for each data element, of how it will be represented in context in machine readable form is called an “encoding”. Each encoding of information requires that some description exists, specifying the encoding. This can be implicit, i.e., described in technical documents and implemented in the source code of computer programs. It can also be done with metadata, i.e., the encoding and structure are specified in a data description language (which itself is implicitly specified). Another option is to create self-describing data, where the encoding and structuring are at a sufficiently low level that the metadata can be incorporated with the data. The encoding problem has generated considerable effort at standardization as well as innovation. All of these strategies have a place in biomedical informatics.

**Structure** For each kind of data in the various contexts, the structuring or grouping of data elements into larger logical units, and the organization of these logical units in a system needs to be decided. Appropriate structuring of biomedical information depends (among other things) on its intended use. Many ideas are applied here, including tables consisting of records, each with identical structure and encoding, object-oriented styles of structure, with many complex data types, often hierarchically related, logical networks, logical expressions with symbols, and others. The organization of such structures ranges from relational databases to object-oriented databases to semi-structured systems of self-describing data, as well as biomedical ontologies with their own external forms.

In this chapter we present a systematic treatment of all of these dimensions. Depending on the design, the dimensions can be orthogonal (independent) or very dependent on each other. Work in all of these areas is guided by some considerations of ease of use, i.e., is it easy to retrieve the data with a computer program on behalf of a user (through a suitable user interface). However, another important consideration is to facilitate sharing of data between laboratories. For clinical practice, too, sharing of medical record data between different health care providers for an individual patient is important. The goal of a data interchange standard is to decouple the details of representing data in any particular computer program from the details of the storage format of a data source. An interchange standard enables data to be shared between different laboratories, even if they choose to use different local storage schemes. As we shall see, these are sufficiently difficult problems to occupy a significant amount of research effort in biomedical informatics.

In the next section, we address the first dimension, that of data description. Next we present several examples of how to structure patient data, using tags and values, using declarative structures in program code (classes, in the usual object oriented style), and using metadata accessible within a program. We then introduce the idea of tabular data and the relational database model. In all of these approaches, maintaining data quality is important and that is the next topic. In the context of reading and writing files as well as network message processing, streams have to be *parsed*, or transformed from byte sequences to data structures, so the next section deals with ideas about parsing. Finally, the last major topic in this chapter is network message formats and data interchange standards.

\*\*\* be sure to include citations to research later in the chapter

## 2.1 Data Description and Metadata

All biomedical data are composites of a few primitive types, namely characters, numbers (integer and decimal), symbols, and sequences or lists. (Sequences of characters are also called “strings”.) The sequences or lists can of course have sequences or lists as elements, allowing arbitrary complex structure. However, it is convenient to also have facilities for efficiently creating some commonly used complex structures, and most programming languages provide such facilities. These include support for arrays, structured types (also sometimes called “record” types), and yet higher level generalizations, usually called “classes”. Despite the typical programmer’s focus on these primitive types, it will become apparent that most biomedical data are *symbolic* and *structured* rather than numbers and text. Their representation as numbers and strings more often is a consequence of the constraints around building systems in conventional programming environments than as an intrinsic property.

Often a distinction is made between text files and binary files. In fact all files are binary files. The difference is in the interpretation of the bytes. A text file is one in which the bytes are interpreted as codes for characters in some character set, usually ASCII or its 8-bit extension, Latin-1, also known as ISO8859-1. More recently, systems have begun to use UTF-8 character encoding. Many other character sets are in use, mainly to represent text in languages other than English.

In network data packets, too, everything is really binary, and the distinction between text and binary is again the interpretation of bytes as character codes in the case of protocols that are described in terms of text strings. An example of a fundamentally binary protocol is the DICOM protocol for transmitting and receiving medical images and related data (see Chapter 13 for a detailed description of DICOM and how to implement DICOM servers and clients). The most common protocols that use plain text are SMTP (electronic mail), and HTTP (the World Wide Web protocol). Both of these have been extended to support the transmission and reception of non-text data in the form of attachments (mail) or other types of data such as images. Web browsers can also receive and send files containing arbitrary data, which of course leads to security problems if the data consist of an

executable program and the browser is able to execute it.

### 2.1.1 Metadata

Data are meaningful only when accompanied by some interpretation, that defines what the symbols mean. While this may seem obvious, it is easy to overlook in the rush to create repositories of “self-describing data”. For example, what does the following list of numbers mean?

```
5984107
8278719
2214646
8220013
5433362
4274566
...
```

Are they telephone numbers? Perhaps they are patient identification numbers assigned by a hospital. They might also be identification numbers assigned to terms in a terminology system such as the Gene Ontology (see Chapter 7) or the Foundational Model of Anatomy (see Chapter 9). They could equally well be a series of pixel values in a medical image, a microscope image, a packed, encoded representation of expression array data, or a very compact encoding of laboratory test data. Some conventions are needed as well as labeling to clearly identify the meaning.

The description, interpretation or identification of the meaning of a particular set of data can take many forms. The data set above may be accompanied by a written document specifying the meaning and format of the data. Another approach is to include with the data some identifying tags that label the elements, so-called “self-describing” data. Here is how the numbers above might appear using such a scheme:

```
<phone numbers>
  <work>5984107</work>
  <home>8278719</home>
</phone numbers>
<phone numbers>
  <work>2214646</work>
  <home>8220013</home>
</phone numbers>
<phone numbers>
  <work>5433362</work>
</phone numbers>
<phone numbers>
  <cell>4274566</cell>
</phone numbers>
...
```

Now we can see that indeed they are telephone numbers, and they are organized so that numbers belonging to a single person or entity are grouped together. Of course there must be some agreement as to the meaning of the tags, which is still outside the context of the data and tags. One might then say, “We just include schemas or data type definitions that define the meaning of the tags.” Very well, then there must be some external document describing the agreement on the meaning of the data type description language. And so it goes. There is always some boundary beyond which a separate documented agreement about meaning must exist. The ongoing debate about self-describing data is not about whether data should be self-describing (which is impossible), but about where this boundary will be, and what description languages will be created for use inside the boundary (and documented outside the boundary).

There is yet another serious problem. In order for a computer program to process the data, the program itself must recognize the tags, so the meaning at some level is encoded in the program. Using human readable tags can mislead the person unfamiliar with how computer programs work. A computer program does not understand the meaning of the phrase “phone numbers”. It can be written to take some particular action when it encounters that text string as a piece of data, but the way it recognizes the tag is by comparing with some other information containing the tag. So, a computer program is also in a sense a data description. The designer(s) of such systems and data repositories will have to choose how this matter is addressed.

The data that describe the tags, i.e., how their associated data elements should be used, what are the allowable value types, etc. are called “metadata”, i.e., data about data. Similarly, if a record structure is defined without tags, in which the data follow some fixed conventions for format and encoding, as in DNA and protein sequence data, there may still be metadata. In a relational database, the kind of data stored in each field of a record is specified by a statement in the Structured Query Language (SQL).<sup>2</sup> This SQL statement also is a piece of metadata. The distinction is important in designing systems. One can design a system that has a high level of generality by making the system use the metadata as a guide to how to handle the data. This idea makes a database system possible - you write a description of your data in SQL, and the database system interprets the SQL to facilitate entry, storage and retrieval of your data, using the tags you specified.

\*\*\* develop this a little further - show an example where we associate “type” information with tags, and other metadata, relationship of metadata to data structure declarations in programs

The use of structured data types in ordinary programming languages is an even more generalized version of the same thing. However, in this case, we are referring to parts of the program as (meta) data, although normally we think of structure definitions as code. The distinction between code and data in a program is not always as one conventionally learns in introductory C or java programming. The

---

<sup>2</sup>later in this chapter we introduce the Relational Model, relational databases, SQL and related ideas

possibility that a piece of code in a computer program can itself be metadata is very important, as we will see later in this chapter. Another example where pieces of code can be metadata is developed in great detail in Chapter 13.

\*\*\* the following is opaque to the beginner at programming - fix it up, explain functional abstraction

A program that is highly modular typically uses lots of procedures from libraries. These libraries can be viewed as analogous to the rules of a more declarative programming style, and the “main” program is then analogous to a rule interpreter. Alternately one could imagine the subroutine libraries as something like a database of code fragments. Alon Halevy observed that a program would then correspond to a “view” on a database.<sup>3</sup>

Taking this to the extreme, some of the early single chip CPU designs did not include much of the instruction set of their minicomputer predecessors. The unimplemented instructions would trigger calls to very low level subroutine libraries, so that much of the compiled code consisted of such calls. This was called “threaded code”, since the CPU was threading through a series of subroutine calls, rather than executing in-line code.

\*\*\* maybe a sentence or two about bit slices and microcode

## 2.1.2 Biomedical Encoding Examples

In order for computer programs to deal with biomedical data, the data must be *encoded* according to some plan, so that the binary numbers in the computer’s memory represent numeric data, or represent text, character by character, or possibly more abstract kinds of data. Many encoding schemes have been used for biomedical data. We describe a few here, to give a flavor for how this might work. We also show how small programs can express or embody the encodings, and in some cases convert from one encoding scheme to another. Examples include biomolecular sequence data (DNA and proteins), laboratory data from tests done on blood samples and other substances obtained from patients in a clinic or hospital, and electronic medical record data.

### Biomolecular Sequence Data

To illustrate how encoding can make things more or less transparent, consider a gene, a sequence of nucleotides, as described earlier in this chapter. A typical representation in text would consist of a sequence of letters, using only the four letters G, C, A, and T, to represent each of the possible four nucleotide (also called “base”) pairs that could appear in a double helix DNA strand. Although the DNA molecule is double stranded, the bases are paired uniquely, A with T and G with C, so that only the bases on one strand need to be represented.

---

<sup>3</sup>Alon said this to me after a talk at the University of Washington Computer Science Colloquium by Gregor Kiczales on “Aspect-Oriented Programming”.

A gene is a subsequence that can be translated into a protein. Proteins are also large molecules made up of long sequences of small units. In proteins, the small units are a set of 20 different molecules called *amino acids*. As with the nucleotides, each amino acid can be symbolized by a single letter, so that a protein also can be represented by a letter sequence. This sequence is called its *primary structure*. Proteins have many different roles in cells and tissues. They can serve as enzymes to facilitate biochemical reactions, such as metabolism of glucose. They can also regulate other processes by serving as signals. Proteins also are components of cellular and tissue structure.

The relation between genes, DNA and proteins is called the “Genetic Code”. Each amino acid corresponds to one or more patterns of three nucleotides. For example, the nucleotide sequence “GGA” corresponds to the amino acid glycine, and “TTC” corresponds to the amino acid phenylalanine. Each combination of three nucleotides is called a codon. With four possible nucleotides in three places, there are 64 (4X4X4) codons altogether. Not all codons correspond to amino acids; there are three special codons that signal the end of a sequence, TAA, TAG and TGA. For most of the amino acids, there are several codons that represent the same amino acid. For example, the amino acid Lysine is represented by two codons, AAA and AAG. Table 2.1 shows the complete correspondence between codons and amino acids. This correspondence is the same in all known organisms.

An entire genome of a human is available from the NCBI database, GENBANK, as are a number of other animal and plant sequences. Figure 2.1 shows a small fragment of the region around a well known gene associated with breast cancer, called BRCA1.<sup>4</sup>

```

CACTGGCATGATCAGGACTCACTGCAGCCTTGACTCCCAGGCTCAGTAGATCCTCCTACCTCAGCCTCTC
GAGTAACTGGGACCACAGGCGAGCATCACCATGCTCAGCTAGTTTTTGTATTTGTAGAGATGAGGTTTCA
CCATATTGCCAGGCTGGTCTTGAATCCTGGGCTCAAGCAAGCCACCCACCTTGGCCACCCAAAGTGCT

```

Figure 2.1: An excerpt of a DNA sequence from the BRCA1 gene

In a typical computerized encoding, each letter is represented by its ASCII code so each occupies 8 bits (although ASCII is a 7 bit code it is usual to use 8 bit “bytes”). Allowing for letters that represent alternative bases (e.g. for pattern matching), e.g. where a position might hold either purine base, or any except A, or similar such combinations, we can represent base sequences as letter sequences, or long strings. However, when doing pattern matching, it is a bit cumbersome to match letters, since a letter might represent more than one possible base. The bases are of two types, the *purines* (A and G) and the *pyrimidines* (C and T), and sometimes a location is known to have only a purine, or only a pyrimidine (e.g., R, meaning either purine, A or G). In such a case, it should match not only with itself, but with any of the single bases. Similarly, the other combinations of two of the four bases can be characterised according to other properties they have in common, and letter codes are reserved to represent each of those. To simplify such code, we

add some notes on ASCII, UTF, etc. and the fact that these are also binary numbers, but different ones than one might use below

<sup>4</sup>Obtained from NCBI Genbank web site, contig NT\_010755.15 starting at position 4859744

Table 2.1: The Genetic Code by Amino Acid

Letter	Abbrev	Full Name	Codons
A	Ala	Alanine	GCA GCC GCG GCT
C	Cys	Cysteine	TGC TGT
D	Asp	Aspartic acid	GAC GAT
E	Glu	Glutamic acid	GAA GAG
F	Phe	Phenylalanine	TTC TTT
G	Gly	Glycine	GGA GGC GGG GGT
H	His	Histidine	CAC CAT
I	Ile	Isoleucine	ATA ATC ATT
K	Lys	Lysine	AAA AAG
L	Leu	Leucine	TAA TTG CTA CTC CTG CTT
M	Met	Methionine	ATG
N	Asn	Asparagine	AAC AAT
P	Pro	Proline	CCA CCC CCG CCT
Q	Glu	Glutamine	CAA CAG
R	Arg	Arginine	AGA AGG CGA CGC CGG CGT
S	Ser	Serine	AGC AGT TCA TCC TCG TCT
T	Thr	Threonine	ACA ACC ACG ACT
V	Val	Valine	GTA GTC GTG GTT
W	Trp	Tryptophan	TGG
Y	Tyr	Tyrosine	TAC TAT

use a binary representation as an alternate to the character representation. With the right binary representation we can use logical operations on bit patterns, which will be fast as well as simple to express in code.

The encoding we will present and the code that implements it are based on Alberto Riva's CL-DNA project [107]. Rather than represent each base as a two bit code, we allocate one bit within a byte to encode for each base. Thus, one byte corresponds to a position in a DNA sequence. If the value of the byte is 8 (1000 in binary), that stands for A, and similarly, 4 (0100 in binary) for C, 2 (0010 in binary) for G and 1 (0001 in binary) for T. This allows for encoding a position as having an ambiguous value, i.e., if the byte value is 10 (1010 in binary), the position could hold either A or G. The value 15 (1111 in binary) corresponds to "any base". This is useful for representing and solving pattern matching problems. For example, 15 can represent a possible gap in a DNA sequence. In addition to the binary encoding, we will use a text representation in which the bases and their combinations are denoted as characters, as shown in Table 2.2.

To turn these ideas into code, we will use the Lisp programming language (see the discussion in the Preface, page xii, for why Lisp is the language to use). Chapter 1 provides a very brief overview of the essentials of Common Lisp, and how to use it interactively. The references in Appendix A.1 provide a more thorough introduction.

Table 2.2: Character symbols for bases and combinations, shown with names and binary (4-bit) codes

Character Code	Bases	Description	Binary Code
X	()	Stop	0000
T	(T)	Thymine	0001
G	(G)	Guanine	0010
K	(G T)	Keto	0011
C	(C)	Cytosine	0100
Y	(C T)	Pyrimidine	0101
S	(C G)	Strong interaction	0110
B	(C G T)	Not-A	0111
A	(A)	Adenine	1000
W	(A T)	Weak interaction	1001
R	(A G)	Purine	1010
D	(A G T)	Not-C	1011
M	(A C)	Amino	1100
H	(A C T)	Not-G	1101
V	(A C G)	Not-T	1110
N	(A C G T)	Any	1111

Writing a program involves writing expressions that define named constants, variables and functions. These expressions can in turn use other constants, variables and functions that have already been (or will be) defined. If a function is used, but not defined here, it is a built-in function in the Common Lisp standard environment.<sup>5</sup> In accordance with good design, we define relatively simple functions, from which we can then build more complex ones. Once a function is defined, it can then be used in the interactive Lisp environment.

In the following, keep in mind that arrays are sequences indexed by integers. Each element in the sequence can be any Lisp data, including another sequence. A string is just a one dimensional array (also called a **vector**) whose elements are a subtype of the **character** type. The type **sequence** includes both vectors and lists. Some functions can be applied to both, while others only apply to one type or the other. The two types, **vector** and **list** are disjoint types (even though in some Common Lisp implementations, arrays may be implemented as nested lists).

The characters and codes in Table 2.2 can be very compactly represented by the string defined as the value of the global **\*base-chars\***. This definition enables us to use the variable **\*base-chars\*** as a table for converting from the character representation to the binary representation and back. We will write small functions

---

<sup>5</sup>Graham [41] includes an appendix with brief descriptions of all the ANSI standard built in functions, macros etc. as of the writing of that book. Steele [127] has much more complete descriptions and explanations, but is not as up to date. I use both.



that do this lookup on demand.

```
(defconstant *base-chars* "XTGKCYSBAWRDMHVN")
```

Thinking of this string, a sequence of characters, as an array, the index (or position number) of each character is (in binary) exactly the corresponding code. For example, the fifth character in the sequence is “C”, which is also position number 4 (in Lisp, as in several other popular programming languages, the first element of an array or list has index 0). So that exactly matches the table entry for “C”, whose binary code is 0100.

In many genomes, one can find short base sequences that appear many times. These are called “repeats” or repeat sequences. Some of these have significant biological function, so it is useful to be able to search for the occurrence of such patterns. Following Riva, we also define bit 5 in each (binary) byte code to indicate that the base is part of a repeating element. Bases that are part of identified repeats are represented in the character representation by *lower case* letters, rather than upper case as shown in Table 2.2.

```
(defconstant +repeat+ 16)
```

Bits 6-8 are unused, so the binary encoding is one byte per base. The position of each character in the string **\*base-chars\*** is the binary code for the base or base combination that the character represents. This string provides a simple way to convert from one representation to another.

To convert a character (actually ASCII or other code for a character) to a binary code, just look up its position in the character array (or string) **\*base-chars\***. The **encode-base** function processes a single base character, and the **encode-bases** function processes a list of characters, e.g. for the case of encoding the multi-base cases described above. Note that a *list* of characters is not a string. Later, we will present functions for handling base sequences in both encoded and string formats.

```
(defun encode-base (base-char)
  "Turn a base character into a 4-bit representation."
  (let ((b (position (char-upcase base-char) *base-chars*)))
    (if b
        (if (lower-case-p base-char)
            (+ b +repeat+)
            b)
        0)))
```

The expression above, when evaluated by the Lisp interpreter, has the effect of putting a new function definition in the working environment. The new function will be named **encode-base**. When used it will need one input parameter, here called **base-char**, which is expected to be a single character (an actual character, not a one character string). The quoted string in the function definition is a documentation string, and it can be retrieved by another function within a program if desired.

The **let** expression creates a local variable named **b**, and gives it the initial value computed by the **position** function (a built-in Lisp function that takes a value,

any type of Lisp datum, and a sequence as inputs, and looks for the value in the sequence - if found, the result of the function is the position, using zero-indexing, but if the value is not found in the sequence, the function gives a result of `nil`).

The remaining expression is the body of the `let` operator, and it is evaluated using the value assigned to `b`. The `if` operator takes three inputs, an expression that is used as a logical test, a result to return if the test is successful, and a result to return if the test returns `nil`. In this case, if `b` is not `nil`, i.e., the character was found and the value of `b` is its position in `*base-chars*`, another `if` expression checks if it is lower case. If the character was lower case, the value of `+repeat+` is added to `b` so that it is flagged in the binary representation as part of a repeat sequence, otherwise the result of the `if` is just the value of `b`, i.e., the index of the character in the array of characters, or exactly the right binary number corresponding to the input character. The value of the `if` expression is then also the value of the `let` expression, which is then also the value the function returns as its result. Here are some example of the function in use.

```
> (encode-base #\G)
2
> (encode-base #\g)
18
> (encode-base #\R)
10
>
```

The Lisp interpreter generally prints binary numbers as their decimal integer equivalent, as shown here.

```
(defun encode-bases (base-chars)
  "Turn a list of base chars into a 4-bit representation."
  (reduce #'logior (mapcar #'encode-base base-chars))))
```

The function `encode-bases` takes as input a list of one or more bases, up to all four, and returns the binary code for the combination. This handles the possibility that we might want to construct the list of possible bases in a position in a gene, and then convert that list to a binary code. Here we see several powerful built in Lisp functions that can be used to do complex tasks with simple code. The `mapcar` function takes as input a function (in this case the function `encode-base` that we just defined), and a list, here called `base-chars`, and applies that function to each element of the list, giving a new list as the result. Each element of the new list is the result of applying the input function to each element of the input list. The following examples show how that expression works.

```
> (mapcar #'encode-base '(#\G #\C))
(2 4)
> (mapcar #'encode-base '(#\g #\A #\T))
(18 8 1)
> (mapcar #'encode-base '(#\R))
```

```
(10)
>
```

Then, the `reduce` function applies its first argument (in this case, the function `logior`, which computes the logical or of its binary number inputs) successively to the first two elements of the list which is its second argument, and then that result with the next element and so on until the end of the list, i.e., in this case it computes the logical “or” of all the binary numbers, to get the right binary number for the list of bases. Logical “or” is used instead of adding the numbers because adding the numbers could count a base more than once. As an example work by hand the conversion of the list (`#\R #\G #\C`). Adding the individual codes would give 16, which is wrong, but logical “or” would give 14, which is correct.

The `mapcar` function is a simple way to take a list of data and apply a function to every data element to get a new list with the results in it. Using `reduce` with `mapcar` is a nice idiom for transforming a list of data and then combining all the results. The single line of code above does the work that would otherwise involve writing iterations over the list, keeping temporary values, and so on. This style is the style of functional programming, where a lot of processing is done by just composing functions together rather than writing loop code, assigning values to local variables, and having to keep track of when you are done.

Now we are ready to write the reverse functions, that map binary codes into characters. To convert a binary code to a character, just use the code as an index into the same string, since a string is an array of characters. The character at that position is the right character for the binary code. We will need one additional function, `base-masked-p`, to determine if the base was marked as a repeat, in which case, we then change the character to lower case.

```
(defun base-masked-p (code)
  "Returns T if the base was marked as part of a repeat."
  (plusp (logand code +repeat+)))
```

Here we use a logical “and” (the `logand` function) to check if the repeat bit is set. If the bit is set, the `plusp` function will return `t` for “success” or “true”, otherwise it will return `nil` for “false”.

Then, to decode a byte into its character representation, we use the lower 4 bits as the index into the base string, and change the result to lower case if repeats are being checked. This is specified by the optional input, named `repeats` here. In Common Lisp function definitions, input parameters that are optional are preceded by a special tag, `&optional`. The `aref` function is used to get the array element in `*base-chars*` at the position corresponding to the lower 4 bits of `code` (the result of logical “and” of `code` and the number 15, which is binary 1111).

```
(defun decode-base (code &optional repeats)
  "Return the character corresponding to the encoded base,
   e.g., (DECODE-BASE 7) => #\B"
  (let ((ch (aref *base-chars* (logand code 15))))
    (if (and repeats (base-masked-p code))
```

```
(char-downcase ch)
ch)))
```

The function `decode-base` returns the single letter corresponding to the combination of bits in the case of a composite value. Sometimes it is useful to have a list of the individual bases that are specified by the encoding. This is provided by `decode-bases`. Here we have to create a list and add to it the base character for each base that is present according to the binary code that is the input of `decode-bases`. Note that it does not handle bases that are part of a repeat. Adding that capability is left as an exercise.

```
(defun decode-bases (code)
  "Return a list of characters corresponding to the bases
   that are 1 in code. E.g., (DECODE-BASES 5) => (#\T #\C)"
  (declare (type base code))
  (let ((result nil))
    (when (plusp (logand code 1))
      (push #\T result))
    (when (plusp (logand code 2))
      (push #\G result))
    (when (plusp (logand code 4))
      (push #\C result))
    (when (plusp (logand code 8))
      (push #\A result))
    result))
```

A few other useful functions include `base-complement`, which returns the complement of its argument, `base-match`, which returns T if its two arguments have at least one element in common, i.e., for combinations of bases, the codes both have at least one bit in the same location, and `base-match-a`, which is similar to `base-match` except that it returns false if its second argument is 15, corresponding to N, any base. This last is useful when an N represents a gap, which should not match anything. In addition we define one more predicate, `single-base-p`, which returns T if its argument encodes a single base, rather than a combination. Note that `base-complement` works from a lookup table similarly to the encoding and decoding functions. The `make-array` function constructs an array with the specified initial contents, arranged in a complementary pattern to the array `*base-chars*`, e.g., in the position where `*base-chars*` has A, `*base-complements*` has the code for T, the complement of A.

```
(defconstant *base-complements*
  (make-array '(16)
    :initial-contents
    '(0      ; x
      8      ; a
      4      ; c
      12     ; a c
```

```

2      ; g
10     ; a g
6      ; c g
14     ; c g a
1      ; t
9      ; t a
5      ; t c
13     ; t c a
3      ; t g
11     ; t g a
7      ; t g c
15)    ; t g c a))

```

```

(defun base-complement (base)
  "Return the complement of a base, in the biological sense,
   using codes, e.g., (BASE-COMPLEMENT 1) => 8."
  (aref *base-complements* (logand base 15)))

(defun base-match (b1 b2)
  "Return T if b1 and b2 match (i.e., if they are the same base, or
   if one represents a set of bases that includes the other."
  (plusp (logand b1 b2 15)))

(defun base-match-a (b1 b2)
  "Same as above, except that it fails when b2 is an N (to prevent
   gaps in sequences from matching everything)."
  (unless (= b2 15)
    (plusp (logand b1 b2 15))))

(defun single-base-p (bits)
  "Return T if bits indicates a single base."
  (or (= bits 1)
      (= bits 2)
      (= bits 4)
      (= bits 8)))

```

Now that we have a representation and encoding for bases and sequences, what will we do with the data? Many commonly used operations on DNA sequences can be implemented, as described in much more detail in bioinformatics texts [8]. A simple example is searching for a known short sequence in a larger sequence: `seq-find` looks for the first exact match.

```

(defun seq-find (sub seq &optional (start 0))
  "Find the position of 'sub' in 'seq' starting at 'start'.
   Exact matches only."
  (declare (type (simple-array base (*)) sub seq)

```

```

      (fixnum start))
  (search sub seq :start2 start :test #'base-match-a))

```

This definition uses the built-in Common Lisp `search` function, which looks for a sequence within another sequence, using a specified test. This is not the same as the graph searching idea mentioned in Chapter 1 and developed further in Chapter 3.

The optional parameter to `seq-find` makes it possible to search for and accumulate all the exact matches, by starting just after each match as it is found. The `seq-nfind` function does just this, and returns a list of *all* exact matches.

```

(defun seq-nfind (sub seq)
  "Find all occurrences of 'sub' in 'seq', using SEQ-FIND."
  (declare (type (simple-array base (*)) sub seq))
  (let ((result nil)
        (pos -1))
    (declare (fixnum pos))
    (loop
      (setq pos (seq-find sub seq (1+ pos)))
      (if pos
          (push pos result)
          (return)))
    (nreverse result)))

```

Searching while allowing mismatches, insertions or deletions is much more complicated. A wide range of software tools are available to do this. A good survey of such tools is the book by Baxevanis, et. al.[8]. We present only a small sampling of these operations in Chapter 7.

\*\*\* put here stuff about FASTA files and web access

\*\*\* add here Alberto's protein encoding also, and then maybe a reference to Larry Hunter's code to search for names in protein sequences

\*\*\* maybe also put here SWISS-PROT samples as examples of tagged data, BioDBloader as example of processing such files, point being that the structure of the files does not need to be predefined or uniform.

## Microarrays

\*\*\* get some data and ideas from Mark Whipple

URL for gene expression array data: <http://www.ncbi.nlm.nih.gov/geo/>

## Medical Laboratory Data

Gene and protein sequence data use a small “alphabet” of symbols, four for elements of gene sequences and 20 for the amino acids that make up proteins. This encoding problem is simple by comparison with the problem of representing clinical laboratory test data. Laboratory data are the results of tests that the clinical laboratory performs when a sample of a patient’s blood is drawn from a vein, and put in a tube or tubes to be sent to the laboratory. Some examples of blood tests are: red and white blood cell counts, hemoglobin, creatinine level, glucose level, etc.

need help from Dave Chou here

Many other kinds of samples of body tissues also can be used to do tests, including urine and bits of material obtained by swabbing your throat, ear, nose, eye, etc. There are thousands of types of laboratory tests, each requiring its own identifier or code, as well as its own conventions for representing the test result itself. Tests can include determining what chemical substances are present in the sample, and inoculating agar gel containers (called “cultures”) to determine if bacteria are present in the sample.

A well known scheme for labeling and encoding such test results is the Logical Observation Identifier Names and Codes (LOINC)<sup>6</sup> system [84]. The goal of LOINC is to assign unique codes and names to each of the thousands of observations and laboratory tests in use in clinical practice, so that the associated data for a particular patient can be labeled in a computerized message transmitting the data between various electronic medical record (EMR) systems and other application programs using medical data. Similarly, these same codes can be used by an electronic medical test order entry system to identify what a care provider has ordered for a patient.

The LOINC system proposes a set of names and codes for labeling laboratory results and clinical findings. This involves achieving agreement about the type of a datum, its external representation, the units in which it is measured, and other factors that affect the interpretation of the result being reported.

...LOINC example here

\*\*\* pointer to new chapter on Data Qyality?

Information such as pathology and radiology reports often take the form of dictated text. Although highly stylised, such reports are difficult to represent in as structured a form as LOINC. As a result, considerable research has been conducted to devise methods of automatically extracting structured data from such reports, using natural language processing techniques.. Many other places exist in the EMR for such “free text”.

cite Ricky Taira and others here

\*\*\* add DICOM and some other examples here

---

<sup>6</sup> “LOINC” is a registered trademark of Regenstreif Institute, Inc.

## Electronic Medical Record Systems

\*\*\* get example data from PTH, Tom Payne or Jim Hoath maybe

\*\*\* Discuss the “curley braces” problem here, perhaps.

\*\*\* pointer to new chapter on Data Qyality?

\*\*\* Discuss comments made by Don Lindberg (NLM Director) at the NLM Training Program Directors meeting, July 2005 – where does this belong?

Electronic health record – should be a life long record, including

- genome analysis
- family relationships (pedigree) (PSZ work on pedigree analysis)
- ethical conventions
- patient’s own observations

Need to also provide way of anonymizing for research use.

\*\*\* also, Lindberg comments on importance of “molecular libraries”

## 2.2 Patient Data: An Illustration

The code we develop in this section will show how to store (in files) and represent (in a program) patient data that might be found in a typical electronic medical record system. The examples use text files for external storage, and tags (also called “keywords”) to label the data items. The patient example developed here is simple and unconventional, but it illustrates some ideas that can help to manage complexity. Many electronic medical record (EMR) systems are built on relational database technology. The conventional wisdom is that a relational database is currently the most efficient, scalable architecture. However, the complexity of medical data suggests that the relational model may not be the best fit. The challenge is to be able to store and retrieve heterogeneous collections of data that have highly variable structure. Although the implementations we show here might be hard to scale up, they illustrate important ideas which can guide a more scalable design. Later in this chapter, we will also examine the relational model to see how it compares with these methods.

Three methods will be developed, to show how structured data may be encoded and manipulated. The first is a simple arrangement of tagged data, where the structure of a complex record is represented by hierarchical grouping of items in nested lists. The second method, using standard object-oriented programming ideas, shows



how the *class definition* facilities of typical object-oriented programming languages provides the abstraction layers that had to be done explicitly in the first method. The third method, using *metaobjects*, shows how you can use the implementation of the object-oriented system itself to write even more abstract (and therefore more powerful and general) procedures and programs.

In this chapter we are representing and manipulating *data* but the structures that describe the data could well be argued to be a representation of some *knowledge* about the data. Later in this chapter (Section 2.6.3), and in Chapter 3, we will introduce the syntax and semantics of XML, RDF and OWL, which provide methods and vocabulary for tagging data and representing structure and some forms of knowledge in a standard way. These are important because, like the examples we develop in the next sections, they use plain text (e.g. ASCII) and thus are a means for encoding data and knowledge for export and import from one system to another. Further, the syntax of XML has become very popular, so there is now a large code base of programs that implement XML generators and parsers. We will illustrate the use of such code to store and retrieve information from such external formats.

### 2.2.1 A Simple Solution Using Tags (Keywords)

Patient data are a good match to hierarchical representation, where structures include elements which themselves have structure. Using symbols, strings, numbers and lists, we can organize information about a patient in terms of keyword-value pairs. (Here, we are using “keyword” to mean the tag that labels a datum, which could itself be structured as a list of tagged elements, and so on.) Here is an example of an information record on a single patient.

```
(patient (name "John Q. Smith")
  (hospital-id "123-45-6789")
  (age 62)
  (address (number 211)
    (street "Willow Street")
    (city "Browntown")
    (zip-code 90210))
  (diagnosis (name "prostate cancer")
    (psa (("23-Aug-2003" 4.6)
      ("12-Mar-2004" 7.2))))
  ...)
```

As explained in Chapter 1, text enclosed by double-quote (") characters forms a *string* data type. The unquoted text items are symbols and numbers. Numbers are represented here in the usual way. Parentheses delimit lists, which may consist of an arbitrary number of any kind of elements, including other lists. We represent keyword-value pairs as lists of length 2. Parentheses provide an unambiguous way to group related items, and no other special syntax is needed.

This keyword-value representation is easy for a person to read, especially with the indentations. The indentations and line breaks have no impact on the logical

structure. Neither does the amount of space between items (though at least one space character is needed, it does not matter how many there are). The extra “whitespace” is simply to make the data more easily human-readable.

This idea of grouping items and providing tags is such a powerful way to organize information that an internationally popular (and rather more complicated) syntax called XML was invented to support the idea. We will deal with it later in the chapter. For now, parentheses and symbols will suffice. For this simple example, we represent all the data as text in a file consisting of nothing but printable characters.

If the data above appeared in a text file, a single call to the Common Lisp `read` function would return a list structure exactly as shown, which could be examined by a computer program, by standard list operations, e.g., `first` and `rest`. Rather than having to define data structures and access methods, and then decode and parse the input to assign the values to the right places in the data structures, the above text representation is already a valid representation of a Lisp data structure.

This format takes advantage of the fact that the Lisp reader and printer can do some of the work for us. All elementary Lisp data types have well defined external or printed representations, even including user defined structured types (defined with `defstruct`) and arrays. For reading and writing these simple data types from/to files, we can just use `read` and `print`. We don’t need to do any lexical analysis of the text on input or pay much attention to formatting on output (though some formatting would enhance human readability of the data).

add some notes about pretty-printing, also about the fact that data and code use the same representation

However, writing a program to manipulate the above structure could get tedious. Some work would be involved in creating abstraction layers, so that the code that works with data elements goes through accessor functions, e.g., to get the diagnosis, we would define an accessor, named `diagnosis`:

```
(defun diagnosis (patient-data)
  (find 'diagnosis (rest patient-data) :key #'first))
```

and similarly for each of the other fields and subfields. Like the SWISS-PROT files and other keyword labeled records, this scheme has the advantage that records do not need to have uniform structure, and so the structure does not need to be predefined. This is workable if we impose some restrictions on the structure of the data, i.e., that certain items are always found at the top level of the list of data for the patient. Additional assumptions or restrictions must be made about items that themselves have structure.

\*\*\* Expand with example, perhaps tumor registry or lab data, to be revisited in the DB section, do size estimate for institution, state and national

Changing the structure of the data, perhaps by adding another field or subfield, could require changing many of the accessors.

\*\*\* other complications?

### 2.2.2 An Object-Oriented Design With Classes

In using a simple scheme with tags, we had to write lots of accessor functions and also keep in mind some definition of the structure of the patient record. This work will be required for each kind of structure we introduce into our medical record system. An object-oriented programming language provides a standardized collection of syntax and semantics that make this work much easier. We can write definitional expressions that specify the structure and the names of the accessors, and the object-oriented programming environment automatically generates the support code.

In effect we are creating an abstract model of medical information about patients, and letting the modeling system translate the formal model into lower level code. When the model changes, the program need not change except to accomodate the specific additions or deletions. Aggregated data units become, in a typical object-oriented programming language, instances of classes, whose definitions represent the model implemented in an actual program. Figure 2.2 is an example of a part of an object-oriented model for patient information. The figure uses a notation called Entity-Relationship (ER) Modeling. Each box in the figure represents a *class* specification, including the class name and the names of the attributes of that class. Each instance of a class can have different values assigned to the attributes. The lines connecting the boxes represent relationships between classes, indicating that one class is a subclass of another, or that an attribute of one class can have values which are instances of another class.

The connections between the boxes represent the typical kinds of relationships found in object-oriented models. A more thorough discussion of object-oriented design may be found in [115].

\*\*\* still need more here - describe the notation and its meaning

The model can be implemented with a set of class definitions, in the Common Lisp Object System (described in Chapter 1, and in the standard references). For each box we have a class, and we implement families of classes using inheritance. Keeping in mind that a type (class) may be defined directly as a subtype (subclass, or specialization) of a parent type (or class). Thus, in the patient data model, the **patient** entity and the **provider** entity can each be defined as a subclass of a more general entity, **person**. As a first try, we define a structure type for each entity.

```
(defstruct person ())
  name birthdate telephone email ...)
```

```
(defstruct patient (person)
  diagnosis appointments ...)
```

```
(defstruct provider (person)
  specialty title office patients ...)
```

Specifying that **patient** and **provider** are subtypes of **person** will automatically make any slot defined in the **person** type also be present in both the **patient**

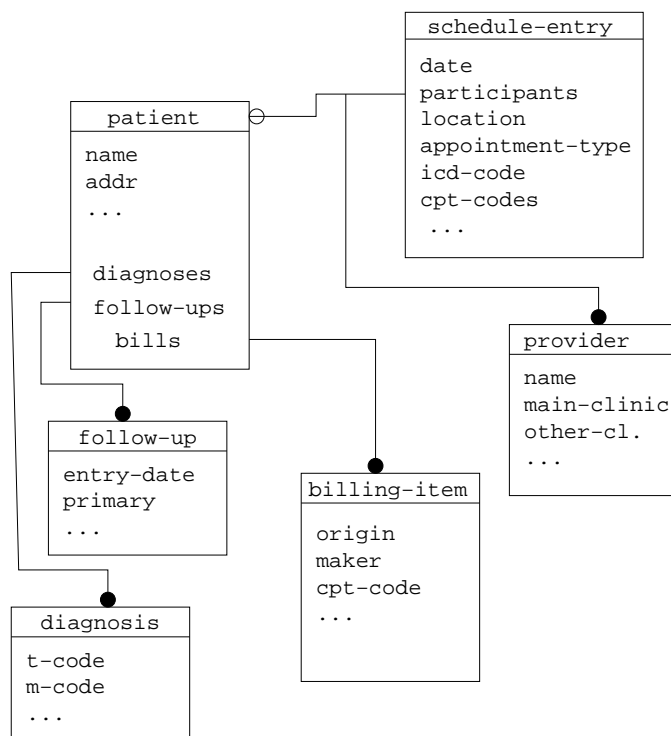


Figure 2.2: An Entity-Relationship model for a subset of patient data

type and the **provider** type. This is one aspect of inheritance. These specializations also add their own slots to the inherited ones. Similarly any accessors for **person** are inherited and applicable to both subtypes.

\*\*\* Need example here - call **make-provider**, create schedule entry, then send confirming email to provider, use the accessor, **person-email**, not **provider-email**, which is useful when iterating over the participant slot, since there are multiple types of participant, all subtypes of **person**.

Inheritance in relation to the type/subtype relationship provides a way to build a program system incrementally and reliably. Adding the subtypes takes advantage of all the support code that is already written for the parent types, and does not in any way impact or require revision of code that supports the parent types. This “code reuse” has been one of the most attractive aspects of “object-oriented programming”.

The idea is not without its problems and limitations. In an electronic medical record system that supports appointment scheduling and billing as well as recording clinical data, the **provider** class might have a slot for a list of **patient** instances, the provider’s patients. The model is complex because a patient typically has a

relation to multiple providers, and vice versa. In principle, one only need represent one relation and the other can be computed, but this would be very inefficient for a large hospital or clinic. For efficiency, redundant references should be maintained in the system. However, this makes updating entries more complex, because the redundant information has to be kept consistent. For example, when a patient sees a new provider, multiple modifications are made, to the patient record, and to the provider record. Similarly, when a relationship ends, the relevant information should be removed everywhere, so the system has to have code to do the removal in all the redundant locations.

Managing this kind of complexity is a big challenge for system designers, as it requires the implementation of consistency constraints, and results in a system that is no longer made of relatively independent parts. Designing such systems involves considering the tradeoffs among simplicity, reliability, efficiency and ease of use.

The main differences between structures and classes are that structures provide an efficient but restricted system for defining slots, accessors for slots (functions that return and set the slot values) and other properties of each structure. Classes provide a wider range of possibilities. In particular, the accessor for a slot is named uniquely for each structured type, even if the slot has the same or a related meaning for different structures. We may have several kinds of things in our program, each part of a human body, and each is a structure, e.g., heart, kidney, tumor, etc. Each will have a location (of its center, perhaps), specified by coordinates  $x$ ,  $y$ , and  $z$ .

```
(defstruct heart ()
  size beat-rate x y z ...)

(defstruct kidney ()
  side x y z ...)

(defstruct tumor ()
  size grade tissue-type x y z)
```

The accessors, however will be named differently, i.e. `heart-x`, `kidney-x`, `tumor-x`, because the storage arrangement for each may differ and there has to be some way to distinguish what to do in each case. So, you have to write different code for each type, in order to get and store the data. In Common Lisp, *classes* provide a nicer way to handle this problem.

\*\*\* Specific ref here to explanation of classes and `defclass` in Chapter 1.

The accessors of slots in a class defined by `defclass` are not provided automatically as for structure definitions, but these accessors can be *generic* functions, i.e., they can share a common name, but have different methods depending on the type of the object. This is an example of *type-based dispatch*. Here is a rewrite with classes instead of structures. Note that the slots need some additional specification in order to have the accessors created.

```
(defclass heart ()
```

```

((size :accessor size)
 (beat-rate :accessor beat-rate)
 (x :accessor x)
 (y :accessor y)
 (z :accessor z)
 ...))

(defclass kidney ()
  ((side :accessor side)
   (x :accessor x)
   (y :accessor y)
   (z :accessor z)
   ...))

(defclass tumor ()
  ((size :accessor size)
   (grade :accessor grade)
   (tissue-type :accessor tissue-type)
   (x :accessor x)
   (y :accessor y)
   (z :accessor z)
   ...))

```

In this case, a single *generic* function, `x`, can retrieve the  $x$  coordinate of any instance of these objects. Just as `defstruct` automatically created accessor functions for each structured type, `defclass` will create *methods* for the generic functions named as the accessors. However, the names can be shared among classes. Generic functions are described in Chapter 1.

### The Patient Class and Related Code

Using CLOS, as in any object-oriented programming language, we define a **patient** class, with slots that will hold the individual attributes for each patient that is an instance of this class. Here, we follow the ER diagram, Figure 2.2, though it would be a better design to include the **person** class and implement **patient** and **provider** as subclasses of **person**.

```

(defclass patient ()
  ((name :accessor name :initarg :name)
   (hospital-id :accessor hospital-id :initarg :hospital-id)
   (age :accessor age :initarg :age)
   (address :accessor address :initarg :address)
   (diagnosis :accessor diagnosis :initarg :diagnosis)
   ...)
)

```

Note that the `address` and `diagnosis` slots will contain instances of structured objects. We define classes for them too. These kinds of objects will appear in other parts of the medical record too.

```
(defclass address ()
  ((number :accessor number :initarg :number)
   (street :accessor street :initarg :street)
   (city :accessor city :initarg :city)
   (zip-code :accessor zip-code :initarg :zip-code)
  )
)
```

In defining the diagnosis class there is a problem. What slots are to be included in the diagnosis class will depend on what the diagnosis is, i.e., different diagnoses will require different laboratory and other findings to be recorded. So the following naive translation of our keyword-value pair representation is not adequate. We will develop a more comprehensive approach to this problem later.

```
(defclass diagnosis ()
  ((name :accessor name :initarg :name)
   (psa :accessor psa :initarg :psa)
   ...)
)
```

We would like as before to store the information on a patient or series of patients (instances of the `patient` class), in a file, in a “serialized” form. By reading the file appropriately we should be able to reconstruct an object that is equivalent to the original instance in our program.

The `read` and `print` functions will not work for data that are instances of classes defined by `defclass`. The Common Lisp `print` function does not print an instance of a class in a way that it can be reconstructed.

We could make an instance of this class and try to print it.

```
> (setq temp
    (make-instance 'patient :name "Ira K"
                   :hospital-id "99-2345-6" :age 60))
#<PATIENT @ #x7161b75a>
> (print temp)
#<PATIENT @ #x7161b75a>
```

This instance of the patient class is printed as an unreadable object (the `#<` reader macro denotes an unreadable object, followed by the class name and some implementation dependent information). We will need to provide some extension to support serialization of class instances, including classes where slot values are themselves class instances.

A seemingly elegant solution is to use the CLOS facilities themselves to extend the `print` function’s capabilities. According to Steele [127, page 850], the printing system (including `format`) uses the standard generic function, `print-object`. We

can therefore write specialized methods for it, corresponding to each class we define in our program, so that when instances of those classes are printed using any print function, they will be handled by the specialized method, and not the system generic method.

This is analogous to operator overloading in C++.

\*\*\* (to be expanded to reference `java` too). Also reference marshalling and unmarshalling in `java`.

To print something readably, we start by defining a method for `print-object` when the object is a patient.

```
(defmethod print-object ((obj patient) strm)
  (format strm "(Patient ~%")
    (format strm "  :name ~S~%" (name obj))
    (format strm "  :hospital-id ~S~%" (hospital-id obj))
    (format strm "  ...etc. ) ~%" ...))
```

Now when we try to print our instance, it will use our specialized method instead of the system default.

```
>(print temp)
(Patient
  :NAME "Ira K"
  :HOSPITAL-ID "99-2345-6"
  ...etc. )
>
```

This can also get tedious, verbose and error prone when a program has many different classes. It does have one nice property. The calls to `format` will be implemented through the `print-object` generic function, so if a slot value is another class instance and there is a corresponding `print-object` method, the information will be printed properly nested *logically*. The resulting text file will not be formatted nicely for the human reader, however.

This output can be read as input to recreate an instance equivalent (`equal`) to the original. We get the entire list with one call to `read`, and then hand it to `make-instance` using `apply`. Unlike `print-object`, there is no built-in function `read-object` for which we can write methods, so we define one here.

```
(defun read-object (stream)
  (apply #'make-instance (read stream)))
```

Unfortunately, unlike `print-object`, if there is a slot whose value is an instance of a class, the `read` function will simply fill that slot with the list that was written out, instead of creating an object from it. So, this `read-object` function only works for classes whose slots do not themselves have values that are instances of classes.

It would seem that we could solve this problem by writing out the call to `make-instance` in the file. Then the nested objects would be created along with



the top level one. This is essentially a program that writes a program out to a file, to be executed later. However, the `read` function will just return a list representing an expression to evaluate. So, the `read-object` function becomes

```
(defun read-object (stream)
  (eval (read stream)))
```

or, another way is to just call the `load` function. But this must be considered extremely bad practice. In essence, instead of writing data to a file to be later processed, we are writing an entire program to a file, later to be executed. We have not even begun to look into searching files, which would be very difficult with this arrangement, and one can imagine how inefficient and inflexible the whole scheme might become.

Another way to design `read-object` would be to make it a generic function, like the built-in `print-object`. Then we would be able to write methods for `read-object` for all the relevant classes. For a class that has a slot with a value that is itself an object, the method for that class would need to make an explicit recursive call to `read-object`. Just as with `print-object`, the code will soon get tedious, verbose and error-prone, even more so because the calls need to be explicit, whereas in `print-object` the method dispatch handles object-valued slots.

### 2.2.3 A Better Solution Using Metaobjects

While conventional object-oriented programming is nice, it still requires a lot of work for each new class that you add to the system. What if the information about each class were available to the running program? Then instead of writing explicit code for printing and reading instances of each class, we could write a function that gets the information about the class of the object it is printing or reading, and uses that information. If this worked, we could write one general function that would work for all defined classes, and we would not need to write extra code when adding a new class to the program.

The Common Lisp Object System is implemented in a way that uses its own basic machinery, so that the programmer can indeed write code that can access class data, such as the class name, the superclasses (parent classes), lists of slot names and other information about a class. This is known as the CLOS Meta-Object Protocol (MOP) and it is briefly described in Chapter 1. Here we show how the MOP can help solve the problem of storing complex medical data in the style of keyword-value pairs, without writing a lot of extra code. Our plan is that the code will use the class definition itself as metadata, to decide what to read and write.

In particular, in Section 1.7 of Chapter 1, we showed how to define a function, `slot-names`, that returns a list of symbols naming the slots defined in a class. Applying this to the `patient` class instance on page 55, we get

```
> (slot-names temp)
(NAME HOSPITAL-ID AGE ADDRESS DIAGNOSIS ...)
```

Now, we will build the code that can read and write a representation of the patient data described above. The serial representation we will use is still based on the idea of keyword-value pairs, something like that of the previous section. Each keyword will be either a class name or a slot name, but we will not use parentheses to group data, except when they are a simple Lisp expression that should go into a particular slot. We will therefore need some way for the reader function to determine that the end of the slot name/value pairs is reached for a given instance. We will use the convention that the keyword symbol `:END` is never the name of a class or a slot, so it can be used to indicate that there are no more slots for the current instance being read.<sup>7</sup>

The `get-object` function will read from a data stream that looks like this (the ellipses just mean “lots more keyword-value pairs”):

```
PATIENT
  NAME "Ira K"
  HOSPITAL-ID "99-2345-6"
  AGE 60
  ...
:END
```

The indentations and appearance of each slot on a separate line are ignored by the Lisp `read` function, but this formatting makes it easier for human inspection. The `get-object` function starts by reading the first symbol in the input, expecting the name of a class that has a class definition already in the system. Then it reads slot-name/value pairs, repeating until the `:END` tag is encountered instead of a slot name. Indefinite iteration (`loop`) is used here because the code cannot “look ahead” to see how many slots are present in the file, or in what order they appear. This implementation of `get-object` will handle any ordering of slot data. Not all slots need to be specified in the data file (some might have suitable default values).

```
(defun get-object (in-stream)
  (let* ((current-key (read in-stream))
        (object (make-instance current-key)))
    (loop
      (setq current-key (read in-stream))
      (if (eq current-key :end)      ;; no more slots?
          (return object)
          (setf (slot-value object current-key)
                (read in-stream))))))
```

Note that the `get-object` function will work for *any* class defined in our program, because it uses the class definition rather than having explicit names here. We do not have to write any methods for anything. It does not even have to be a generic function itself, since the same exact code executes for all classes. Thus, the class definition serves as the metadata.

---

<sup>7</sup>Note that here `:END` is a keyword in the technical sense that it is in the Common Lisp **keyword** package. Elsewhere in this section we are using the term “keyword” in the generic sense of “label”.

In an industrial-strength program, we might add some error checking to detect malformed or unrecognizable input, to make the code more robust. One problem certainly is that the code will signal an error if it encounters a symbol naming a non-existent slot for the class being read. Similarly, an error will be signaled if at the beginning a class name is encountered for which there is no class definition.

However, in well controlled circumstances this may not be necessary. In the author's radiation therapy planning system, the code described here has been used for over ten years, to plan radiation treatments for over 5,000 patients, and includes no error checking code. Error checking is not needed because these data files are only created by the same program, and do not come from any other source. The program code can guarantee that the syntax and semantics are correct, as defined here.

In order to create these files, we need a complementary function to write the data to a file, in the format shown. This function, which we call `put-object`, works with the same machinery we described above. It uses the `slot-names` function defined previously to get a list of the slots, and iterates over them writing out the contents of each.

```
(defun put-object (object out-stream &optional (tab 0))
  (tab-print (class-name (class-of object)) out-stream tab t)
  (dolist (slotname (slot-names object))
    (when (slot-boundp object slotname)
      (tab-print slotname out-stream (+ 2 tab))
      (tab-print (slot-value object slotname)
                  out-stream 0 t)))
  (tab-print :end out-stream tab t))
```

The `tab-print` function provides the indentation and new lines when needed to keep the file reasonably human readable. It puts in as many leading spaces as specified by the `tab` parameter, and includes a trailing space, so that keywords and values will be separated.

```
(defun tab-print (item stream tab &optional (new-line nil))
  (format stream "~A~S "
    (make-string tab :initial-element #\space)
    item)
  (when new-line (format stream "%")))
```

This works fine if all the slots have values that are simple Lisp objects, i.e., can be printed readably by the standard printing system (including lists, arrays and structures). However, it will fail if the slot value is itself an object (instance of a class) or contains objects, e.g., in a list. In the example on page 49, the `diagnosis` slot actually contains a structured object, an instance of the `diagnosis` class. It would fail because `format`, like `print`, will not have any way to deal with an instance of a user-defined class. This is exactly the same problem we encountered in Section 2.2.2. This time, however, we can extend the metadata to enable the code to handle slots whose values are class instances and other ordinarily unprintable data.

A straightforward way to extend `get-object` and `put-object` to handle this case is to make them recursive, i.e., if the slot contents is an object, then make a recursive call to read or print it. Similarly, if the content of the slot is a *list* of objects, an iteration over the list will be needed. (Iteration is not needed for a list of simple Lisp data, since the Lisp system can handle it directly.)

In order to decide which kind of slot contents we are dealing with, we will need a new function, `slot-type` which will return one of three keywords: `:simple`, `:object` or `:object-list`. The slot type `:simple` means that the contents of the slot is a printable and readable Lisp data type, one that has a well defined printed form. The slot type `:object` indicates that the content of the slot is an instance of some kind of object, and therefore cannot be read by the standard Lisp function `read`. In this case, as noted, we just make a recursive call to `get-object`. The slot type `:object-list` means that the slot contents are a list of objects, which will require an iterative call to `get-object`.

A `case` form takes care of the three cases, depending on the value returned by `slot-type`. Here is `get-object` with these recursive and iterative calls added.

```
(defun get-object (in-stream)
  (let* ((current-key (read in-stream))
        (object (if (eq current-key :end)
                     nil ;; end of object list
                     (make-instance current-key))))
    (loop
      (setq current-key (read in-stream))
      (if (eq current-key :end) ;; no more slots?
          (return object)
          (setf (slot-value object current-key)
                (case (slot-type object current-key)
                  (:simple (read in-stream))
                  (:object (get-object in-stream))
                  (:object-list
                     (let ((slotlist '())
                           (next-object nil))
                       (loop
                         (setq next-object
                               (get-object in-stream :parent object))
                         (if next-object
                             (push next-object slotlist)
                             (return (nreverse slotlist)))))))))))
```

Indefinite iteration is used also in the inner loop because the code has no way of determining in advance how many objects there might be, if the slot value is a list of objects (in the file). We need a way to indicate the end of the list, and here again, the keyword `:END` serves the purpose. If `:END` is detected where a class name should be, it signals that there are no more objects, so `get-object` should return `nil`, which terminates the loop when `next-object` is tested.

In the inner loop, we find a code pattern that occurs often, in which a value is computed, tested, and if non-`nil`, used in a subsequent expression. In such cases, it would be nice to use a “pronoun” type of reference, i.e.,

```
(if (compute-something) (use it) (return "result was nil"))
```

where the symbol `it` refers to the result of `(compute-something)`. Such expressions are called “anaphors” by Graham [40, Chapter 14], and he proposes the following “anaphoric” version of the `if` operator, called `aif`, for “anaphoric if”.

```
(defmacro aif (test then &optional else)
  '(let ((it ,test))
    (if it ,then ,else)))
```

Note that all the inputs are *code* to be restructured and executed. The `then` and `else` expressions can use the symbol `it` to refer to the result of the `test` expression. Using `aif` we can rewrite `get-object` as follows:

```
(defun get-object (in-stream)
  (let* ((current-key (read in-stream))
        (object (if (eq current-key :end)
                     nil ;; end of object list
                     (make-instance current-key))))
    (loop
      (setq current-key (read in-stream))
      (if (eq current-key :end)      ;; no more slots?
          (return object)
          (setf (slot-value object current-key)
                (case (slot-type object current-key)
                  (:simple (read in-stream))
                  (:object (get-object in-stream))
                  (:object-list
                     (let ((slotlist '())
                           (next-object nil))
                       (loop
                         (aif (get-object in-stream :parent object)
                             (push it slotlist)
                             (return (nreverse slotlist)))))))))))
```

Similarly for `put-object`, we add a `case` form, to handle each of the three possible slot types, and here is the expanded implementation.

```
(defun put-object (object out-stream &optional (tab 0))
  (tab-print (class-name (class-of object)) out-stream tab t)
  (dolist (slotname (slot-names object))
    (when (slot-boundp object slotname)
      (tab-print slotname out-stream (+ 2 tab))
```

```

(case (slot-type object slotname)
  (:simple
   (tab-print (slot-value object slotname)
              out-stream 0 t))
  (:object
   (fresh-line out-stream)
   (put-object (slot-value object slotname)
               out-stream (+ 4 tab)))
  (:object-list
   (fresh-line out-stream)
   (dolist (obj (slot-value object slotname))
     (put-object obj out-stream (+ 4 tab)))
   (tab-print :end out-stream (+ 2 tab) t))))
(tab-print :end out-stream tab t))

```

In `put-object`, the iteration over the list in a slot containing an object list can be definite iteration using `dolist` or `mapc`, since the list is available at the outset (the object already exists and the slots have values). These two ways to do iteration over a list are equivalent, and it is a matter of style which you might prefer.

Note that the recursive calls to `put-object` and `tab-print` include increments to the amount of whitespace to indent, so that objects that are values in slots get nicely pretty-printed in the data file. The choice of 4 and 2 for the incremental amounts of whitespace is arbitrary but seemed to work well in the author's radiation therapy planning system.

One detail remains. There is no built-in function `slot-type` so we have to write it. This function should return a keyword identifying the contents of the slot as `:SIMPLE` for readable and printable Lisp objects, `:OBJECT` for an instance of a user-defined class, and `:OBJECT-LIST` for a list of such instances. It is the one element that needs to be defined for each class, if the slots are expected to contain other objects or lists of objects. In a typical application of this code, most classes will have simple slots and we only need a default method. For more complex classes, it is not hard to write a method that just returns the non-simple slot types, as compared with writing a method for `print-object`, which would be many times larger. Here is the default method:

```

(defmethod slot-type ((object t) slotname)
  (declare (ignore slotname))
  :simple)

```

For the patient class, we might have the following:

```

(defmethod slot-type ((object patient) slotname)
  (case slotname
    ((address diagnosis) :object)
    (otherwise :simple)))

```

In the `otherwise` clause, if this class is a subclass of a big class or is several layers down in a class hierarchy, it might be onerous for the programmer to look up

all its parent classes, so all the slots can be accounted for. Since there is always a default method, one only needs to put in the **case** form any *new* non-simple slots, and a simple way to handle the **otherwise** clause is to use the standard CLOS function **call-next-method**, which will then use the previously written methods to look up any inherited slot. Here is a version of **slot-type** for the patient class, using **call-next-method**.

```
(defmethod slot-type ((object patient) slotname)
  (case slotname
    ((address diagnosis) :object)
    (otherwise (call-next-method))))
```

An alternative way to write these methods would be to write a method for each value of **slotname**, using **eql** specializers. This is much less desirable because it requires that any slot with a given name (possibly used in many different classes) must have the same meaning in terms of content type. This restriction creates global constraints on names, making it more work to add new classes.

Other enhancements are possible, e.g., adding a **:BIN-ARRAY** type, for large binary data arrays stored in separate files. To handle such files we could create a function **read-bin-array** for reading such files and a companion **write-bin-array** for producing them. Since the binary data are in another file, the text file should contain information for how to read that file, i.e., the filename and the array dimensions. Medical image data, discussed in Section 2.2.4, are an example of the need for this. In that section we show how to implement these two functions.

Although it would seem that this is an inefficient solution to storing patient data, it has been used successfully for over ten years in the author's radiation therapy planning (RTP) system, *Prism* [68], which has been in clinical use at the University of Washington Medical Center, as well as at several other clinics.<sup>8</sup> Over 40,000 complete records of patient data and radiation treatment plans are kept on line, need to update this number from the first day in 1994 when the system was put into clinical use. Index files help to find individual patients' data, and retrieval is fast enough for clinical use. Each patient case instance is stored in a separate file in the file system, with a unique filename generated by the system. Thus, each file is small, and access is quick, requiring only a lookup of the filename in the index, and the opening and reading of a relatively small file. It is possible that a relational database back end could be used in this application, but it is not at all clear that this would perform better. A brief introduction to relational database ideas is in Section 2.3. An object-oriented database would be a good match here, but no reasonable implementations were available in the early 1990's when Prism was designed and implemented. In Section 2.3.7 we give a brief introduction to the ideas of object-oriented databases.

---

<sup>8</sup>A brief introduction to cancer, radiation therapy and radiation treatment planning may be found in Chapter 14, which describes other applications of the principles and methods of Part I to problems of cancer treatment.

### 2.2.4 Medical Images: Incorporating Binary Data

So far, in the patient record example, we have considered only data that are easily and logically stored as text. Medical image data, on the other hand, are not efficiently represented as character code sequences. An image is a two dimensional array of numbers, where the number at a pixel location represents the brightness (or color) of the image at that spot. The pixel array is understood to represent samples of image data on a regular rectangular grid. The numbers can be integers in the range 0 to 4095 (typically) for Computed Tomography (CT) images, and small decimal numbers for Positron Emission Tomography images. Such numbers, when represented in the form of text, will take up four or more bytes (one for each digit) plus space characters or other delimiters. In the case of CT and Magnetic Resonance (MR) images, only two bytes are required to represent each number in binary form (because the range of values is limited by the machine design to 16 bit signed or unsigned integers). When reading binary numbers into a computer program data structure, no conversion is necessary. By comparison, reading a character (digit) representation will require some parsing and conversion. So, for image data, the bytes (or multi-byte sequences) should instead be written and read directly as binary numbers representing numerical data, image pixels.

\*\*\* put here sample (small) image and pixels in hex.

Medical image sets can also include “annotation” data, or information about the images. Typically they include: the patient’s name (a character sequence, or text), birth date (also text), the inter-pixel spacing of the grid, in cm (may be binary, floating point format rather than simple integer, or fixed point), and many other attributes. So the parsing of an image data file can be very complex, since the interpretation of the byte stream is mixed and arbitrary according to some software designer’s choices. These choices vary widely among the different manufacturers of medical imaging systems. The difficulty of dealing with such arcane file formats led eventually to the creation of a network protocol for exchanging image data, called DICOM (see Chapter 13). The DICOM protocol still presents the problem of parsing an arcane sequence of heterogeneously encoded data items. The advantage of DICOM is that you only need to learn and implement this one protocol, which is vendor and system independent.

The DICOM protocol only addresses the exchange of data between client and server programs, but not how the individual applications should store it. This allows the application software designer to choose a storage scheme that is best for her application. After presenting a little background about medical images, we show how to extend the keyword-value pair representation to support medical images in binary form.

#### Background on Medical Images

The use of X-rays to obtain images of the internal structure of the human body has a venerable history, dating from the discovery of X-rays by Wilhelm Conrad Roentgen [110]. In recent years, the availability of small detectors for X-rays and



mathematical methods for computing two-dimensional data from one-dimensional projections has made possible the direct production of digital images by the Computed Tomography (CT) scanner [53]. This can be said to be the beginning of digital imaging. As detector technology advanced, it became possible to directly produce digital X-ray images that were previously done with planar sheets of film (Computed Radiography). Other kinds of digital imaging techniques as well were developed during this period, the last three decades of the 20th century. The repertoire of the modern radiology department now also includes magnetic resonance imaging (MRI), ultrasound imaging, positron emission tomography (PET), and various kinds of nuclear isotope scans. A good introduction to the physical aspects of medical image production and processing is the textbook of Hendee and Ritenour [50].

Digital images consist of two kinds of information. The most apparent is the image itself, consisting of an array of numbers, which can be integer or decimal. Each number represents a picture element, or *pixel*. At that spot in the image, the display may show a monochrome brightness corresponding to the value of the number, or a color with brightness, hue and saturation qualities (or equivalently, a mixture of red, green and blue intensities). The size of the array may vary for a given modality. For CT images, an array of 512 rows by 512 columns is common. MRI data more often are in smaller arrays, 256 by 256. The array may not be square; computed radiographs that correspond to chest films are typically W by H pixels. The second kind of information associated with a digital image is the descriptive data that specifies what kind of image it is, how it was obtained, who the subject (patient) is, where in some patient or machine-centric coordinate system the image is located, its orientation, and many other possibly useful items. These are encoded in many different ways.

find out what these values typically are

Further, images may be aggregated into sets, and the set of images may have its own attributes, which apply in common to all the images. One very important concept is that of a position related set, where all the images share the same coordinate system, and are oriented and located in that coordinate system, relative to each other. This allows the possibility that the set of two-dimensional images can be reformatted to provide two-dimensional images in other planes through the volume that the set represents. The images can be coalesced or interpolated to form a regular three dimensional array of image data. Solid models or surfaces that represent the three-dimensional structures seen in cross-section in the images can then be computed from it.

\*\*\* some more samples here...

## Representing Image Data

A typical encoding of an image as a data structure in a computer program is as an instance of an image class. The following code shows how such a class might be

defined:<sup>9</sup>

```
(defclass image ()
  ((uid :type string :accessor uid :initarg :uid)
   (patient-id :accessor patient-id :initarg :patient-id
                :documentation "The patient id of the
                                patient this image belongs to.")
   (image-set-id :accessor image-set-id :initarg :image-set-id
                  :documentation "The image set id of the
                                primary image set the image belongs to.")
   (position :type string
              :accessor position :initarg :position
              :documentation "String, one of HFP, HFS,
                                FFP, FFS, etc. describing patient position as
                                scanned (Head/Feet-First Prone/Supine, etc).")
   (description :type string
                 :accessor description :initarg :description)
   (origin :type (vector single-float 3)
            :accessor origin :initarg :origin
            :documentation "Origin refers to the location in
                                patient space of the corner of the image as defined
                                by the point at pixel array reference 0 0 or voxel
                                array reference 0 0 0.")
   (size :type list ;; of two or three elements, x y z
          :accessor size :initarg :size
          :documentation "The size slot refers to the physical
                                size of the image in each dimension, measured in
                                centimeters in patient space.")

   ;; ...other slots

  )
  (:default-initargs :id 0 :uid "" :patient-id 0
                     :image-set-id 0 :position "HFS"
                     :description ""
                     ;; etc.
  )
  (:documentation "The basic information common to all types of
images, including 2-D images, 3-D images."))
```

This general class is then specialized to `image-2d` and `image-3d` which have further special attributes. An `image-2d` depicts some two-dimensional cross sectional

---

<sup>9</sup>In this class definition, we have included a new element, documentation for each slot, and for the class as a whole. In Common Lisp, documentation strings are part of the program, and are accessible to a running program, unlike comments in source code, which are ignored by compilers and interpreters and not available to the program itself.

or projected view of a patient's anatomy and is typically a single CT image, an interpolated cross section of a volume, or the result of ray tracing through a volume from an eyepoint to a viewing plane. Here is the `image-2d` subclass:

```
(defclass image-2d (image)
  ((thickness :type single-float
              :accessor thickness :initarg :thickness)
   (x-orient :type (vector single-float 3)
              :accessor x-orient :initarg :x-orient
              :documentation "A vector in patient space defining
the orientation of the X axis of the image in the
patient coordinate system.")
   (y-orient :type (vector single-float 3)
              :accessor y-orient :initarg :y-orient
              :documentation "See x-orient.")
   (pix-per-cm :type single-float
                :accessor pix-per-cm :initarg :pix-per-cm)
   (pixels :type (simple-array (unsigned-byte 16) 2)
            :accessor pixels :initarg :pixels
            :documentation "The array of image data itself.")
  ))
```

The `pixels` slot for an `image-2d` contains a binary two-dimensional array, in which the value at each location in the array refers to a sample taken from the center of the region indexed, and values for images with non-zero thickness refer to points mid-way through the image's thickness. The origin of the `pixels` array is in the upper left hand corner, and the array is stored in row-major order so values are indexed as row, column pairs, i.e., the dimensions are y, x.

Typically, cross sectional medical images such as X-ray Computed Tomography (CT), Magnetic Resonance Images (MRI), and Positron Emission Tomography (PET) are generated in sets, in which the orientation of the images, their scale factors and their positions in the patient are all related. Such a collection of images is called a "position related set", or "series". The simplest way to represent such a collection in a program is as a list of `image-2d` objects. The list can be searched and the data extracted for display and other computations.

\*\*\* maybe put in code for creating sagittal and coronal reformatted images from transverse sets?

In addition to being useful as 2-dimensional images, the set can be reassembled into a three-dimensional image. This kind of object is represented by the class `image-3d`.

```
(defclass image-3d (image)
  ((voxels :type (simple-array (unsigned-byte 16) 3)
           :accessor voxels
```

```

:initarg :voxels
:documentation "a 3-D array of image data values")

)

(:documentation "An image-3D depicts some 3-D rectangular
solid region of a patient's anatomy.")

)

```

In the 3-D image, `voxels` is the three-dimensional array of intensity values or colors coded in some standard way, e.g., as RGB values. The value at each index of the array refers to a sample taken from the center of the region indexed. The origin of the `voxels` array is in the upper left back corner and the array is stored in row, then plane major order, so values are indexed as plane, row, column triples, i.e. the dimensions are ordered z, y, x.

Rearranging the set of 2-D pixel arrays into a single 3-D voxel array makes some image processing calculations simpler. These include volume rendering and the computation of projected images (simulated X-ray films, also called Digital Reconstructed Radiographs, or DRR).

### Reading and Writing Image Data

Software packages to display and manipulate medical images use many different forms of storage of the image data in files. One way to do this is to store the pixel values as 16 bit binary numbers, and put the other information in a separate text file, formatted as a tagged data stream, just as we did for the patient data example in Section 2.2.3.

Figure 2.3 is an example of a keyword-value arrangement for a text file containing the “annotations” or descriptive information about the images. Each instance begins with a keyword naming the class, `IMAGE-2D`, and then keyword-value pairs for each slot in the class, i.e., slot names followed by values in a format acceptable directly to the Lisp `read` function. These slots correspond to the `image-2d` class (including slots inherited from the `image` class). This file can be processed and ones like it produced by the `get-object` and `put-object` functions described in Section 2.2.3. Note that the `pixels` slot does not contain actual image pixel values, but instead carries a list of information describing a separate file containing the pixel values. This information would be used by a new procedure `read-bin-array` to read in the actual array data and store it in the slot of the newly constructed object, as we suggested earlier.

We now turn to the implementation of `read-bin-array` and `write-bin-array` mentioned previously. By separating the pixel array from the annotations, we can read in the pixel array data very efficiently and simply, using the standard Common Lisp function `read-sequence`. This function reads bytes from a binary file until the sequence passed to it is filled. The sequence will contain binary elements as specified by the element type of the file. The `read-sequence` function can determine the size

```

IMAGE-2D
  DESCRIPTION "Pancreatic cancer: abdominal scan"
  ACQUISITION-DATE "24-May-1994"
  ACQUISITION-TIME "16:40:23"
  IMAGE-TYPE "X-ray CT"
  ORIGIN #(-17.25 17.25 0.0)
  SCANNER-TYPE "AJAX 660"
  HOSPITAL-NAME "University Hospital, Hackland, WA"
  RANGE 4095
  UNITS "H - 1024"
  SIZE (34.504 34.504)
  PIX-PER-CM 14.8388
  THICKNESS 0.500
  X-ORIENT #(1.000 0.000 0.000 1.000)
  Y-ORIENT #(0.000 -1.000 0.000 1.000)
  PIXELS ("pat-1.image-1-1" 512 512)
:END
IMAGE-2D
  DESCRIPTION "Pancreatic cancer: abdominal scan"
  ACQUISITION-DATE "24-May-1994"
  ACQUISITION-TIME "16:40:23"
  IMAGE-TYPE "X-ray CT"
  ORIGIN #(-17.25 17.25 1.0)
  SCANNER-TYPE "AJAX 660"
  HOSPITAL-NAME "University Hospital: Hackland, WA"
  RANGE 4095
  UNITS "H - 1024"
  SIZE (34.504 34.504)
  PIX-PER-CM 14.8388
  THICKNESS 0.500
  X-ORIENT #(1.000 0.000 0.000 1.000)
  Y-ORIENT #(0.000 -1.000 0.000 1.000)
  PIXELS ("pat-1.image-1-2" 512 512)
:END
IMAGE-2D
  DESCRIPTION "Pancreatic cancer: abdominal scan"
  ...

```

Figure 2.3: A portion of a sample image set data file

of the sequence passed to it, since in Common Lisp sequences are objects that carry with them information about their size. Here is a version that creates a new array, given the total size in bytes. The array is a one dimensional array, since that is the only type that `read-sequence` can process. The elements are 16 bit binary quantities, not 8 bit bytes.

```

(defun read-bin-array (filename size)
  (let ((bin-array (make-array (list size)

```

```

                                :element-type '(unsigned-byte 16)))
(with-open-file (infile filename :direction :input
                                :element-type
                                '(unsigned-byte 16))
  (read-sequence bin-array infile))
bin-array))

```

Alternately, if the receiving array were already in existence it could be passed in, instead of passing a size parameter. In this case, the function need not return the array as a value, since it directly modifies the array that is passed in.

```

(defun read-bin-array (filename bin-array)
  (with-open-file (infile filename :direction :input
                                :element-type
                                '(unsigned-byte 16))
    (read-sequence bin-array infile)))

```

As nice as this is, it is better from the point of view of the program that uses the array if it is a 2-dimensional array instead of a 1-dimensional array. If it is a 2-D array, the program that uses it can explicitly use row and column indices instead of explicitly computing offsets into the linear array. However, as noted, the `read-sequence` function handles only 1-dimensional arrays. One solution to this problem is to put a loop in `read-bin-array`, and read a row at a time, but this turns out to be tricky and inefficient.

This problem of needing to index an array in different ways comes up sufficiently often that the Common Lisp standard includes a facility to do this in a standard way. We can create a 2-dimensional array, and then create another array which references exactly the same elements but is indexed as a 1-dimensional array. This is called a *displaced array*. It is used for other purposes as well, e.g. referring to a part of an array offset by a fixed amount, e.g., as a “subarray.” So, here is the 2-D version of `read-bin-array`:

```

(defun read-bin-array (filename dimensions)
  (let* ((bin-array (make-array dimensions
                                :element-type '(unsigned-byte 16)))
         (disp-array (make-array (array-total-size bin-array)
                                :element-type '(unsigned-byte 16)
                                :displaced-to bin-array)))
    (with-open-file (infile filename :direction :input
                                :element-type '(unsigned-byte 16))
      (read-sequence disp-array infile))
    bin-array))

```

Alternatively, as in the 1-D case, if we want to supply the array to be filled as an input,

```

(defun read-bin-array (filename bin-array)

```

```

(let ((disp-array (make-array (array-total-size bin-array)
                             :element-type '(unsigned-byte 16)
                             :displaced-to bin-array)))
  (with-open-file (infile filename :direction :input
                  :element-type '(unsigned-byte 16))
    (read-sequence disp-array infile))))

```

Since the array passed to `read-sequence` is a 1-D array, the code works efficiently. In particular, there is no need to read in a one-dimensional array and then copy it to a separate two-dimensional array. The 3-D array case is left as an exercise for the reader.

For generality, the three blocks of code can be combined into one function with a conditional branch on the rank of the desired (or supplied) array. To get the ultimate efficiency achievable in these functions, it would be important to include declarations so that the compiler can generate in-line code and optimize the use of storage for the array values.

Now we can add this support to the `get-object` function. It only requires adding one more branch to the `case` expression, to handle the `bin-array` slot type. We use here the first version, where the second argument is the array dimensions list. In the file, it is assumed that the slot data are the filename of the binary data and the array dimensions, as shown in Figure 2.3.

```

(defun get-object (in-stream)
  (let* ((current-key (read in-stream))
        (object (if (eq current-key :end)
                    nil ;; end of object list
                    (make-instance current-key))))
    (loop
      (setq current-key (read in-stream))
      (if (eq current-key :end) ;; no more slots?
          (return object)
          (setf (slot-value object current-key)
                (case (slot-type object current-key)
                  (:simple (read in-stream))
                  (:object (get-object in-stream))
                  (:object-list
                     (let ((slotlist '())
                           (next-object nil))
                       (loop
                         (aif (get-object in-stream :parent object)
                             (push it slotlist))
                         (return (nreverse slotlist))))))
                  (:bin-array
                     (let ((bin-info (read in-stream)))
                       (read-bin-array (first bin-info)
                                         (rest bin-info))))
                  ))
          (return object))))))

```

Now we need the counterpart of `read-bin-array`, which takes an image pixel array and writes it to a specified file. This function, `write-bin-array`, will take as input a file name for the file to write, and the binary array. It is not necessary to also provide the dimensions, since the array has that information available. The idea is to get the array dimensions and in an iteration (using `dotimes` is probably the simplest) just call `write-byte` to write each pixel value to the file. The only thing to be careful of is to open the file with the specification that the element type is (`unsigned-byte 16`). This is an optional input to `with-open-file` or the `open` function.

Then, the `write-bin-array` function can be used in an extended version of `put-object`, in the same way as we extended `get-object`. Both of these functions are left as exercises for the reader.

Many other arrangements are used by image processing software. A typical approach is to put everything in one file, mixing text and binary representations for numbers, with all the descriptive information first, followed by the pixels in the same file.

\*\*\* illustration and discussion of what a hassle this would be, perhaps could put excerpts from PLAN-32 readtape program? or a translation into Lisp

\*\*\* Might also discuss various attempts at watermarking, and something about image compression

## 2.3 Database Systems and Ideas

Although the tag-value system described in Section 2.2.3 works well in some biomedical applications, often the regularity of structure of the data suggest that a more efficient system could be built. The problem of dealing with large and complex but regular data is sufficiently broadly important that an entire subfield of computing, the study of database systems, has developed to address it. The following are definitions of some useful terms in reference to database systems.

**Entity** a collection of data treated as a unit, e.g., a radiologic image, a patient, a radiation beam, a laboratory test panel, an encounter record.

**Relationship** expresses associations between entities, e.g., a particular radiologic image *belongs to* a particular patient. This is *not* the same as a *relation*, or *table*, which is specific to the Relational Data Model, and is described in Section 2.3.1.

**Schema** a description of the kinds of entities in a database, and their relationships.

**Query** specifies an operation on a database, resulting in a new collection of data, arranged in some way, possibly a new schema.



**Data model** a general framework for defining schemas and queries. Examples are the relational model and the object oriented model.

**View** a table similar to a relation, but views are not stored in the database system; they are computed as a result of a query.

Many different data models are in use, including the relational model, the object-oriented model, the (hybrid) object-relational model, the entity-attribute-value model, the entity-relationship model, and others. In Section 2.2.2 we gave an illustration of object-oriented modeling with entity-relationship diagrams. A more elaborate treatment may be found in a new edition [11] of a classic text [115].

### 2.3.1 The Relational Model

When data logically match the idea of many identically structured records, the most powerful model for describing, storing and querying large data sets appears to be the Relational Database Model [20, 104]. So much of the data used in business, education, government and other organizations has this regularity that the relational model is the dominant model for database systems. Many products implement relational database systems, and large numbers of applications are built with a relational database back end. In this section we give only a brief sketch of the core ideas of relational databases. A more in-depth treatment can be found in textbooks devoted specifically to database systems [104, 30].

The core idea of the relational model is that data are organized into a set of *relations*, also called “tables”. Each table has rows and columns, in which a row represents a single record and a column represents an attribute that has some value in each record. In a given column, the type and format of the data that can be in that column are the same for every row. Every row has the same number of data elements, in the same order as every other row. An example is shown in Figure 2.4.<sup>10</sup>

<i>Name</i>	<i>Hosp_ID</i>	<i>Tumor_site</i>	<i>Age</i>
Joe Pancreas	11-22-33	Pancreas	38
Sam Vacuum	15-64-00	Glioblastoma	60
Sue Me Too	99-88-77	Lt. Breast	45
David Duck	88-77-99	Pancreas	46
Norbert Pivnick	-6	Prostrate	62
...			

Figure 2.4: A table for the *Patients* schema

Thus, a table is *not* a spreadsheet, which allows free form values in any box representing a particular row and column. In a spreadsheet, the row numbers just

<sup>10</sup>You may notice that there is a misspelling in this table, where someone has put in “Prostrate” instead of “Prostate”. This is deliberate on the part of the author, to illustrate a point about data quality (see Section 2.4).

serve to index the rows and similarly the column numbers index the columns. In a spreadsheet, anything can go in any box, and a single column can have different types of data in different rows.

Some characteristics of a relational database are:

- Data types are limited to a few primitive types: fixed length strings, integers in a predefined range, floating point numbers in a fixed range, and some provisions for pointers to large data aggregates such as medical images (so-called binary large objects, or BLOBS).
- Schemas consist of descriptions of tables. All rows in a table have identical form.
- The query language is SQL. It provides for creating, updating, and searching tables, and forming new tables (views) from existing ones.

### 2.3.2 A Brief Introduction to SQL

SQL (for “Structured Query Language”) is the standard relational database query language. It provides statements for defining tables, (and more generally, database schemas) and statements for populating, updating, and querying them. The following illustrates the essential ideas.

To define a table structure, one uses the `CREATE TABLE` construct. Here is an example. The *Patients* schema consists of a single table, described by the following:

```
CREATE TABLE Patients
    (Name CHAR(20), Hosp_ID CHAR(10),
     Tumor_site CHAR(20), Age INTEGER)
```

The name of the table follows the `CREATE TABLE` command, then in parentheses, separated by commas, is a sequence of attribute descriptors. Each attribute descriptor is required to have a name and type declaration.

The table shown in Figure 2.4 is an excerpt from an instantiation of this table definition. The table is maintained by `INSERT`, `DELETE`, and `UPDATE` commands. Here are some examples of those.

\*\*\* add explanation of each of these, describe the `WHERE` clause

```
INSERT
INTO Patients (Name, Hosp_ID, Tumor_site, Age)
VALUES ('A. Aardvark', '00-12-98', 'Nasopharynx', 57)

DELETE
FROM Patients P
WHERE P.name = 'I. Kalet'

UPDATE Patients P
SET P.Hosp_ID = 65-21-07
WHERE P.Name = 'Norbert Pivnick'
```



This query would produce the result shown in Figure 2.7.

<i>Name</i>	<i>Tumor_site</i>
Sam Vacuum	Glioblastoma
Norbert Pivnick	Prostrate
...	

Figure 2.7: A view table resulting from a nested query involving both the Patients relation and the Treatments relation

While SQL does not have the full power of a “Turing-equivalent” programming language like Lisp, C, FORTRAN or java, it provides a very powerful way to define data organization and to search and manipulate it as is most often needed with database systems. Most database management system implementations also include a way to embed SQL expressions in a regular programming language, so that ultimately any computation may be done with the data.

### 2.3.3 Constraints and Keys

A reasonable database system would support ways of insuring that the data stored in it are consistent with the schemas, and also allow for additional constraints to be specified where needed. Already we have enforceable constraints on elements that can be in each column, because we have included *type* specifications in the table definitions. These can be enforced by the data entry facilities. In addition, no two rows in a table can be identical, i.e., each pair of rows must differ in the value of at least one column, or attribute. Further constraints can be specified in SQL, called *key constraints*, by defining one or more fields in a relation to have the property of being a *key*. Here are definitions for the various types of key.

\*\*\* expand the following or add explanation below

**Key** is a set of fields (in a table) for which each row has a unique combination, e.g.

```
CREATE TABLE Patients
  (Name CHAR(20), Hosp_ID CHAR(10),
   Tumor_site CHAR(20), Dose INTEGER, UNIQUE (Hosp_ID))
```

specifies that no two records will have the same value for the **Hosp\_ID** field.

**Superkey** set of fields containing a key as a proper subset, e.g. in the above, (Name, Hosp\_ID) would constitute a superkey, since the second element is a key. Note that a superkey also satisfies the uniqueness property, but only for the key or the aggregate, not for a field that is included but is not itself a key.

**Primary key** at most one key can be declared to be the *primary key* for a relation. While this is not necessary to relate one table to another, it helps the database system optimize queries, for example by generating an index.

**Foreign key** links information from one table to that in another table, e.g., for the Treatments table, a better design would be

```
CREATE TABLE Treatments
  (Hosp_ID CHAR(10), Treatment_Type CHAR(10),
   PRIMARY KEY (Hosp_ID),
   FOREIGN KEY (Hosp_ID) REFERENCES Patients)
```

### 2.3.4 Relational Algebra

We already showed the *selection* operation, which picks some subset of rows from a relation.

The *projection* operation picks a subset of columns (fields).

A *cross product* forms a new composite table from two other tables – for every combination of an element from one with an element of the second, there is a row in the cross product.

A *join* is a cross product followed by a selection and/or projection operator. A common kind of join is one that selects rows in which fields with the same (field) name are equal. So Patients joined with Treatments gives a table where the “Radiation type” and “Dose” fields are added on for each patient for which there is a record in both tables. The result of such a join looks like Figure 2.8.

<i>Name</i>	<i>Hosp_ID</i>	<i>Tumor_site</i>	<i>Age</i>	<i>Radiation_type</i>	<i>Dose</i>
Sam Vacuum	15-64-00	Glioblastoma	60	neutrons	960
Sue Me Too	99-88-77	Lt. Breast	45	electrons	5040
Norbert Pivnick	-6	Prostrate	62	neutrons	840
...					

Figure 2.8: A table showing both tumor site and treatment for each patient

\*\*\* show formal notation for relational algebra here

### 2.3.5 Relational Calculus

\*\*\* some examples, notation and forward pointer to FOL in Chapter 3

### 2.3.6 More on the Relational Model

\*\*\* Lisp interface to SQL - see PK code to abstract the Franz interfaces to Oracle and MySQL

\*\*\* advantages, disadvantages of SQL. Maybe some code illustrating how to

fake a RDBMS, illustrating the idea of “metadata”.

\*\*\* Refs to Hao Li’s project and conversion of ontologies to RDB schemas and vice versa

### 2.3.7 Other Data Models

#### Object-oriented Databases

\*\*\* Object-oriented databases and object-relational systems

#### The Entity-Attribute-Value (EAV) Model

We consider one more flexible approach related to the tagged data idea, that of an Entity-Attribute-Value (EAV) database. In this model, every piece of data is in the same form, a triple, in which the three elements are: a reference to some object, the name of an attribute of that object, and the value associated with that attribute. The value could possibly be a reference to another object, or it could be a basic datum (a number, a string or other elementary datum). Some attributes may have multiple values, e.g., PSA test results, in which case there will be multiple EAV triples, one for each test result.

Examples:

`(patient-6 creatinine 1.2)`

`(radiation-beam-3 gantry-angle 45.0)`

A value could be another entity, as in

`(patient-6 treatment radiation-beam-3)`

This could be represented in a relational database as a simple three column table. The efficiency is offset by the complexity of queries, which may not be possible in SQL.

\*\*\* need more elaborate example here

\*\*\* put forward pointer to end of RDB section, showing how to implement EAV in an RDB system

\*\*\* use MIND as an example here?

## 2.4 Data Quality

\*\*\* also include data presentation? Possibly make this a separate chapter along with controlled vocabulary stuff, etc.

All the discussion to this point somewhat implicitly assumes that the data contained in the various long term storage schemes are of high quality, i.e., there are no misspellings of names of patients, or diseases, or other text strings. By now you will have noticed that in Figure 2.4, the term “Prostate” is misspelled as “Prostrate”, which is a position, not a disease or an organ. This typographical error then propagates to views, as in Figures 2.5 and 2.8. This kind of error often happens when the data entry is done by typing into a text field in a computerized form. One problem with this is that the record will not turn up or be counted in a search, e.g., for all patients with *prostate* cancer. To alleviate the data entry problem, large “controlled vocabularies” have been built to standardize the terminology that people use in recording medical information. Another strategy for reducing such errors is to provide structured data entry facilities, i.e., choosing from menus rather than typing in a word or phrase that is part of a standardized vocabulary. The complexity of biological and medical data make this a challenging design problem. Especially in the context of medical practice, time pressure actually favors traditional dictation and transcription rather than direct data entry.

In many circumstances, data are already available in computerized form, and can be transmitted from system to system. An example that has been available for a long time is the planning and delivery of radiation therapy (see Chapter 14 for more background). For many decades it has been possible to create a plan for radiation treatment of tumors using a computer simulation system. The system displays cross sectional images of the patient, and also can display the radiation pattern produced by the treatment machinery. The planner, called a “dosimetrist”, uses interactive controls on the display to position the radiation beams. When the plan is completed, it can be written on tape, disk or other media, or transmitted over a computer network to a computer controlled radiation treatment machine. In the early days of such facilities, the radiation therapist was required to enter the machine settings from a printed sheet produced by the planning computer. The computerized version of the settings was used by what was called a “Record and Verify” system, to have the machine check if the therapist entered the data correctly. In my opinion (loudly expressed at the time, early to mid-1980’s) this is exactly backward. The computerized data and network connection have the potential to save time and improve reliability. The role of the radiation therapist should be to check that the machines are operating correctly, and not for the machines to check the humans. Fortunately, this view prevailed and current practice is to use a network protocol (DICOM, described in Chapter 13) to transfer the data, and to have the operators (dosimetrist and therapist) provide the quality assurance of human oversight, to insure that the machines are set up properly.

\*\*\* also include examples of quality problems from UAA

## 2.5 Decoding and Parsing of Data Streams

\*\*\* This might end up distributed in the sections above. Also, Peter Karp's Biowarehouse code, at least reference it, to be treated in Chapter 17

So far we have dealt only with input data streams that are already in a form easily decoded (and in some cases parsed) by the Lisp reader. In many situations (e.g. for receiving medical images using the DICOM protocol, described in Chapter 13) the input data stream is structured but we will need to write explicit code to extract the tokens and organize them into data structures.

Principles of parsers, hard coding vs. rules, etc.

Problems of reading various kinds of files, accessing data sources.

\*\*\* sketch the DICOM PDV parser here. Describe relation between tagged data and nested structures, relate to XML in following sections.

Here is a function to produce a small integer (`fixnum` in Lisp) from some number of bytes in a byte sequence. The `ash` (binary shift left) operation returns an integer from its input, with the bits shifted the specified amount, i.e., multiplied by the corresponding power of 2.

```
(defun decode-integer (buffer index length
                      &optional (order :big-endian))
  (assert (and (typep buffer '(array (unsigned-byte 8) (*)))
              (fixnump index)
              (not (minusp index))
              (plusp length)))
  (labels ((decode-it (start inc count answer)
              (if (>= count length)
                  answer
                  (decode-it (+ start inc) inc (+ count 1)
                              (logior (ash answer 8)
                                       (aref buffer start))))))
    (if (eq order :big-endian)
        (decode-it index 1 0 0)
        (decode-it (+ index length -1) -1 0 0))))
```

This can be used anywhere that such byte sequences need to be decoded from byte streams (usually files, but also network data streams).

Sometimes, characters are encoded as part of the same byte stream along with binary numbers representing integers. Although it is usual to use the same representation in a program for characters as their byte codes in external files, it is not always good practice to treat the data casually. The stream is defined as a sequence of bytes (8 bit binary numbers), while characters are a distinct data type, and strings consist of arrays of characters. By providing a thin layer of abstraction that “converts” the bytes into character data types in our programming language, we can write more transparent code, that can be understood at the level of text and



character manipulation without concern for its underlying representation. This thin layer of abstraction can serve as a basis for eventually using extended or alternate character sets. The following function just iterates over the byte sequence `buffer` starting at `index` to produce a string, `result` of `length` characters.

```
(defun decode-string (buffer index length)
  (do ((result (make-string length))
      (i 0 (1+ i))
      (ptr index (1+ ptr)))
      ((= i length) result)
    (setf (aref result i)
          (code-char (aref buffer ptr)))))
```

\*\*\* explain code-char here

In the typical arcane formats for storing text strings or character data, it is not unusual to have “padding”, sometimes space characters, sometimes the Null character, so the `decode-string` function can optionally trim off trailing pad characters. These padding characters are included for various reasons, including: to make the next data start on an even byte boundary, or to make a fixed length field for storing the information. In either case it appears to have been included as a convenience for programmers using certain programming languages, rather than a fundamental principle. It is easy to postprocess the converted string to trim off the trailing pad characters. Here is a version of `decode-string` that includes this.

```
(defun decode-string (buffer index length &optional (pad nil))
  (do ((result (make-string length))
      (i 0 (1+ i))
      (ptr index (1+ ptr)))
      ((= i length) (case pad
                      (:Null-Pad
                       (string-right-trim '(#\Null) result))
                      (:Space-Pad
                       (string-right-trim '(#\Space) result))
                      (t result)))
    (setf (aref result i)
          (code-char (aref buffer ptr)))))
```

Here we have included one additional input, `pad`, which is a keyword specifying the type of padding, or `nil` to indicate no padding.

## 2.6 Data Interchange Standards

All these message and data interchange schemes are independent of how medical software might store the data in a file or database. This is a critical difference between the medical approach to data representation and the analogous problem in

molecular biology. In bioinformatics much of the handling of data concerns storage in large files, in various file formats, so that file formats become the framework for discussion, e.g., the so-called FASTA format for sequence data. Programs that read files must similarly deal with parsing, but it is in a different context. The closest connection between the biological and medical data representations is access to data via web servers. The NCBI provides such access for example to GENBANK and related resources.

### 2.6.1 HL7

The HL7 standard for network interchange of clinical data introduces a framework or wrapper that enables such data to be encoded in a message.

\*\*\* critique following Barry Smith, and pointers to DICOM

### 2.6.2 DICOM

The DICOM standard for medical image data (now extended to include radiation therapy data) takes this to yet another level, in which the exchange of data is mediated by a protocol specifying the order and type of messages that can be exchanged.

See Chapter 13 for how to build DICOM client and server programs.

### 2.6.3 XML, RDF and OWL

How XML addresses the data interchange problem and storage problem, limitations of XML in expressivity, extensions, i.e., RDF and OWL (pointer to KR chapter and/or IR chapter).

#### XML – a fancier kind of parentheses

Drug information can be represented in text files with tags, just as we did with patient data.

DRUG

```

PRODRUG  NO
CONTRAINDICATION  NIL
MAJOR-PATHWAY  NIL
PRIMARY-CLEARANCE-MECHANISM  METABOLIC
PRIMARY-FIRST-PASS-ENZYME  NIL
LEVEL-OF-FIRST-PASS  (CYP3A4 INTERMEDIATE)
NARROW-THER-INDEX  YES
INDUCES  NIL
INHIBITS  ((CYP3A4 COMPETITIVE STRONG)
           (P-GLYCOPROTEIN UNKNOWN STRONG))
SUBSTRATE-OF  (CYP3A4 P-GLYCOPROTEIN)

```

```

SIDE-EFFECTS  NIL
INDICATIONS  NIL
FORM  NIL
DOSAGE  NIL
INGREDIENT  NIL
DRUG-CLASS  NIL
GENERIC-NAME  CYCLOSPORINE
:END

```

To use the drug data in a program, we define:

- a drug class (in the usual object-oriented programming sense), which is an example of an external description, i.e., program code as metadata,
- input and output functions (the functions, `get-object` and `put-object`, described in Section 2.2.3 may be used here),
- a way to find drug records or instances by name, in the drug information repository.

The idea of XML is to provide more standardized and possibly more sophisticated support for *markup*, i.e., to embed tags into the text of a document to label its parts. This allows for retrieval of the parts as entities. It is an alternative to the tag-value representation we presented.

Here is the drug record, redone in XML:

```

<?xml version="1.0"?>
<drug generic-name="CYCLOSPORINE">
  <prodrug>no</prodrug>
  <contraindication>none</contraindication>
  <major-pathway>none</major-pathway>
  <primary-clearance>metabolic</primary-clearance>
  <level-of-first-pass>
    <enzyme>CYP3A4</enzyme>
    <level>intermediate</level>
  </level-of-first-pass>
  <narrow-ther-index>yes</narrow-ther-index>
  <induces>none</induces>
  <inhibits enzyme=CYP3A4 type=competitive level=strong>
  <inhibits enzyme=p-glycoprotein type=unknown level=strong>
  <substrate-of>CYP3A4</substrate-of>
  <substrate-of>p-glycoprotein</substrate-of>
  ...
</drug>

```

XML is intended only to be a “markup” language, a way to support somewhat self-describing or semi-structured data. As pointed out in an excellent book introducing XML [47], XML is *not* a programming language and it is *not* a database system or database schema language.

**RDF – adding some predefined tags****OWL – tags for specifying classes**

Although one cannot write a complete program in XML, the extension known as the Ontology Web Language, OWL, provides some reserved keywords and associated semantics to define classes, instances and even rules. However we will still need a translator that converts this representation into the appropriate constructs in a full programming environment. A program to apply the rules and perform operations on the collection of classes and instances is also necessary. (ref to Chapter 3)

## 2.7 Data, Information and Knowledge

Before leaving the subject of biomedical data, it is worth considering how terms are used. In the next chapters we will show how to encode biomedical knowledge and compute with it. Is knowledge different from data? What about the broader term, “information”?

In distinguishing among the commonly used terms, “data,” “information” and “knowledge,” the dictionary is not much help. The Random House Dictionary of the English Language [128] defines “data” as “facts, information, statistics or the like, either historical or derived by calculation or experimentation.” It does, however, also offer the definition from philosophy, “any fact assumed to be a matter from direct observation,” which seems to exclude the derivation by calculation. “Information, on the other hand, is defined as “knowledge communicated or received concerning a particular fact or circumstance.” The words “data” and “facts” are presented as synonyms for “information.” The entry for “information” distinguishes between information as unorganized and knowledge as organized. The dictionary defines “knowledge” as “acquaintance with facts...” and suggests that “information” is a synonym.

An additional complication is that the term “information” is indeed used in the sense of communication of data or knowledge, as in *Information Theory*. This is the quantitative study of how data are transmitted and received through communication channels, accounting for noise, redundancy, synchronization, and other matters. Two good elementary introductions to this subject are the reprints of the original pioneering work of Claude Shannon [119] and a very entertaining treatment for the layman by John R. Pierce [103].

For our purposes, we use the convention that “data” refers to matters of direct observation of specific instances, or computed assertions about those specific instances. So, blood test results such as glucose level, red blood cell counts, etc. for a particular blood sample from a patient are data. We will usually use “facts” as a synonym for “data.” We reserve the term “knowledge” to refer to statements that are true of classes of individuals, or what we might also call “general principles.” So, a general principle, or chunk of knowledge, might be that, in general, the drug heparin prevents clotting of blood. Together with some facts about an individual, these chunks of knowledge can be applied to derive new facts, by logic or other calculations. When there are enough chunks of knowledge to provide a comprehensive

will we include this somewhere in the book?

and internally consistent representation of a biological system or process or entity, we call that collection a “theory”.

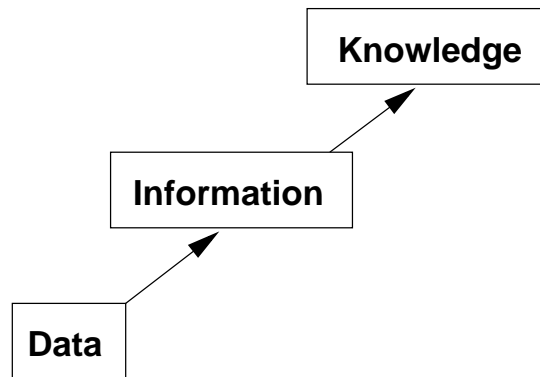


Figure 2.9: A diagrammatic illustration of the progression from data to information to knowledge

What about information? The dictionary seems to suggest that unorganized information is data, and organized information is knowledge. This does not distinguish between information about individuals vs. information about general principles, nor does it address information as a way to quantify communication (Information Theory). We will use the term “information” in all of those ways, but only where the context makes it clear what is meant. We will see that similar ideas about looking up information will apply to both data and knowledge in various forms. So, although one might imagine a progression from data to information to knowledge as shown in Figure 2.9, in reality the progression is two-dimensional, from unorganized to organized, and from particular instances to general laws and principles, as shown in Figure 2.10.

<b>Organized</b>	<b>Processed Data</b>	<b>Biomedical Theories</b>
	<b>Raw data (Facts)</b>	<b>Unstructured Knowledge</b>
<b>Unorganized</b>		
	<b>Specific</b>	<b>General</b>

Figure 2.10: A two dimensional view of data, information and knowledge

The general laws and organized information can be used to infer more facts,

and possibly to discover further general laws. Data are the means by which we test theories, and also the source of insights and patterns from which we might guess theories. This is the activity of the theoretical side of scientific research.

In clinical practice, too, one can see the same process. For example, we start with some facts from a medical record. Patient X has an adenoid cystic carcinoma (a kind of cancer) in the nasal cavity, behind the nasal vestibule, between the upper left turbinate and the nasal bone. This is information (data) about patient X. We know that this location is drained by the submandibular lymph nodes and the retropharyngeal lymph nodes. This is knowledge, because it is true in general about human anatomy, not just for a particular patient. But, it can be applied to this patient to suggest that these two nodes are at risk for metastatic disease and should be treated with therapeutic irradiation. So, the combination of data and knowledge leads to more data and a useful clinical insight. The process of medical diagnosis can also be seen as an application of knowledge to data and a stepwise theory refinement. In both diagnosis and treatment planning, instead of trying to construct a theory of biomedical or pathological processes in general, the physician is constructing a theory for a particular patient at a particular time. This is referred to as the hypothetical-deductive approach [120, pp. 70–71]. Some early medical expert systems were designed along these lines.

This neat distinction, while useful, is not so rigid as one might surmise. Consider the Foundational Model of Anatomy [112], described in more detail in Chapter 9, which we will often refer to simply as “the FMA.” It is an *ontology*, a collection of terms naming the entities in the human body and links naming the relationships between them. The intent of the authors of the FMA is to represent as precisely as possible what is known about human anatomy. To a student learning anatomy by browsing the FMA, it is a collection of information, to be understood and perhaps to some extent memorized. To a software designer, the terms and links may be considered data that are subject to search and manipulation by a computer program. To a biological scientist it is a formal theory of human body structure. The difference in labeling the contents of the FMA are really about different uses of the same things. As long as the meaning, structure and intended use are clear, we will not worry too much about the labels, “data,” “information” and “knowledge.”

## 2.8 Summary and Further Reading

The main points:

- Biomedical data are heterogeneous, highly variable in size as well as type and structure, but from molecules to humans to populations, there are some common themes.
- Self describing data (and use of metadata) organization strategies can pay off in simplifying access and storage. These methods also provide flexibility and robustness in the face of change.
- When raw speed is needed, e.g., for medical images and large sequences, special techniques are appropriate, but abstraction is still important.

- When dealing with generic descriptive data rather than characteristics of individuals, the distinction between data and knowledge is less clear.

\*\*\* to be concluded - the next chapter discusses representing and computing with *knowledge* as distinct from *data*

References:

- For XML: “XML in a Nutshell” [47]  
and see also

<http://www.w3.org/TR/REC-xml/>

- For databases, many good texts exist [104]





## Chapter 3

# Symbolic Biomedical Knowledge

*A little knowledge is a dangerous thing.*

– Alexander Pope  
An Essay on Criticism

In Chapter 2, we distinguished between data and knowledge by reserving the term “knowledge” for general principles and assertions about classes or categories, rather than individuals. The power of such general principles comes from their compactness. A few general principles may be sufficient to infer or calculate or derive a huge range of assertions that are entailed by these principles. The derivation of a particular result may involve a lot of computation, but it is done only when needed. Precomputing all possible results (or alternatively, doing experiments to observe, record and catalog all possible facts) is usually (though not always) out of the question. Even when it is possible, it raises substantial questions about indexing and retrieval, which will be addressed in Chapter 5.

A second important aspect of representing knowledge in the form of general principles is that one may take the view that the principles are descriptive, i.e., they are not just a compact summary of the facts of the world, but statements about how the world really works. These two points of view were espoused by two very prominent scientists. Ernst Mach held that the theories (equations and formulas) of physics were powerful and compact summaries of all the known and not yet observed data of the physical world, and nothing more. Albert Einstein, on the other hand, took the view that the theories were approximate descriptions of how the world really works.<sup>1</sup> This is a substantial point of discussion among biologists and medical scientists. One example is the study of biomedical pathways (e.g. for

Need citation here for Mach

---

<sup>1</sup>See for example, [27], which has been reprinted in several anthologies of Einstein’s essays, including [28] and [29].

metabolism). One view of pathways is that they do not exist as such but are only convenient constructs to organize information about biochemical reactions. Others may take the view that such pathways represent fundamental principles of operation of living cells and organisms. Both views have a place in biomedical informatics.

In fact, three views of knowledge representation are possible. One is the view of Mach, mentioned above, which is knowledge representation as information management. If the focus is on organizing what is known (or believed) for efficient retrieval and communication, it may be acceptable to have contradictory assertions in the knowledge repository. The human users of such systems would make judgements about what to do in cases of conflicting information. Information retrieval is dealt with in more depth in Chapter 5. A second view, which will extensively be developed here, is that of knowledge representation ideas and methods as the basis for constructing biomedical theories. In this context, a particular theory must be constructed free from contradictions in order to be able to unambiguously determine the predictions of the theory. When there are competing ideas, we construct multiple theories, and separately compute their predictions. The experimental facts will ultimately decide which theory is right (of course they may both be wrong). Finally, there is the view of knowledge representation as a model of human activity or human intelligence, i.e., modeling what people believe or do. This is the point of view of much research in the field of Artificial Intelligence. The goal is to understand the nature of intelligent thinking, not so much to create a model of the world of things. The only area in this book where we address this is in Chapter 4, where we present decision modeling that incorporates preferences and beliefs rather than truths about biomedical entities and relationships.

Thus, in this chapter, rather than settling this philosophical debate, we will develop the ideas, methods and formalisms from which biomedical theories may be constructed. This in turn provides the foundation on which many applications described in Part II of this book have been built. Logic is at the base of the rest of the formalisms that are widely used today so it is the first topic to be developed here. Biomedical knowledge also involves cataloging and describing relations between categories, so systems for representing categories and relations come next, in particular, semantic nets, frame formalisms, and description logics. Since so much of biomedical computation involves traversing graphs, implicitly or explicitly, the next topic is methods for search of knowledge networks or graphs. Also included here is some discussion of classification trees<sup>2</sup>, modeling temporal processes, and simulation systems using state machines and other related ideas.

### 3.1 Biomedical Theories and Computer Programs

Formally speaking, a theory is a set of statements called “principles,” or “axioms,” and a reasoning method from which you can infer new statements from the principles or axioms, given specific assertions, or facts, about a particular case. In sufficiently rich theories, one may also infer general statements that are consequences of the

---

<sup>2</sup>they are sometimes called “decision trees”, though that is a misnomer, since it invites confusion with Decision Theory and Bayes Nets, which will be covered in Chapter 4.

axioms. Such derived generalities are called “theorems” in a mathematical context. These in turn can be applied to specific cases just as if they were axioms. The language in which the statements are expressed is characteristic of the topic or domain. In physics, algebraic symbols are used to represent variables or elements of the theory, since their values may vary with time and position. A frame of reference is needed to relate the symbols to things we see or otherwise experience. The axioms or laws are represented by equations that relate quantities to other quantities. The methods of algebra, geometry, calculus and higher mathematics can then be used to manipulate the equations using some particular values for the symbols, and arrive at other values.

### 3.1.1 A World Class Reasoning Example

A remarkable example of the power of mathematics and reasoning is the method by which Eratosthenes determined the circumference of the Earth [93, Volume 1, pp. 206–208]. The diagram in Figure 3.1 shows the geometric and physical picture from which the reasoning follows.

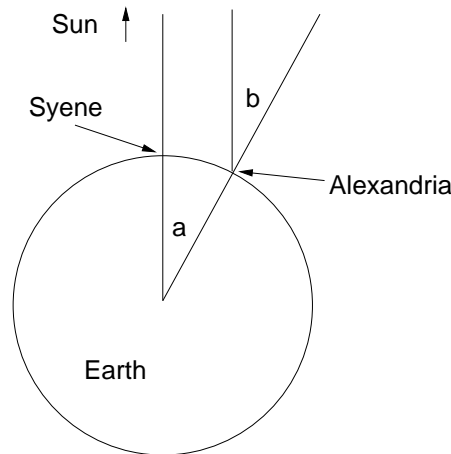


Figure 3.1: Diagram showing the sun’s rays striking the Earth (vertical lines) at Syene and Alexandria, in Egypt. The angles **a** and **b** are equal, and measured in Alexandria (**b**) to be about  $7^\circ$ .

The argument of Eratosthenes is as follows:

1. Axiom: light from the Sun comes to the Earth in parallel rays,
2. Axiom: the Earth is round (remember, this took place around 200 B.C.E.),
3. Fact: at Noon in the Summer, sundials cast no shadow in Syene (in Egypt),
4. Fact: at Noon in the Summer, sundials *do* cast a shadow in Alexandria (also in Egypt), about  $7^\circ$  off of vertical,

5. Fact: the distance from Alexandria to Syene is about 4,900 stadia (a stadion is an ancient unit of distance, corresponding roughly to about 160 to 180 meters)

By Euclidean geometry, the angle subtended by the arc from Alexandria to Syene (as measured from the center of the Earth) is the same as the angle of the Sun from vertical (by a theorem from Euclidean geometry: when a line intersects parallel lines, corresponding angles are equal).

So the distance from Alexandria to Syene represents about  $7^\circ$  of arc length, from which Eratosthenes calculated about 700 stadia per degree. Then the circumference of the Earth is simply  $700 \times 360$  or about 252,000 stadia.

The size Eratosthenes used for a stadion is not known exactly. Eratosthenes' result was somewhere between 39,690 and 46,620 km. The current best value is about 40,000 km.

What would it take to represent the reasoning of Eratosthenes in a computer program? Certainly the arithmetic is easy. What about the logic that led to the right arithmetic formula? That requires a computer program that can implement symbols representing the propositions, methods for combining the symbols in abstract formulas, and finally a proof method to derive the final formula (i.e., deriving theorems or results from axioms and other assertions).

\*\*\* Add a paragraph on the importance of abstraction and approximation to this derivation, BUT also the need for consistency and sound reasoning - cite "Paradox Lost and Paradox Regained" in World of Mathematics, Volume 3

### 3.1.2 Biological Examples

Physiological systems can be described by mathematical variables, the general knowledge about such systems as equations, and the solution of those equations for a particular individual case may be compared to measurements for that case. The measurements are the data, the formulas or equations are the knowledge.

\*\*\* physiology example here. . .

However, much of what we know in biology and medicine is not quantitative, as it is for the subfield of physiology. Even in biochemistry, where we might also have equations relating concentrations of biochemical compounds, more often the form of the equations is unknown, or the rate constants are unknown. Nevertheless, what *is* known is that certain molecules react with certain other molecules in various ways, i.e., we know the reactants, products and catalysts, but not the numerical details. Is this useful? It turns out that catalogs of such reactions and indeed whole biochemical pathways are very useful for answering important biological questions. For example, the EcoCyc knowledge base [72] contains the complete metabolic pathways of the *E. coli* organism, in symbolic form. Figure 3.2 shows a diagram of an example pathway, glycolysis, which transforms glucose to pyruvate. This

pathway is taken from the EcoCyc knowledge base, but also appears in many other organisms.

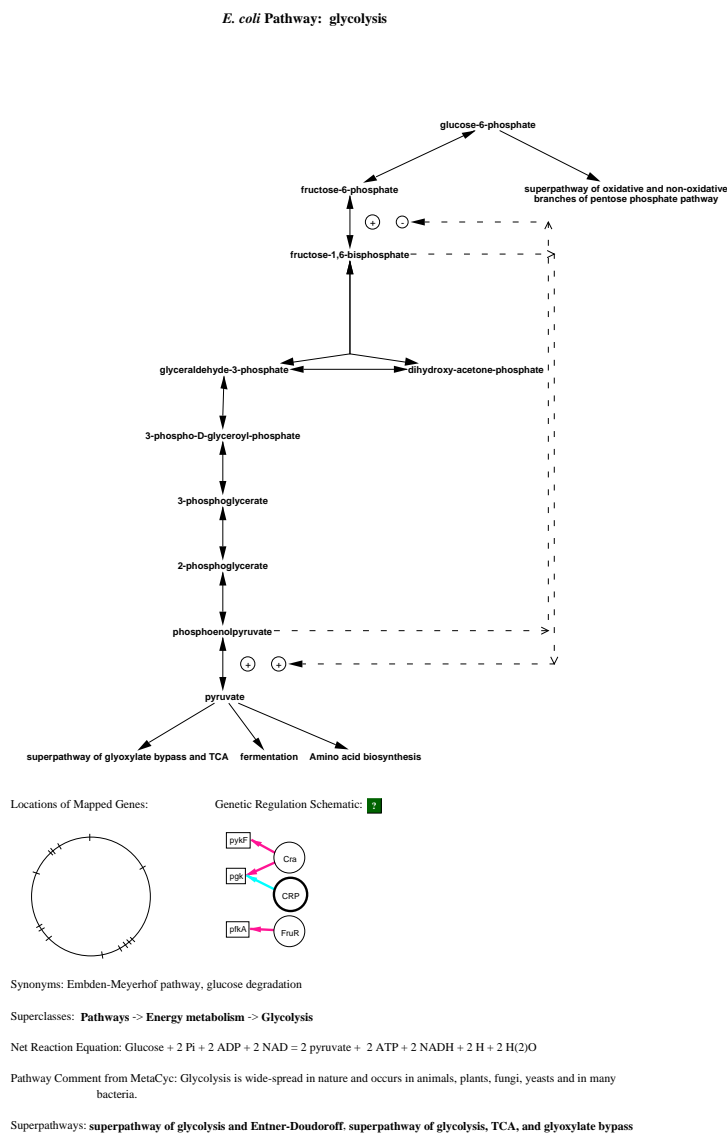


Figure 3.2: A diagrammatic representation of the glycolysis pathway in E.Coli

Such a collection of pathway representations can answer a question like: “can E. coli grow in a medium consisting of the following material...?” It is a graph search problem. Are there pathways that start with the given nutrients and end

(individually or in combination) with all the necessary compounds for the organism to build proteins, etc. that it needs to survive? In Chapter 8 we develop this idea in detail.

### 3.1.3 Symbolic Theories

The variables in some theories thus take on discrete values rather than continuous ones, or sometimes just one of the values “true” and “false.” Functions can also have discrete values in their range as well as their domain. If a function’s range is the set consisting only of the values “true” and “false,” it is called a *predicate function*, or simply, *predicate*. We will still be able to write relations between these variables, functions and values, and in some sense solve the “equations” that we write as the axioms of our theory. These ideas form the basis of *first order logic*, abbreviated as “FOL”.

The discussion of the nasal cavity tumor that may metastasize (in Chapter 2, page 86) can be put in the form of first order logic. We define the location function, a function of two variables, called “loc”, such that if  $\text{loc}(x, y)$  has the value “true” (which we represent by the symbol  $\mathfrak{t}$ ), then the object denoted by  $x$  is located at the anatomical site  $y$ . Similarly the lymphatic-drainage function, called “drains”, has two variables, an anatomic location and a lymphatic tree or chain. If  $\text{drains}(x, y)$  has the value  $\mathfrak{t}$  that is interpreted to mean that the lymphatic chain  $y$  drains the anatomic location  $x$ . Finally we define the metastatic risk function, called “risk”, a function of one variable, a lymph node. This means that  $\text{risk}(x)$  has the value  $\mathfrak{t}$  if the lymph node is at risk of metastatic disease. Then the first axiom of our theory may be written as a logic formula (the notation is described in more detail in Section 3.2):

$$\forall x, y, z \quad \text{tumor}(x) \wedge \text{loc}(x, y) \wedge \text{drains}(y, z) \rightarrow \text{risk}(z) \quad (3.1)$$

Just as before,  $x$ ,  $y$ , and  $z$  play the role of variables. Here, however, their values are not numbers but *symbols* representing things in the world, like a particular tumor, an anatomic location (from the FMA, perhaps, another way in which the FMA appears as a collection of data, rather than knowledge), and a lymph node (also from the FMA).

\*\*\* need to write out the following, not distinct enough in data chapter to cite here

In the example in Section 2.7, we have a particular application of the “loc” function and the “drains” function:

$$\text{loc}(T4, NC) = \mathfrak{t} \quad (3.2)$$

$$\text{drains}(NC, RP) = \mathfrak{t} \quad (3.3)$$

where  $T4$  is the tumor,  $NC$  is the nasal cavity, and  $RP$  is the retropharyngeal node. Then we can infer from formula 3.1 the result

$$\text{risk}(RP) = \mathfrak{t}. \quad (3.4)$$

What about the functions? How are they defined? The “drains” function can be implemented as part of an anatomy model such as the FMA. This is possible because an ontology like the FMA serves not just as a standardized vocabulary, but also contains relations between the terms, such as **contained-in**, **part-of** and **lymphatic-drainage**, as well as subsumption relations (one thing is a specialization of another, sometimes called an **is-a** relation). From this perspective the FMA represents knowledge, not just data.

In this simple example, it is a lot of notation for something easy to glean from reading the textual version of the knowledge. But even in this example, it is known that tumor cells may spread through lymphatic vessels for some distance beyond the initial drainage sites. Thus, to predict the extent of metastatic (but unseen) disease, one must know or look up many details of connectivity of the lymphatic system. When the number of formulas gets only a little larger and the number of anatomic locations and lymph nodes also gets large, it will pay to formalize the knowledge. Why? Once we have a symbolic form for it, we can have a computer do the work. Even better, if the symbolic language we use for formalizing the data and knowledge is a computer interpretable language (a programming language), we can eliminate one translation step.

The EcoCYC knowledge base mentioned earlier is another example of a symbolic theory. It is a theory in the sense that it represents a collection of known biochemical principles, basic knowledge about biochemical compounds, and a collection of known biochemical pathways. By applying reasoning principles or methods, one can specify a particular problem or condition and then predict the behavior of the *E. Coli* organism under that condition.

One can go further and predict possible pathways by searching at a more detailed level, chaining together known biochemical reactions to see if there is a way to, for example, metabolize glucose, i.e., manufacture other useful molecules from it, with the available enzymes and reactants. The Pathminer project illustrates an approach to this idea [86]. These and other examples are developed in Chapter 8. Pathminer uses A\* search, described in Section 3.5.

\*\*\* This idea of “theoretical biomedicine” needs to be explored more deeply. How does a biomedical informatics theory differ from a physics, chemistry or biology explanation of a biomedical phenomenon? Lots of specific assertions rather than a few general rules, also in medicine (and biology) lots of uncertainty about the rules, lots of lore rather than believable formulas, and also in medicine the practice context.

Expressing ideas in biomedicine as computer programs gives three benefits. One is that a program gives precision to the theory, so that its properties can be formally examined, deduced, and discovered. It is possible, for example, to determine through the formalism if the proposed theory is self-consistent or not. The second is that the proofs, derivations, and calculations that in the physical sciences until recently have been done by hand (the algebraic and other mathematical manipulations of the formulas, not the computation of specific numeric results) can be done automatically by the computing machinery. Third, the program can in principle be

incorporated into a practical system for use in the laboratory, the clinic, or in other biomedical contexts.

These are lofty goals. The extent to which they are easy or difficult to achieve will be seen in the following sections. The citations and summary at the end of this chapter provide an entry to research aimed at extending the ideas and solving the difficult problems.

## 3.2 Logic and Inference Systems

Logics are formal languages for representing information and knowledge such that new assertions can be inferred from the initial knowledge, assertions that were not part of the initial information or knowledge. Many different kinds of logic systems can be invented, each with a range of capabilities and limitations. The most widely used and studied, propositional calculus and first order logic, will be described here.

A logical system consists of a collection of statements, or sentences, made up of symbols that have some predefined meanings. In order to make it possible to do automated deduction and analysis, we have to specify what constitutes a sentence and what is not a sentence, just as in ordinary written languages. The allowed symbols constitute the *vocabulary* of our logic. The rules for constructing grammatically correct sentences specify the *syntax*. In order to make any practical use of the logic system, we need to specify also an *interpretation* of the symbols, essentially a mapping of symbols to things in the world of our experience. This mapping is also called the *semantics* of the logic system.

Some of the symbols will have meanings that can be completely specified independently of interpretation of the other symbols. These are the familiar *connectives*, such as *and*, *or*, and *not*, and (when dealing with variables) *quantifiers*, like *for all* and *there exists*. For example, if  $A$  and  $B$  are sentences that could be either true or false, then  $A \wedge B$  is also a sentence, where the symbol  $\wedge$  represents the “and” connector. The meaning of  $\wedge$  is that if the sentence  $A$  is true and  $B$  is also true, the compound sentence is true, otherwise the compound sentence is false. This only depends on the truth value of the two terms,  $A$  and  $B$ , not on their actual interpretations or meanings.

In any logic system there are also *primitives*, whose meaning depends on an interpretation. Because of this, a sentence that is true in one interpretation may well be false in another. For example, the statement, “Mach is the best in the class” could be either true or false, depending on what the terms “Mach”, “best” and “class” refer to. One interpretation is that Mach is the famous 19th century physicist, “class” refers to a mathematics class he enrolled in, and “best” compares his numeric grade in the class to that of the other students. I don’t know if this is true; it certainly could be false. On the other hand, in more modern times, this sentence could refer to Mach, a variant of the Unix<sup>TM</sup> operating system, and the class referred to is the class of Unix<sup>TM</sup> variants. “Best” could mean many things, and this too could be either true or false. In particular, in one of the two interpretations, the sentence could be true and in the other, false. So, also, with formulas in our logic systems. The overall plan for building logical theories is to



start with sentences that are true in the interpretation of interest (these are *axioms*), and then by a reliable method of proof, find out what other sentences are true in that interpretation (these are the *theorems*).

Thus a logic consists of:

- a set of primitives,
- ways of building formulas (connectives, quantifiers, punctuation), and
- inference rules (how to derive new formulas).

### Predicate calculus

A simple logic in which to illustrate the ideas is *predicate calculus*, sometimes also called *propositional calculus*.

The standard connective symbols are:

$\vee$	or (inclusive)
$\wedge$	and
$\neg$	not
$\rightarrow$	implies

Using these symbols and also symbols for primitives, we can construct sentences in our logic, called “Well Formed Formulas” (sometimes abbreviated as “wff”). Here are the rules for constructing a wff from primitives, connectors and other wffs:

- Every primitive symbol is a formula
- If  $\alpha$  is a formula, then  $\neg\alpha$  is a formula
- If  $\alpha$  and  $\beta$  are formulas, then  $\alpha \wedge \beta$ ,  $\alpha \vee \beta$ , and  $\alpha \rightarrow \beta$  are all formulas.

In cases where combining formulas with connectors might result in ambiguity as to the meaning (for example because the resulting series of symbols might be grouped in different ways), parentheses are used to make clear the order in which the connectors are applied. For example,  $(\neg\alpha) \vee \beta$  is not the same as  $\neg(\alpha \vee \beta)$ .

The language of logic is compositional, i.e., the meaning and value of compound formulas depends on the meaning of the individual terms. When a term appears in several places in a single formula, it is required to have the same interpretation everywhere. For the relation  $P \rightarrow Q$ , for example, we can assign truth values that depend only on the truth values of the individual primitives,  $P$  and  $Q$ ,

		Q	
		T	F
P	T	T	F
	F	T	T

and similarly for  $P \vee Q$  and  $P \wedge Q$  and  $\neg P$ .

Here are some useful relations in predicate calculus:

- $\alpha \rightarrow \beta$  is equivalent to  $(\neg\alpha) \vee \beta$  (prove with truth tables)
- DeMorgan's Laws

$$\neg(\alpha \wedge \beta) \text{ is equivalent to } (\neg\alpha) \vee (\neg\beta)$$

and

$$\neg(\alpha \vee \beta) \text{ is equivalent to } (\neg\alpha) \wedge (\neg\beta)$$

Also, there are distributive, commutative, and associative laws:

- the Distributive Laws

$$\alpha \wedge (\beta \vee \gamma) \text{ is equivalent to } (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$$

and

$$\alpha \vee (\beta \wedge \gamma) \text{ is equivalent to } (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$$

- the Commutative Laws

$$\alpha \vee \beta \text{ is equivalent to } \beta \vee \alpha$$

and

$$\alpha \wedge \beta \text{ is equivalent to } \beta \wedge \alpha$$

- the Associative Laws

$$\alpha \wedge (\beta \wedge \gamma) \text{ is equivalent to } (\alpha \wedge \beta) \wedge \gamma$$

and

$$\alpha \vee (\beta \vee \gamma) \text{ is equivalent to } (\alpha \vee \beta) \vee \gamma$$

### Truth and consequences

An *interpretation* for a set of formulas assigns a meaning to each term and function name. The meanings determine whether each formula is true or false in that interpretation. In a set of formulas (we sometimes call such a set a “knowledge base”), when a primitive appears in different formulas, its meaning should be the same everywhere. This enables us to determine what else is true if all the sentences in the set are true.

- If a sentence is true in an interpretation, we say it is *satisfied* in that interpretation.
- A sentence  $\alpha$  is said to be *entailed* by a set of sentences  $S$  if  $\alpha$  is true in every interpretation in which all the sentences in  $S$  are true.

Determining entailment is a difficult problem. Generally it involves computing and examining truth tables for every combination of primitive terms that appear in the set of sentences, and that make the sentences true.

\*\*\* need some examples here of computing entailment, perhaps the transitivity of implication?

### Inference rules

Since entailment (using interpretations or truth tables) is so difficult, it is important to find more reasonable strategies for determining it. Such strategies are called *inference procedures*. Inference procedures are purely formal manipulations of formulas from which we derive new formulas. If we can depend on them to produce only formulas that are true, when we start from a set of formulas that are true (assuming of course that the original set is consistent), we say that the inference procedures are *sound*.

The following are some sound inference procedures for propositional logic.

- Modus ponens:

From the two formulas,  $\alpha$  and  $\alpha \rightarrow \beta$ , we can infer  $\beta$

- Resolution:

From  $\alpha_1 \vee \dots \vee \alpha_n \vee \gamma$  and  $\beta_1 \vee \dots \vee \beta_m \vee \neg\gamma$

we can infer

$$\alpha_1 \vee \dots \vee \alpha_n \vee \beta_1 \vee \dots \vee \beta_m$$

It might appear that one can write inference rules as sentences *within* the logic system. For example, one might think that modus ponens is equivalent to the wff  $\alpha \wedge (\alpha \rightarrow \beta) \rightarrow \beta$ , but this formula, even though it is always true, is *not* the same as the assertion that we can *add* the *formula*  $\beta$  to the collection.<sup>3</sup> How do we then determine whether a particular inference procedure is sound or not? Indeed there is a connection with the reformulation of inference rules as formulas *within* the system. If such a formula can be shown to be true no matter what the interpretation is, then the inference rule corresponding to it is sound [131].

---

<sup>3</sup>A droll treatment of this matter, by Lewis Carroll, can be found reprinted in Hofstadter's "Gödel, Escher, Bach: an Eternal Golden Braid", [52, pages 43–45]

### Theorems and Proofs

Entailment is *not* the same as proof. Entailment concerns what is actually true as a consequence of a collection of sentences being true. It is conceivable that there may be truths that follow from a collection of sentences, but there is no way to prove them (in the sense we usually mean by proof, using sound inference procedures, not by checking truth tables). Furthermore, we may invent proof methods that are not reliable, in that they may allow us to find proofs for sentences which are *not* entailed by the given sentences.

In the following, we are starting with a set of *axioms*, which are formulas that are assumed to be true in the (usually biomedical) interpretation of interest, and which form the basic assertions of our knowledge base (or, put another way, our biomedical theory). The goal is to compute (eventually by an algorithm that can be implemented in a computer program) the entailments of such a set of axioms (usually together with some additional assertions about a particular problem case). Note that an axiom is *not* the same as an assertion that may be true (entailed by the other assertions) but cannot be proved. An axiom is also different from a *definition*, which serves to associate a name with some conditions, entities or both. Definitions say nothing about the world, but axioms do.

- A *proof* is a sequence of formulas, where the next formula is derived from previous ones using sound inference rules.
- A *theorem* is a formula that can be derived (proved) from a set of *axioms*.
- A *tautology* is a formula that is true regardless of the interpretation. An example is  $A \rightarrow (A \vee B)$ , but  $A \rightarrow B$  is not a tautology. (A tautology is sometimes called a *valid* formula.) Sound inference rules correspond to tautologies. From above we have the example,  $\alpha \vee (\alpha \rightarrow \beta) \rightarrow \beta$ .

It is important to note that a tautology is simply true. An axiom is a formula that you assert to be true as part of a theory about something in the world. It might not actually be true, but it might also not be possible to do an experiment or measurement to determine whether the assertion is true. In that case, the theorems that follow from that and the other axioms are used to test the theory. If a theorem's statement is in disagreement with the observed world, one or more axioms must be wrong too.

\*\*\* can we give an example here? maybe just a problem statement, leaving the proof to the next section? Modus ponens equivalent wff as a tautology? Also transitivity of  $\rightarrow$ ? (and its equivalence to resolution?)

\*\*\* discussion about completeness and decidability here? Propositional logic is complete and decidable, FOL is complete but undecidable, and arithmetic is *incomplete* and undecidable. Forward ref to Completeness section

\*\*\* brief mention of control strategies, clauses, Horn clauses, CNF, backward

chaining, forward chaining, SLD resolution here, and forward reference Section 3.2.2, where we will put code for each, examples, like the MEBI 550 homework from BL

\*\*\* note that recursive is not equivalent to exponential and iterative is not equivalent to linear. Factorial function and Tower of Hanoi as examples

### Unsound inference: abduction and induction

\*\*\* develop this paragraph into a fuller subsection.

From  $P$  and  $P \rightarrow Q$  we can infer  $Q$ . What about the other way around? From  $Q$  and  $P \rightarrow Q$  can we infer  $P$ ? This is known as “abduction”, or abductive reasoning. Where in biology, medicine or public health might this apply? From  $P$  and  $P \rightarrow Q$  we can infer  $Q$ . What about inferring the  $P \rightarrow Q$  relation when we observe both  $P$  and  $Q$ ? This is known as “induction”, or sometimes in a more general form, machine learning. Where in biology, medicine or public health might this apply?

#### 3.2.1 First order logic

First order logic (FOL) extends predicate calculus by introducing the idea of a variable. Variables are symbols that can take on values representing individuals, just as in ordinary algebra. Then, in addition to writing formulas expressing assertions about individuals, we can write more general formulas expressing relations among classes of individuals.

Primitives in first order logic can be:

- constant symbols (individuals)
- function symbols
- predicate symbols

A function symbol names a function, whose value depends on the “inputs” to the function, which can be named individuals, or variables. Predicate symbols name special functions whose values can be only the values “true” and “false”.

Quantifiers allow iteration over individuals that are referred to by variables. There are two kinds of quantifiers:

$\forall$	for all
$\exists$	there exists

Using quantifiers, we can construct a much wider range of wffs.

\*\*\* explain the meaning in terms of interpretations, and provide biomedical examples

\*\*\* discuss inference in FOL, tractability clearly much worse, since may have

an infinite number of cases; see BL 2.3.3

A *sentence* is a formula with no unquantified variables. The same (quantified) variable can appear in many different sentences, and in each case it may take on values independently of values it takes on in the other formulas. What this means is that we can “reuse” variable names when writing many formulas. However, *within a single formula or sentence*, the variable’s value must be the same everywhere (in that sentence).

### Quantifier order

Changing the order of universal quantifiers does not change the meaning of a sentence:

$$\forall x \forall y : \text{likes}(x, y) \rightarrow \text{buys-present}(x, y)$$

$$\forall y \forall x : \text{likes}(x, y) \rightarrow \text{buys-present}(x, y)$$

The same is true of existential quantifiers:

$$\exists x \exists y : \text{glass}(x) \wedge \text{color}(x, y)$$

$$\exists y \exists x : \text{glass}(x) \wedge \text{color}(x, y)$$

Changing the order of a universal with an existential *does* change the meaning:

$$\forall x \exists y : \text{likes}(x, y)$$

$$\exists y \forall x : \text{likes}(x, y)$$

and similarly within rules:

$$\forall x \exists y : \text{glass}(x) \rightarrow \text{box}(y) \wedge \text{inside}(x, y)$$

$$\exists y \forall x : \text{glass}(x) \rightarrow \text{box}(y) \wedge \text{inside}(x, y)$$

### Quantifier movement

You can generally move quantifiers to the left of a formula:

$$\forall x : \text{rock}(x) \rightarrow \forall y : \text{as-heavy-as}(x, y)$$

$$\forall x \forall y : \text{rock}(x) \rightarrow \text{as-heavy-as}(x, y)$$

$$\forall x : \text{glass}(x) \rightarrow \exists y : \text{box}(y) \wedge \text{inside}(x, y)$$

$$\forall x \exists y : \text{glass}(x) \rightarrow \text{box}(y) \wedge \text{inside}(x, y)$$

But you can *not* move quantifiers past a negation:

$$\forall x : \text{rock}(x) \rightarrow \neg \exists y : \text{heavier-than}(y, x)$$

$$\forall x \exists y : \text{rock}(x) \rightarrow \neg \text{heavier-than}(y, x)$$

The solution - move the negations to the atoms, changing any quantifiers they pass:

$$\forall x : \text{rock}(x) \rightarrow \neg \exists y : \text{heavier-than}(y, x)$$

$$\forall x : \text{rock}(x) \rightarrow \forall y : \neg \text{heavier-than}(y, x)$$

$$\forall x \forall y : \text{rock}(x) \rightarrow \neg \text{heavier-than}(y, x)$$

\*\*\* change all the following to biomedical examples

**Formulas involving cats and fish**

Here we just provide a simple example of some assertions about some common knowledge from the animal world.

1.  $\forall x \forall y : \text{cat}(x) \wedge \text{fish}(y) \rightarrow \text{likes}(x, y)$
2.  $\forall x : \text{calico}(x) \rightarrow \text{cat}(x)$
3.  $\forall x : \text{tuna}(x) \rightarrow \text{fish}(x)$
4.  $\text{tuna}(\text{Charlie})$
5.  $\text{calico}(\text{Pudge})$

The following are some additional sound inference procedures that apply to first order logic.

- Universal specialization:

Replace variable with individual, so from  $\forall x : p(x)$  we can get  $p(A)$ , where  $x$  is a variable, and  $A$  is some particular entity.

- Unification: essentially pattern matching and variable substitution

An example involving bacteria, not entirely biologically accurate is the following. These sentences form a collection of axioms, a simple theory of bacterial infection and treatment.

- Bacteria are either Gram positive or Gram negative (the Gram stain is a procedure that colors bacteria either pink or deep purple, depending on the permeability of their cell membranes).
- There are antibiotics for treating infections from Gram positive bacteria.
- There are antibiotics for treating infections from Gram negative bacteria.

This set of axioms entails the statement that all bacterial infections are treatable, i.e., no matter what organism attacks, there is an antibiotic treatment for it. However, this statement cannot be proven by forward chaining or backward chaining, but it is provable by resolution.

\*\*\* show m.p. and resolution proof examples, also mention Boyer-Moore theorem prover

### Completeness

FOL is complete: if a set of sentences  $S$  entails another sentence  $R$ , then a proof of  $R$  exists. For a given set of axioms, a procedure exists that can find a proof if it exists. However, no automated procedure (reasoner) can exist that is guaranteed to terminate if the proof does not exist. Since no automated reasoner can be both sound and complete, we say that FOL is *undecidable*. Arithmetic is *worse*. Any system as complex as arithmetic is guaranteed to have a proposition that is entailed by the system but for which a proof *cannot* exist. (add historical refs, e.g. to Gödel and others)

What is the significance of these things for biomedicine?

### Some quotes

The grand aim of science is to cover the greatest number of experimental facts by logical deduction from the smallest number of hypotheses or axioms.

- Albert Einstein

There is something fascinating about science. One gets such wholesale returns of conjecture out of such a trifling investment of fact."

- Mark Twain, "Life on the Mississippi", 1874

## 3.2.2 Rule-based Programming and Proof Strategies

\*\*\* describe here in more detail control strategies, clauses, Horn clauses, CNF, forward chaining, backward chaining, SLD resolution and code for each, including examples, like the MEBI 550 homework from BL, IK predicate calculus Horn clause representation and translator to BL data structures

\*\*\* Production rules and related stuff, so-called "expert systems".

Introduce Prolog, and the usual control strategy for Prolog.

Describe Graham simple mock Prolog, reference to Norvig for more sophisticated version. Include here discussion of how to initialize a KB from CLOS instances, with and without the instance tracking metaclass. Develop an example app with Graham's inference code.

My supplements to Graham's code:

The `<--` macro is like `<-` but replaces the hash table entry rather than adding it.

```
(defmacro <-- (con &optional ant)
  '(length (setf (gethash (car ',con) *rules*)
    (list (cons (cdr ',con) ',ant)))))
```



The `assert-value` function converts an object value pair to an assertion. Note that the clauses in the assertions and rules will *not* be evaluated.

```
(defun assert-value (pred obj &optional val)
  (if val
    (eval '(<- (,pred ,obj ,val)))
    (eval '(<- (,pred ,obj)))))
```

The `replace-assert-value` function is just like `assert-value` except that it replaces previous values for that predicate

```
(defun replace-assert-value (pred obj &optional val)
  (if val
    (eval '(<-- (,pred ,obj ,val)))
    (eval '(<-- (,pred ,obj)))))
```

Here's a way to make the connection between slot values in CLOS classes and mock Prolog facts.

```
(defun assert-slot (slot obj &optional replace)
  "deals with slots that have a single item in them"
  (if replace (replace-assert-value slot obj (funcall slot obj))
    (assert-value slot obj (funcall slot obj))))
```

\*\*\* explain the difference between these two...

```
(defun assert-list-slot (slot obj &optional replace)
  "deals with slots that have a list of items in them"
  (dolist (item (funcall slot obj))
    (if replace (replace-assert-value slot obj item)
      (assert-value slot obj item))))
```

\*\*\* Also include something about KIF, and other RBS implementations, e.g., CLIPS, JESS, etc., also about storage, transmission and retrieval, including cross ref to Chapter 5

\*\*\* write something about storage of rules in KBs, analogous to storage of facts in tables (relational databases), tagged representations (XML), structures (OODB), and for rules we have KIF, OWL (DL), etc.

### 3.2.3 Limitations of FOL

As expressive as First Order Logic is, many important queries cannot be expressed in this formalism. Moreover, many areas of biomedical knowledge do not seem

a natural fit to the logic formalism. For example, one might have a set of logic formulas describing parts of the human body. The following formulas assert that the heart is an organ, as are the lung, and the kidney. Here we are considering the predicates `heart`, `lung` and `kidney` to define the corresponding classes. The variables can have values which refer to particular organs in a particular person, e.g., my heart or yours.

```
(<- (organ ?x) (heart ?x))
```

```
(<- (organ ?x) (lung ?x))
```

```
(<- (organ ?x) (kidney ?x))
```

We can also express `part-of` relations in the same way.

```
(<- (part-of ?x ?y) (and (right-atrium ?x) (heart ?y)))
```

One such relation will be required for every part. Not only is this an extremely verbose way to manage this information, it also does not allow for queries about the kinds, or predicates, themselves. One cannot, for example, ask in such a knowledge base, “What are the parts of the heart?”, or “What kinds of organs are there?” Such questions are not expressible in First Order Logic, as they are queries about predicates, not about individuals.

What about defaults?  
maybe just leave it out?

One way to handle such queries is to create a new representation language in which the categories, individuals and relations (predicates) are all on an equal footing. In this new language there are a limited number of operators (analogous to predicates in FOL) which connect categories, individuals and relations with themselves and each other. The queries that were higher order in the old representation will become first order in this new language. This approach is developed by Norvig [94, Chapter 14]. An alternative is to develop an entirely different approach, a *frame* formalism, which relies on structured descriptions. This has the virtue of being a good match to large bodies of biomedical symbolic knowledge. Norvig’s development also shows how to construct a frame language by providing translations into his relational language. Our approach will be more along the lines of the informal description of frames in Brachman and Levesque [13, Chapter 8].

### 3.3 Frames, Semantic Nets and Ontologies

In large data stores such as GENBANK, the sequences are often thought of as the primary data, and are the subject of much computation (sequence comparisons, sequence searching for small patterns like motifs). However, there is a lot of knowledge in the annotations, the text information that accompanies each sequence. This is generally not in an easily computable form, but it is computationally powerful material. For example, information about which protein(s) a gene encodes, and what the function(s) and pathways in which it participates is useful for reasoning about cellular metabolism. Projects like the Gene Ontology [5] are a step towards

making these pieces of knowledge the basic elements of a computational scheme. Another example is the EcoCyc ontology of metabolic pathways in *E.coli*, and its generalization to many other organisms [72, 71]. The information represented here is symbolic, to be manipulated as such. When an enzymatic reaction is specified in a system like EcoCyc, the symbols and structures represent objects such as a molecule, a particular kind of molecule, a reaction (a relation between molecules as reactants and products). As Karp [73] points out, such structures enable computations that are not possible by searching text or XML data, or using controlled vocabularies.

Similarly, knowledge about laboratory tests and their clinical interpretation is not part of the data or the coding system. This needs to be represented in a different way, which captures the meaning and relationships of the entities being measured, diseases, diagnostic significance, etc. The Unified Medical Language System is an NLM sponsored project that addresses this goal.

Much effort in bioinformatics is now being expended to extract this symbolic knowledge from the text of published articles, and from data resources that have this knowledge embedded as text. Rather than address the issues of parsing and interpreting this text corpus, we will focus on building a representation language for it. We need a language that will incorporate the idea of classification of entities, particularly hierarchical classification systems that are usually referred to as “ontologies”. After a brief introduction to frame systems and the use of classes in object-oriented programming languages as a more limited alternative, we will see what kinds of computations can be done on these ontologies.

need cite here, also maybe expand this a bit

### 3.3.1 A Simple Frame System

In this section we develop a really simple frame implementation to illustrate essential ideas. While frames can be used to represent data, the emphasis here is on frame systems as knowledge representation languages. They are an example of a variety of knowledge representation systems called “slot and filler” systems, since the basic structural idea is to group together names of attributes and their values (of course the values can be other frames).

What kinds of information do we want to represent? For any entity in the world, whether an individual thing (a laboratory test for a particular patient), or a category representing many individuals all of the same kind (organ, heart, protein, etc.), we will need to encode a name, perhaps a unique identifier, relationships to other entities such as class–subclass relationships and others, properties of the category as a whole, and properties of specific individuals belonging to the class.

For example, for the category, “heart”, we would like to represent its relation to the rest of the body, its parts, and a text description. A complete representation would also include a specification of the blood vessels (arteries) that supply the heart, the veins that return blood from the heart muscle, its connection to the rest of the circulatory system, its lymphatic drainage (what lymph vessels collect the fluid around the heart), and what nerves are connected to the heart muscle. For a specific heart, we would want to be able to record some of its individual properties, such as heart beat rate (pulse), ejection fraction (how much of its volume is it

actually pumping out) and how big is it. For a specific heart we would likely measure the electrical activity (an electrocardiogram, or EKG).

The basic unit of representation of entities in the world is the *frame*. Our frame system will use keyword-value pairs and other kinds of tagging as in the patient example in Section 2.2.1 (page 49). However, here we introduce some additional structure, to allow for abstraction of the computations we will perform. A frame can have *slots*, which are accessible by their names, and which may contain associated values, as well as other properties. Each frame needs to have a unique *identifier* (within the context of the knowledge base, not a UID in the global sense). Frames may have some slots that are understood to have special or designated semantics. We will start with a simple representation and augment it as we go.

In this simple system, a frame consists of a list, in which the first element is a symbol, the name of the frame, and the remaining elements are “slots”, which initially we will represent as lists containing keyword-value pairs. In a slot, the keyword serves as the name of the slot, and the value associated with that name can be (in general) any valid Lisp data type. A slot value can also be a reference, i.e., the name or identifier of another frame, or the actual frame object itself, or a list of such objects.

Frames can describe individuals or classes, and a frame may describe an entity which is at the same time both an individual and a class. We take the body parts as examples. The class, **heart**, could be described by the frame:

```
(heart
  (part-of cardiovascular-system)
  (contained-in middle-mediastinal-space)
  (has-parts (right-atrium left-atrium
               right-ventricle left-ventricle
               mitral-valve tricuspid-valve ...))
  (arterial-supply (right-coronary-artery
                   left-coronary-artery))
  (nerve-supply ...)
  ...)
```

Each slot represents a relation between the frame and the value in the slot. In the example above, the **part-of** slot and its value are equivalent to the FOL formula (which is a Horn clause):

$$\forall x, y : \text{heart}(x) \wedge \text{cardiovascular-system}(y) \rightarrow \text{part-of}(x, y)$$

This can also be expressed in the style of the Graham inference code.

```
(<- (part-of ?x ?y) (and (heart ?x) (cardiovascular-system ?y)))
```

Similarly, each of the other slots can be represented by a Horn clause, as, for example:

```
(<- (has-parts ?x (right-atrium left-atrium
                  right-ventricle left-ventricle ...))
    (heart ?x))
```

It might be more useful to assert the **has-part** relationship for each part, i.e.,

```
(<- (has-part ?x right-atrium)
    (heart ?x))
(<- (has-part ?x left-atrium)
    (heart ?x))
(<- (has-part ?x right-ventricle)
    (heart ?x))
(<- (has-part ?x left-ventricle)
    (heart ?x))
...
```

An individual person might have a particular heart, as follows:

```
(heart-of-joe
  (belongs-to joe)
  (beat-rate 72))
```

Now that we have a sketch of the frame representation, it would be useful to write functions that can access the contents of slots, create new frames, and do other operations. Before we do so, let's reconsider one element of the frame formalism. The symbol naming the frame should be unique, but we have not provided any way to insure it. The names we have chosen so far are descriptive, and more properly a descriptive name should be a value in a **name** slot of a frame. So, with this change, the **heart-of-joe** frame would look like this:

```
(frame3427
  (name "Joe's heart")
  (belongs-to joe)
  (beat-rate 72))
```

The content of the **name** slot of this frame is a string, but we have not specified any restrictions so far, so the name of a frame could also be a symbol. Which is better will depend on the context of use of the frame system. The symbol identifying the frame, **frame3427**, is a unique symbol that would be generated by our frame system's constructor function (which we will write later). Similarly, the **heart** frame would be rewritten with a unique identifier, **frame1342** in this example, and so on for all the frames. We call this symbol the **frame-id** of the frame.

```
(frame1342
  (name heart)
  (part-of cardiovascular-system)
  (contained-in middle-mediastinal-space)
  (has-parts (right-atrium left-atrium
                right-ventricle left-ventricle
                mitral-valve tricuspid-valve ...))
  (arterial-supply (right-coronary-artery
```

```

                                left-coronary-artery))
(nerve-supply ...)
...)
```

Before delving into the details, we need to decide how to keep all the frames in a readily accessible place, so we can look up particular frames, and resolve cross references. A simple way to do this is to arrange for every frame, on creation, to be added to a global list. We use a named global variable, **\*frames\***, for this.

```
(defvar *frames* nil "The global list of all frames")
```

The constructor functions that make frames and the search functions that look up frames will reference this variable, but it should not be explicitly used in a program, otherwise.

### Accessors and Other Abstraction Devices

While it is possible to write explicit code every time we need to find a slot and retrieve its contents, it is better to provide a little abstraction layer. Here are a few simple functions to access frame contents. First, the **frame-id** function is a simple function to return the frame ID of a frame. The **get-slots** function returns a list of all the slots in the frame. These functions define the overall structure of a frame.

```
(defun frame-id (fr) (first fr))
```

```
(defun get-slots (fr) (rest fr))
```

With the **frame-id** function, we can provide a search function that looks up frames by their **frame-id**.

```
(defun find-frame-by-id (frm-id)
  "returns the frame whose frame-id is frm-id"
  (find frm-id *frames* :key #'frame-id))
```

Next we have to decide on the structure of a slot. A simple approach is to just use the keyword-value pair idea, so that a slot is as illustrated above, a list whose first element is a symbol naming the slot, and the next element in the list is the value stored in the slot. We will augment this later when we add attached procedures, but for now, the following code is sufficient to define the slot abstraction layer. The **slot-name** function just returns the name of a slot. The **contents** function returns the value stored in the slot (in this simple case there is no way to distinguish between **nil** as a value and **nil** indicating no value. Finally, we also provide a nice way to use the **setf** macro to update the data in a named slot.

```
(defun slot-name (slot) (first slot))
```

```
(defun contents (slot) (second slot))
```

```
(defun (setf contents) (newval slot)
  (setf (second slot) newval))
```

Now, having a way to refer to the name of a slot, we can provide the next abstraction layer. In this layer, there are functions to retrieve a slot by name, get a list of all the names of slots in a particular frame, and a next level of accessors for reading and writing slot data. The enhancements to this code that support attaching procedures to slots will go in this layer, not the lowest layer. This makes it easier later to change the underlying representation of slots without changing most of the implementation.

The `get-slot` function returns a particular named slot including its contents. The `slot-list` function returns a list of the symbols naming each of the slots, i.e., it answers the question, “What slots does this frame have in it?” Finally the `slot-data` function and its associated `setf` function, provide the higher level accessor layer we will need for later enhancements.

```
(defun get-slot (fr name)
  (find name (get-slots fr) :key #'slot-name))

(defun slot-list (fr)
  (mapcar #'slot-name (get-slots fr)))

(defun slot-data (fr slot-name)
  (contents (get-slot fr slot-name)))

(defun (setf slot-data) (newval fr slot-name)
  (setf (contents (get-slot fr slot-name))
        newval))
```

It will be useful also to have some named functions to refer to specific slots that appear in most, though perhaps not all, frames. The function `frame-name` returns the contents of the `name` slot, if it is present. Using this, we can search the global list of frames, effectively looking up frames by name.

```
(defun frame-name (fr) (slot-data fr 'name))

(defun find-frame (fr-name)
  "returns the frame whose name slot contains fr-name"
  (find fr-name *frames* :key #'frame-name :test #'equal))
```

While this is not much of an abstraction layer, it provides enough insulation so that we can change the underlying representation if necessary while preserving upper layer code. In particular, we can *extend* the representation (as will be seen in the next sections), without changing the code that uses the abstraction layer. In referring to frames, we may refer to them by their names or by their identifiers, depending on the context.

### Classes and Instances

So far, we have a way to express a lot of biomedical knowledge, but it appears that everything must be explicitly written into each frame. When a class such as `heart`

is defined, it is usually because all hearts share some common properties. Some of these properties have the same values for all hearts, and are properly part of the class description. Such slots belong in the **heart** frame. Some properties are present in all hearts but the values associated with them may vary from individual to individual, such as the heart beat rate. Those slots should appear in all the instances, but it would be economic to be able to specify as a part of the definition of **heart** (i.e., in the **heart** class frame) that these slots should appear in each frame representing an individual heart.

Thus, we need a way to distinguish between frames that represent *classes* and ones that represent *individuals* (usually called *instances*). We also need to be able to express relationships, e.g., of which class is a particular frame an instance. In addition to class-instance relationships, we can have frames that describe specialized versions or subclasses of the things described by a class. These relationships are exactly the relationships among sets, subsets and elements of sets.

To implement these ideas, we introduce two new slots with reserved names, whose meaning will be implemented in code. The **instance-of** slot in a frame, *F*, contains the **frame-id** of the frame describing the class of the object represented by *F*. In other words, if an object, *F*, is represented by a frame with a slot that looks like (**instance-of frame2861**), that means that *F* is a member of the class represented by the frame whose **frame-id** is **frame2861**.

In the example frames involving hearts, we indicate that Joe's heart is an individual with respect to the class **heart** by including an **instance-of** slot containing the identifier of the class to which this instance belongs.

```
(frame3427
  (name "Joe's heart")
  (instance-of frame1342)
  (belongs-to joe)
  (beat-rate 72))
```

Similarly, to express the fact that the **heart** frame describes a subclass of the class named "organ", whose frame-id is **frame3356**, we add a slot named **subclass-of**.

```
(frame1342
  (name heart)
  (subclass-of frame3356)
  (part-of cardiovascular-system)
  (contained-in middle-mediastinal-space)
  (has-parts (right-atrium left-atrium
               right-ventricle left-ventricle
               mitral-valve tricuspid-valve ...))
  (arterial-supply (right-coronary-artery
                   left-coronary-artery))
  (nerve-supply ...)
  ...)
```



In this frame system, the **subclass-of** and **instance-of** slots will get some special treatment by providing functions that operate on their contents. For now, it is sufficient to think of a class as representing a category with some members (or possibly none), in which the class frame describes properties common to all the members of the class. An individual is anything that is a member of a class. It is possible to define classes of classes, so it is possible for something to be both an instance and a class at the same time.

The **frame-type** function returns the name of the class of which its input is an instance, or **nil** if it is not an instance of a class frame. This is not just the content of the **instance-of** slot, because that slot contains a **frame-id**, not a name. So we need to look up the frame and get its name. The **superclass** function returns the name of the frame of which the input is a subclass, or **nil** if the input frame has no superclass. This is just the content of the **subclass-of** slot.

```
(defun frame-type (fr)
  "returns the name of the class frame of which the frame fr is an
  instance"
  (frame-name (find-frame-by-id
                (slot-data fr 'instance-of))))

(defun superclass (fr)
  "returns the frame-id of the first superclass"
  (slot-data fr 'subclass-of)) ;; just returns the first one
```

If we restrict a class to have no more than one superclass, there will be a single **subclass-of** slot. This is called *single inheritance*. However, many biomedical entities we would like to represent belong to overlapping classes, and so the classes that represent them will have multiple superclasses. We can implement this by simply allowing more than one **subclass-of** slot. In general, each slot in a frame should have a unique name, so this is an exception. In the discussion of inheritance we will see what can be done with the class-subclass relationship. The **superclasses** function scans all the slots and collects all the references to superclasses.

```
(defun superclasses (fr)
  "returns the frame-ids of all the direct superclasses"
  (remove nil
    (mapcar #'(lambda (slot)
      (if (eql (slot-name slot) 'subclass-of)
        (contents slot)))
      (get-slots fr))))
```

The subclass relation is transitive, in that if a set, A, is a subclass of set B, and B in turn is a subclass of C, then A is also a subclass of C. We don't need to replicate all these transitive relations but we can compute them. By recursively collecting superclasses, we can create a list of *all* the superclasses of any particular class. Here is a function to do that.

```
(defun all-superclasses (fr)
  "returns the frame-ids of all the frames that are superclasses of
  frame fr all the way up the class hierarchy"
  (let ((direct-sups (superclasses fr)))
    (apply #'append
            direct-sups
            (mapcar #'(lambda (frm)
                        (all-superclasses
                         (find-frame-by-id frm)))
                    direct-sups))))))
```

\*\*\* this is a simplified version of finding the class precedence list - might as well do the whole nine yards - see Graham ch.11. This is also path search, so might expropriate the code in the search section, or refer back to this there. Also, maybe talk about computing transitive closures in general here.

Now we can implement some additional relations between classes and instances. Slots that specify properties of the object described by a frame are called “own” slots. So, in the **heart** frame and the instance, “Joe’s heart” above, all the slots are *own* slots. To specify in a class the names of the slots that should appear in its instances, we need a different kind of slot, a “template” or “member” slot. When a template slot appears in a frame (call it **frame1**), and we create another frame, **frame2**, that is an instance of the class defined by **frame1**, the template slots from **frame1** should appear as own slots in **frame2**. How will the frame system be able to identify whether a slot is a template slot or an own slot? Rather than having each of the template slots appear in the frame separately, we can accomplish this by including another special kind of slot, called **template-slots**. This slot will contain a list of descriptions of the template slots for the class described by the frame. For now, the description will just be a symbol naming the slot, but later we will enhance it to allow slots themselves to have properties.

The new version of our **heart** frame then could be:

```
(frame1342
  (name "Heart")
  (subclass-of frame3356)
  (template-slots (belongs-to beat-rate))
  (part-of cardiovascular-system)
  (contained-in middle-mediastinal-space)
  (has-parts (right-atrium left-atrium
               right-ventricle left-ventricle ...))
  ...)
```

We don’t include **instance-of** in the template slots list because we can include that slot automatically in the process of using the **heart** frame information to make an instance of a heart. This procedural encoding of the semantics of the class-instance relationship will be in the constructor function to make new frames,

although the relation itself will be encoded explicitly in the frame instance. Other such relationship semantics can also be procedurally encoded. This is the beginning of the design of the frame system overall, i.e., what rules and constraints will all the frames have, and how does one create, access and relate the frames and their contents.

\*\*\* need to consider what is true of all frames, i.e. our MOP - what about the **frame** class? It could have, e.g., template slots that insure all frames have name slots, template-slots slots, etc. discuss metaclasses here...

### Constructors, Inheritance and Subsumption

Now, we can provide constructor functions for class frames and instance frames. A constructor takes a number of inputs and constructs a frame from them. The inputs will depend on whether we are constructing a class frame or an instance frame. We provide a function for each case.

To make a class frame, we need the name to assign to the new class, a list of superclasses of which the new class is a subclass (this could be empty), and a list of slot names that will be the template slots for this class. It won't be necessary at this point to look up the template slots for the superclasses. They will be used later by the constructor function for instances.

This version commits to the low level detail specified above, so it might be nicer to make another abstraction layer, i.e., to have slot constructors and a low level frame constructor.

It is not required that every class define template slots, so the slots parameter could be nil.

```
(defun make-class-frame (name superclasses slots)
  "returns a frame with specified name.  superclasses is a list of
  names of superclasses, or nil, and slots is a list of names of
  template slots, or nil.  No other own slots here because
  metaclasses are not yet supported"
  (let ((fr (cons (gentemp "FRAME")
                  (cons (list 'name name)
                        (cons (list 'template-slots slots)
                              (mapcar #'(lambda (x)
                                          (list 'subclass-of x))
                                      superclasses))
                          ))))
    (push fr *frames*)
    fr))
```

For an instance frame we need as inputs the class of which this frame will represent an instance, the name to give to the instance, and the initial values of the slots as a sequence of keyword-value pairs. The initial values are optional. As a convenience these pairs should be converted to a list of pairs, rather than a flat sequence. The **make-pairs** function does that.

```
(defun make-pairs (x)
  (if (oddp (length x)) (error "list length not even")
      (if x (cons (list (first x) (second x))
                  (make-pairs (rest (rest x)))))))
```

An instance should have own slots that correspond to all the template slots from all its superclasses. One way to create a list of such slots is to use the `all-superclasses` function to get a list of frame identifiers, and then go to each and copy the template slots.

```
(defun all-template-slots (fr)
  (let ((supers (all-superclasses fr))
        (direct-slots (slot-data fr 'template-slots)))
    (apply #'append
            direct-slots
            (mapcar #'(lambda (frm)
                        (slot-data (find-frame-by-id frm)
                                   'template-slots))
                    supers))))
```

This use of information from superclasses as well as the direct parent class is called *inheritance*. It is so named, because each specialized class inherits properties from its superclasses. This is only one aspect of inheritance. Other aspects of superclasses may also be passed through to their subclasses.

Now we are ready to write the function that makes instances. It gets all the template slots and creates own slots from them, then it puts any initial values in that were passed as inputs to `make-instance-frame`.

```
(defun make-instance-frame (class-name inst-name &rest slot-inits)
  "returns a frame that is an instance of the class whose name is
  class, and initializes slots either to nil or to values specified
  by slot-inits, which is a list of keyword-value pairs."
  (let* ((own-slots (all-template-slots (find-frame class-name)))
        (init-pairs (make-pairs slot-inits))
        (fr (cons (gentemp "FRAME")
                   (cons (list 'name inst-name)
                         (cons (list 'instance-of class-name)
                               ;; every slot's initial value is nil
                               (mapcar #'(lambda (x)
                                           (list x nil))
                                       own-slots)))))))
    ;; and initialize them with any matching slot-inits
    (dolist (pair init-pairs)
      (setf (slot-data fr (first pair)) (second pair)))
    (push fr *frames*)
    fr))
```

The `slot-inits` input is an alternating sequence consisting of slot names and the values to put initially in those slots. Since each value is “tagged” by a slot name, these pairs (slot name and value) can be in any order. They are also optional, so you can provide initial values for some slots and not others, or no initial values.

\*\*\* need examples of use of these fns e.g. class person and individ. Ira

One class of questions to pose in a frame system is whether a class described by one frame is subsumed by (or a subclass of) a class described by another frame. This is computed essentially by following the values in the `subclass-of` slots in successive frames. If there is a path through the `subclass-of` relations, the first class is indeed a subclass of the second. A complication of this is that there could be multiple `subclass-of` relationships so we need to trace them all. So, we build on the frame code, generalizing it to handle multiple `subclass-of` slots, and write a general function to compute this relationship.

\*\*\* the following can also be answered by just searching the result of a call to `all-superclasses`, but that is not as efficient

To answer the question of whether a frame is ultimately a subclass of another frame by a series of class-subclass relationships, we use the `superclasses` function in a recursive search following all the superclass relations at each level. The predicate function, `subsumed-by`, answers this question.

```
(defun subsumed-by (fr1 fr2)
  (let ((id-list (superclasses fr1)))
    (cond ((null id-list) nil)
          ((find (frame-id fr2) id-list) t)
          (some #'(lambda (x)
                     (subsumed-by (find-frame-by-id x) fr2))
                id-list))))
```

This code can then be used with frame representations of biomedical knowledge such as the Gene Ontology (Chapter 7) to answer questions about relationships among Gene Ontology types or subclasses.

For questions having to do with specific types of instances, i.e., instances of a particular class, we can collect all the instances given the class name. Here is a function to do that.

```
(defun get-all-instances (frame-type)
  "returns a list of all the frames that are instances of a class
  frame whose name is frame-type"
  (remove nil (mapcar #'(lambda (x)
                          (if (eql (frame-type x) frame-type)
                              x))
                      *frames*))))
```

\*\*\* Ontologies vs. terminology systems, per AMIA KR-SIG discussions and symposia. UMLS and the Barry Smith papers.

### Attached Functions

While we will want to be able to answer questions concerning inheritance, and other relations implemented in slots, it will also be useful to provide a way to specify computations that should be performed when slots are accessed. When a slot value is needed, it may be present (because it was provided when the frame was created), or it may be computed at the time it is accessed. It is something like storing a formula in a spreadsheet, but is a more general facility, in that the function can take any action, not just compute a value.

\*\*\* maybe include here some discussion of “active values” in LOOPS and other historical antecedents

An example of an alternate use of an “if-needed” function would be to record in a log file when the access occurred and by whom. A medical context for such an access logging function would be an electronic medical record system, where it is required by law (HIPAA 2003) to keep records of all access to patient data, specifically that which is categorized as “protected health information” (PHI).

Similarly, one might want to add to a frame an action that takes place when a slot value is added or modified. An example would be: when a medication order is put into the electronic medical record, a safety check could execute, which would generate an alert if the new medication could cause a hazardous interaction with other medications or be contraindicated by a patient’s condition.

\*\*\* maybe a reference to discussion of the curly braces problem and guidelines

One possible way to provide this capability is to adopt a convention that if the slot value is a function it should be called instead of returned as the value. This does not allow for a function to be stored as an actual slot value to be returned rather than called. Also, a slot might need both an if-needed and an if-added procedure, but the convention does not allow storing both, or distinguishing one from the other.

Here is an alternate: in the slot, if an attached function is present, it appears after a keyword, `:if-needed` or `:if-added`, which follows the slot value. This allows for any combination of a value, an if-needed function and an if-added function. This arrangement has one limitation: a value must be present as a place holder (`nil` is OK, or one could reserve a special indicator value that means “no value stored”).

Thus, we extend the slot structure by including `:if-added` and `:if-needed` keywords that tag functions to be run. They will appear after the slot name and slot value. If there is no slot value, a `nil` must be there as a place holder.

```
(frame2046
  (instance-of person)
```

need cite here

```

(name "Ira")
(age 64)
(occupation professor)
(ssn nil
 :if-needed (lambda (fr slot) "SSN not available"))
(birth-date (9 9 1944)
 :if-added (lambda (fr slot val)
              (schedule-party (slot-data fr 'name) val)))
))

```

In this example, a lambda expression serves as the if-needed function, but a symbol naming a function would do also. In the case above, where the frame is created from source text as a quoted list, one must not use the `#'` syntax to refer to the function described by the lambda expression or the symbol, since the *preceding* quote prevents it from being evaluated; it will just be left in place. When the function call actually takes place, the function name with the `#'` prefix will be invalid. On the other hand, if in running code, an actual function object is put there, it will execute correctly. Chapter 1 provides some further explanation of the use of `quote` and related ideas.

As with the content of a slot, it is a good idea to provide an abstraction layer for the storage and retrieval of such attached functions.

(to be done)

Getting the functions to execute at the proper time can be done in a completely general way by just modifying the accessor functions `slot-data` and `(setf slot-data)` to execute any function that is present and labeled by the appropriate keyword. In this arrangement the slot value itself can be a function, but it will be returned, not executed. This also allows for a slot to have either or both types of attached functions. We use the above abstraction layer, again so that we can replace the slot structure with something better (perhaps more efficient) without having to change the following functions.

NOT DONE yet, fix this up

```

(defun slot-data (fr slot)
  "returns the contents of slot slot-name in frame fr, after
  executing any :if-needed function that might be present"
  (let* ((the-slot (get-slot fr slot))
         (if-needed-fn (second (member :if-needed the-slot))))
    (if if-needed-fn (funcall if-needed-fn fr slot)
        (second (get-slot fr slot))))

(defun (setf slot-data) (newval fr slot)
  "updates the contents of slot slot-name in frame fr, after
  executing any :if-added function that might be present"
  (let* ((the-slot (get-slot fr slot))
         (if-added-fn (second (member :if-added the-slot))))
    (if if-added-fn (funcall if-added-fn fr slot newval)
        newval)))

```

```
(setf (second (get-slot fr slot))
      newval)))
```

In the `slot-data` function, when looking up the slot value after the `:if-needed` function is run, we again call `get-slot` rather than using the value of `the-slot` because the `:if-needed` function can change the slot value. We want the updated value, not the original one. Similarly in `(setf slot-data)` we need the place name for `setf`, not the local variable in the `let` expression.

Of course we need to provide a way to put these functions in slots from a program when needed. The functions for putting functions in slots just look for keywords and update their associated values. If an attached function is already present, the new one replaces it. If there is none yet, the keyword and function are added to the end of the list that makes up the slot. Since the body is the same for all keywords, we can provide one function, parametrized by the keyword.

```
(defun set-attached-fn (fr slot fn key)
  "puts function fn in the slot named slot in frame fr, using key as
  a tag, e.g., :if-needed or :if-added or possibly other types of
  attached functions"
  (let* ((the-slot (get-slot fr slot))
         (length (length the-slot))
         (location (position key the-slot)))
    ;; this is where we need the abstraction layer!
    (if location (setf (elt the-slot (+ location 1)) fn)
        (setf (rest (last the-slot)) (list key fn)))))
```

It is also important to provide a way to remove attached functions from a slot as the knowledge base grows and changes. One way to do this is to literally remove the keyword and the function it labels. This is tricky, as we would have to do some complex list surgery. The implementation above supports the much simpler process of simply putting a `nil` value in place of the function. If `nil` is retrieved, in the `slot-data` and `(setf slot-data)` functions above, the call to `funcall` will not be executed, just as if there were no entry at all.

```
(defun remove-attached-fn (fr slot key)
  "replaces any attached function for key to nil"
  (set-attached-fn fr slot nil key))
```

### Other Kinds of Relations

Up to now we have only examined the relation `subclass-of`, the relation between a class and its superclasses. The `part-of` relation is also important, since it represents, for example in the Gene Ontology, a compositional relation, i.e., a function may consist of a series of sub-functions, or a cellular component may itself have components. In our representation of GO, the `subclass-of` and `part-of` relations are structured identically, so a simple modification of the above code can provide functions to traverse the `part-of` hierarchy. The analogue of the `superclasses`



function is the **part-of** function. Here too, it is possible to have more than one **part-of** slot.

```
(defun part-of (fr)
  (remove nil
    (mapcar #'(lambda (slot)
      (if (eql (slot-name slot) 'part-of)
        (contents slot)))
      (get-slots fr))))
```

The analogue of **subsumed-by** then is the **is-part-of** function. It has almost the same code, except for using **part-of** instead of **superclasses**.

```
(defun is-part-of (fr1 fr2)
  (let ((id-list (part-of fr1)))
    (cond ((null id-list) nil)
          ((find (frame-id fr2) id-list) t)
          (some #'(lambda (x)
            (is-part-of (find-frame-by-id x) fr2))
            id-list))))
```

Now we can see how to generalize this for a possibly richer ontology with many hierarchical relations like **subclass-of** and **part-of**. We just have to pass the function that finds the links as an input parameter. The essential requirement is that the link function represents a relationship of which we can meaningfully compute the transitive closure.

```
(defun connected-upward (fr1 fr2 link-fn)
  (let ((id-list (funcall link-fn fr1)))
    (cond ((null id-list) nil)
          ((find (frame-id fr2) id-list) t)
          ((some #'(lambda (x)
            (connected-upward
              (find-frame-by-id x) fr2 link-fn))
            id-list))))
```

To traverse the **subclass-of** hierarchy, use **superclasses**, as in

```
> (connected-upward fr1 fr2 #'superclasses)
```

To find out if something is a part of something else, use **part-of** instead of **superclasses**. It is essentially a search for a path from one frame to another in a directed frame graph defined by the **link-fn** relationship.

\*\*\* The following should be explained in more generality. This is a way to answer the query, "What kinds of animals are there?" Also add a forward reference

note on pathways, e.g, “Which genes are in a particular pathway?” or “In which pathways is a particular gene?”

In classifying or organizing experimental data it is often useful to identify and collect together data for a collection of related genes or gene products. In GO terms, this means pick a term in the GO hierarchy and find all its lineal descendants. A straightforward but horribly inefficient way to do this is to go linearly through the entire Gene Ontology and test each entry with the chosen entry, using the `subsumed-by` function defined above.

```
(defun get-all-children (fr frame-kb)

  "searches the entire frame knowledge base to collect all the
  subclass-of descendants of frame fr."

  (let ((children nil))
    (dolist (entry frame-kb children)
      (if (subsumed-by entry fr)
          (push entry children))))))
```

Similarly, to find all the parts and subparts corresponding to a GO term, use the `is-part-of` function.

```
(defun get-all-parts (fr frame-kb)

  "searches for items in the part-of subtree"

  (let ((parts nil))
    (dolist (entry frame-kb parts)
      (if (is-part-of entry fr)
          (push entry parts))))))
```

These two can also be combined into a single search function parametrized by the link function to use, either `subsumed-by` or `is-part-of`. might call this `get-subtree`

This is horribly inefficient because the organization of the Gene Ontology (and our frame system in general) is unidirectional, i.e., an entry has parents, but not children. In object oriented programming systems this is the way class hierarchies are organized. A class always includes a list of its superclasses, but never its subclasses. This works just fine for programming with classes, because in terms of software evolution, one generally adds specializations, but not generalizations. Adding a more specialized class does not change the behavior of any existing code. On the other hand, modifying a superclass may change the behavior of all the subclasses.

For dealing with class hierarchies as representations of biological entities, it is important to be able to do the traversal in terms of descendants in a more efficient way. One way to make the search much faster is to include links in both directions in each entry. Another is to go once through the entire table and generate an inverted hierarchy, i.e., compute the links in the opposite direction once and store

them separately. Strategies such as these are provided in object-oriented databases, in order to make such searches and retrieval operations reasonably fast.

Possibly expand this discussion and put some of it in chapter 2

### 3.3.2 Frame System Implementations and Standards

Ontolingua [32], Ontosaurus (interface to LOOM), Protégé and its various plugins.

Ocelot (SRI) and its predecessor, GFP, used by Ida Sim in addition to BioCyc. Really an OODB, from PK's point of view.

OKBC [19]

### 3.3.3 Frames and Object-Oriented Programming Languages

What parts of the frame system are provided by CLOS? What else does CLOS provide? Metaclasses and the MOP. Where in biomedical knowledge modeling do we need real metaclasses, as opposed to just MOP facilities? What about other OOPL?

### 3.3.4 Frames and the Semantic Web

\*\*\* DAML+OIL (Wroe Table 1, GO XML  $\leftrightarrow$  DAML+OIL, OWL (Semantic Web) (illustrate with GO XML?).

## 3.4 Description Logics

*If it looks like a duck, walks like a duck, and quacks like a duck, it's probably a duck.*

– attributed to Senator Joseph McCarthy  
1952

\*\*\* develop description logics, with examples, e.g. FaCT. Also RACER (unfortunately now a commercial product).

```
duck  ≐  [ALL  [FILLS :appearance Duck-like]
           [FILLS :walk Waddle]
           [FILLS :sound Quack]]
```

i.e., if it looks like a duck, walks like a duck, and quacks like a duck, it *is* a duck.

In frame systems and in object oriented programming, it is necessary to explicitly specify the relationships that make up various hierarchies, such as the **subclass-of** hierarchy. However, an alternate formulation is possible, in which we provide *descriptions* of the various entities in the system, in a way that allows for *computation*

of the subsumption relationships from the descriptions. Such systems are called *description logics*.

\*\*\* Explain difference between definition and subsumption!

\*\*\* Biomedical examples - Beck and Schulz, AMIA 2003, Heart and cancer examples in Hartel et. al.

\*\*\* develop example implementation of some subset of CLASSIC or LOOM or some other, maybe FaCT.

### 3.5 Search

So much of the computation we have described so far depends on ideas that involve searching a graph or network of connected nodes or states. This section describes how to do search on graphs in general, in a framework that allows specialized search methods to use the same general code, but just vary the order in which the graph is explored.

\*\*\* develop narrative for all this

Here is Peter's search code with some extras left out.  
The inputs:

**initial-state** is the start state of the search

**goal?** is the predicate that determines whether we have reached a goal

**enough?** is a predicate on the list of states that have reached the goal that determines whether search should stop. NIL means don't stop but generate all answers. T means stop on first answer.

**successors** computes from the current state all successor states it has access to wins for branch and bound

**merge** combines new states with the queue, and winning state with wins.

The main function, including some parameters: **search-trace** turns on some tracing if true, and **left-overs** allows examination of the queue after the search ends.

```
(defparameter search-trace nil)
(defparameter left-overs '())

(defun ssearch (initial-state goal? enough? successors merge)
  (labels
```

```

((search-inner (queue wins)
  (if search-trace
    (progn (prin1 '(search-queue ,queue wins ,wins))
      (terpri)))
  (if (null queue)
    wins
    (let* ((current (car queue)))
      (cond ((funcall goal? current)
        (setq wins (funcall merge (list current) wins))
        (setq left-overs (cdr queue))
        (if search-trace
          (progn (prin1 '(success with ,current))
            (terpri)))
        (if (or (eq enough? t)
          (and (null enough?) (null left-overs))
          (and enough? (funcall enough? wins)))
          (values wins left-overs)
          (search-inner (cdr queue) wins)))
      (t
        (search-inner (funcall merge
          (funcall successors
            current wins)
            (cdr queue))
          wins))))))
  (search-inner (list initial-state) '()))

```

This is then used with particular kinds of merge methods to get the different kinds of search strategies.

```

(defun depth-first-search (initial-state goal? enough? successors)
  (sssearch initial-state goal? enough? successors #'append))

(defun breadth-first-search (initial-state goal? enough? successors)
  (sssearch initial-state goal? enough? successors
    #'(lambda (new-states queue) (append queue new-states))))

```

Hill climbing is a simple kind of informed search.

```

(defun hill-climb-search (initial-state goal? enough?
  successors estim-fn)
  ;; Winston's hill-climbing backtracks.
  ;; Estim-fn is the estimated distance
  ;; to the goal; smaller is better.
  (sssearch initial-state goal? enough? successors
    #'(lambda (new-states queue)
      (append (sort new-states
        #'(lambda (s1 s2)

```

```

(< (funcall estim-fn s1)
   (funcall estim-fn s2))))
queue))))

```

For sophisticated informed search we need some additional help, a priority merge function that merges two priority queues. It merges the queues (lists in ascending order of val-fn) using a collate; `same?`, if given, chooses only the better one of two `same?` elements.

```

(defun priority-merge (a b val-fn same?)
  (cond ((null a) b)
        ((null b) a)
        ((and same? (funcall same? (car a) (car b)))
         (cons (if (< (funcall val-fn (car a))
                     (funcall val-fn (car b)))
                 (car a)
                 (car b))
                 (priority-merge (cdr a) (cdr b) val-fn same?)))
        ((< (funcall val-fn (car a)) (funcall val-fn (car b)))
         (cons (car a) (priority-merge (cdr a) b val-fn same?)))
        (t (cons (car b) (priority-merge a (cdr b) val-fn same?)))))

```

Then we can do best-first search. Best-first is much like hill-climb, except that we do not insist on pushing forward from the most recent node. `Estim-fn` is the estimated quality function; smaller is better. This may be either an estimate of the distance to the goal, as for hill-climb, or the work already done on a path; in the latter case, this works more like breadth-first.

```

(defun best-first-search (initial-state goal? enough?
                        successors estim-fn same?)
  (sssearch initial-state goal? enough? successors
    #'(lambda (new-states queue)
        (priority-merge (sort new-states
                              #'(lambda (s1 s2)
                                  (< (funcall estim-fn s1)
                                      (funcall estim-fn s2)))))
                        queue estim-fn same?))))

```

The above definition of best-first-search in fact implements branch-and-bound-search because the priority queue mechanism assures that any path whose cost estimate exceeds the cost to reach the goal along some other path will sort behind that other path in the queue. Nevertheless, the first goal path will not be found in the queue until all the lower-cost paths have been expanded. This is the essence of branch and bound.

Here is A\* search. The additional parameters are:

`g` is the cost-so-far function, which must be computable from the state



```

                                news
                                (cons (car queue)
                                      result))))))
(random-insert queue new-states '()))))

```

Finally, using the original basic search code, we can create a path search, which returns not just success or fail, but the paths. This is what we will need for the CTV project described in Chapter 14

```

(defun extend-path (path next-fn extender)
  (let ((nexts (funcall next-fn path)))
    (mapcar #'(lambda (new)
                (funcall extender new path))
            nexts)))

(defun path-search (start next-fn extender merge)
  (sssearch (list start) ;; makes start into a path of length 1
            #'(lambda (current) ;; goal - stop when no more nexts
                (null (funcall next-fn current)))
            nil ;; get all the paths
            #'(lambda (current wins)
                (declare (ignore wins)) ;; :( why is wins needed?
                (extend-path current next-fn extender))
            merge)) ;; keeps parametrization of control strategy

```

### 3.6 Classification Trees

How to represent a classification tree, how to create one from examples (a little about machine learning).

This is about classification trees as represented by e.g., clinical protocols, not probabilistic decision networks that use Bayes' Theorem, i.e., influence diagrams or belief networks, and also not decision trees that represent QALY or other outcome valuations.

### 3.7 Modeling Time

\*\*\* this and following stuff might go in a separate chapter

The knowledge representation ideas discussed so far provide a static view of knowledge in biomedicine, i.e., what entities exist and what are their relationships. Even with deduction systems based on first order logic or some other inference system framework, they do not represent change with time, though they *are* able to represent *spatial* relationships (as exemplified by work on the Foundational Model of Anatomy from the University of Washington [112] and recent work on a logic of spatial anatomy relations [26]). Much work has been done on temporal abstraction



[64, 22, 126], and on modeling dynamic systems using Petri nets [7, 101, 100]. There will be some basic discussion of this in the Knowledge Representation chapter, and in some chapters in Part II, which will include specific examples of modeling time and dynamic systems in biology and medicine. PSZ to email some more references

## 3.8 Simulation Models

Here is the place for Petri nets, Finite or infinite state machines, ATN's, etc. - describe ACLS, reference DICOM, describe implementation of a simple state machine.

\*\*\* write FSM code

\*\*\* also Markov models

### 3.8.1 Petri Nets

\*\*\* is this in the right place?

\*\*\* Mor Peleg work here?

### 3.8.2 Modeling of Organizations, Work Flow, and Scheduling

One possible topic is the use of process models to represent knowledge about work-flow and the impact of information systems. Examples in this area include work by Fridsma [35, 36] and Groothuis [42]. Another topic is the use of knowledge representation and reasoning ideas to solve medical scheduling problems. An example is a Prolog program that schedules radiology resident rotations [141].

## 3.9 Summary and Further Reading

The main points:

- Formal systems enable both reasoning and knowledge retrieval (answers to queries)
- Different formalisms have varying expressive power, and also different computational demands or complexity
- There are a range of choices regarding how much reasoning is put into procedures, and how much can be made implicit as part of a declarative formalism.
- No single formalism subsumes all biomedical knowledge representation and reasoning requirements.

- ...

\*\*\* to be concluded - the next chapter discusses probability and uncertainty in modeling biomedical knowledge, and how one reasons or answers questions with probabilistic models

References:  
(to be completed)

## Chapter 4

# Probabilistic Biomedical Knowledge

*Facts are stubborn, but statistics are more pliable*

– Mark Twain

...

When representing knowledge in terms of symbols and formulas of symbols, uncertainty and probability come up in several different ways. One possibility is that our theory, formed by a choice of symbols and a symbol system language like first order logic, a set of axioms, an inference method and an interpretation, might actually be incomplete, i.e., it is not able to infer all the important assertions we will observe in our biomedical or health environment to which the theory applies. Sometimes this can be resolved by adding more axioms. However, sometimes it cannot. In some cases, the behavior of the biological or other system we are modeling is inherently non-deterministic, i.e., the things we observe are not uniquely predictable from other more basic characteristics. Taking cancer as an example, the process of spread of tumor cells from the primary tumor site to form metastatic lesions is inherently statistical. So we need probability and statistics based modeling methods to represent such knowledge.

maybe also a genetics example here?

Another important case is the situation where there may be an underlying deterministic behavior, but it is completely beyond our reach for now, and we must model our uncertain knowledge about the behavior of the system, as distinct from its actual behavior. Again referring to cancer as an example, we know that Interleukin-2 can be an effective agent for treating kidney cancer, but only 20% of the patients so treated respond to the treatment (with tumor growth arrest, regression or complete ablation). We don't know which patients will respond, but one might imagine that at some future time we could discover which characteristics (perhaps genetic markers) would predict whether a patient with kidney cancer would respond to treatment

need cite, perhaps from JAT here

or not. As more information is gained about a particular case or system state, our knowledge becomes more refined, and our probabilistic modeling methods must be able to represent the refinement process.

Finally, to make reasonable practical decisions in clinical care and in public health, we will need to be able to incorporate into our models such factors as the relative importance of outcomes or effects, in cases where there are choices with multiple outcomes affected simultaneously by the choices. Again, referring to cancer, choices of treatments will often require deciding between a treatment that is more effective but more damaging to normal body functions, or a treatment that is less effective but less debilitating. Thus, we need a framework such as *multiattribute decision theory* to represent these tradeoffs and compute the consequences of one choice as compared to another.

In this chapter we introduce basic ideas of probability and statistics and then show how they can be applied to build probabilistic reasoning systems. This is followed by an introduction to multiattribute decision modeling. We conclude with some pointers to how categorical and probabilistic reasoning can be combined.

## 4.1 Basic Probability and Statistics

*A statistician is a person who, with his head in the oven and his feet in the freezer, will say that on the average he feels comfortable.*

– anonymous  
well known joke

This section introduces the basic rules and formulas of probability and statistics. More thorough developments can be found in standard probability and statistics textbooks [14, 137].

### 4.1.1 The Laws of Probability

The laws of probability concern *events*. An event is something with a definite outcome that can be observed. Put another way, we are describing the state of a system by specifying the values of a set of variables. The possible values (or combinations of values) of the variables represent all the possible states. Often rather than predicting the detailed outcomes, we can only predict the probability of any particular outcome or observed state. The classic example is a coin which we toss to see if it will land heads up or tails up.<sup>1</sup> The probability of an event tells us what is the likelihood that event will occur, out of all possible events. It is a fraction between 0.0 and 1.0, so that a probability of 0.0 means that the event will

---

<sup>1</sup>Actually, many coins do have a head, usually of a significant political figure, a president, monarch or some such. However, I personally have never seen a coin with a tail depicted on it. The US “buffalo nickel” comes close, but the whole bison (not a buffalo) is depicted, not just the tail. By “tail” we generally mean the opposite side of the coin from the head.

*never* occur, while a probability of 1.0 means that the event will *certainly* occur. Thus we have (equation 4.1) the first *axiom* of probability.

$$0.0 \leq p(x) \leq 1.0 \quad (4.1)$$

The head and tail outcomes are exhaustive and mutually exclusive, i.e., when we toss the coin it *always* comes up one way or the other, not both (we are not considering the possibility that it lands on edge - we will return to this later). With mutually exclusive events, there is no overlap. Since these events cover all possible outcomes the probabilities must add up to 1.0, i.e., one of them certainly happens. The coin's state after landing is described by a single variable, "heads", whose value is either "true" or "false" (i.e., "tails"). A variable with only these two possible values is called a *Boolean* variable.

For a Boolean variable, one can express the outcomes as an occurrence or non-occurrence, e.g., heads or no heads. For such a variable, the requirement that the sum of all probabilities must be 1.0 is expressed in equation 4.2. Here, the variable  $x$  has the value "true" when the occurrence named  $x$  happens, and "false" when the occurrence does not happen.

$$p(x) + p(\neg x) = 1.0 \quad (4.2)$$

When there are  $n$  possible (independent and mutually exclusive) outcomes,  $x_i$ , this generalizes to equation 4.3. An example would be the selection of a colored ball from a bowl containing many balls, each of which is colored with one of  $n$  different colors. This is an example of a discrete variable (of which a Boolean variable is the special case where  $n = 2$ ). A biological example is the particular base that appears at a specified location in the genome of an organism. In that case,  $n = 4$ , since there are four possible bases that could appear at any point in a DNA sequence.

$$\sum_{i=1}^n p(x_i) = 1.0 \quad (4.3)$$

Sometimes we can have multiple events occur simultaneously, so that we have to consider the probability of one or the other occurring (inclusive "or"), and the probability that both occur. In this case, we need a variable for each event. In general, when talking about events, we are referring to the assignment of values to variables. We denote by  $p(x \vee y)$  the probability that either  $x$  or  $y$  occurs (including the possibility that *both*  $x$  and  $y$  occur). Similarly we denote by  $p(x \wedge y)$  the probability that *both*  $x$  and  $y$  occur. These probabilities are related by equation 4.4.

$$p(x \vee y) = p(x) + p(y) - p(x \wedge y) \quad (4.4)$$

In medical diagnosis it is common for a patient to have multiple symptoms, and possibly multiple diagnoses. These can occur separately or together. This problem would require the use of multiple discrete variables.

When multiple events occur, we can also express relations between them by measuring or computing the *conditional* probability that one event will occur given that another has already been determined to occur. The conditional probability

that event  $x$  will occur, given that event  $y$  has already occurred, is denoted by  $p(x|y)$ .

The conditional probabilities are related to the individual probabilities and the probability of both events by formulas 4.5 and 4.6. These formulas provide a *definition* of conditional probability.

$$p(x|y) = \frac{p(x \wedge y)}{p(y)} \quad (4.5)$$

$$p(y|x) = \frac{p(x \wedge y)}{p(x)} \quad (4.6)$$

biological and medical examples here

Conditional probabilities are often used to express causal relations between events, i.e., when  $y$  occurs we expect most of the time that  $x$  will occur also, through some causal mechanism. However, we can define conditional probabilities whether or not we believe there is a causal relation between the two variables. In fact, the formulas (4.5 and 4.6) are symmetric with respect to  $x$  and  $y$  so there is no way to infer causality from the formulas or from raw data. Instead we construct a causal theory, test it, and if its predictions match the data, where no other theory does as well, we may accept the causal relation provisionally as true.<sup>2</sup>

need some realistic numbers here, and some other examples

When we hypothesize that  $y$  *causes*  $x$ , the conditional probability  $p(x|y)$  is usually close to 1.0. The fact that it is not exactly 1.0 accounts for the statistical nature of this causal relationship. For example, the presence of certain kinds of pollen usually causes allergy symptoms, but it is possible for the pollen to be present without the symptoms, even for the same individual.

Since any particular observation has a particular value, how can we determine experimentally what the probabilities are for the various values? We can't measure probabilities directly. Two approaches are possible, one involving ensembles and the other involving time.

In the ensemble approach, we have a large number of identical systems, e.g., a large number of identical coins. We toss them all, and count the number that came up heads and the number that came up tails. The fraction of the total that came up heads should approximate the probability of a "heads" result, and by definition (and guaranteed by the experimental procedure just described) the probability of a "tails" result should be 1.0 minus the heads probability. One might wonder how such experiments can really provide the information. The question is, "what sampling method provides an *estimate* of the probabilities in the probability distribution?"

\*\*\* random variables and statistical distributions, the joint probability distribution - elaborate on expectation, mean, variance, etc.

In order for this procedure to yield an accurate approximation to the true probability, the number of coins (or systems or replicated event observations) must be

---

<sup>2</sup>Often, there is only anecdotal data to support the theory, but for many people this is tantalizingly appealing, leading to the often used expression, "Coincidence? I think not."

very large.

\*\*\* reference Law of Large Numbers and statistics? Wasserman Chapter 5

\*\*\* next describe time sampling, and ergodicity?

### The Joint Probability Distribution

In general, a domain has multiple random variables, not just one.

- An *atomic event* is an assignment of particular values to all the variables,  $X_1, \dots, X_n$ .
- The table of values is the joint probability distribution  $\mathbf{P}(X_1, \dots, X_n)$ .
- Often, each variable  $X_i$  is a Boolean, but this is not required.

An example of a simple joint probability distribution with just two variables, High PSA and Prostate Cancer, is shown in Table 4.1. A very common test that is used for screening for prostate cancer is the PSA test. PSA stands for “Prostate Specific Antigen”, a protein that may be present in blood in very small concentrations, but in the presence of prostate cancer the concentration is higher. A PSA below 4 nanograms per milliliter of blood serum is considered normal. A PSA of between 4 and 10 that is rising over time is a good indicator of the need for more extensive investigation. Above 10, an even higher level of concern is appropriate. The joint probability distribution table shows the probability for each combination of values of each of the variables. In general, for  $n$  Boolean variables, the joint probability distribution will have  $2^n$  entries. For variables that can have more than two values, the table gets correspondingly larger.

Table 4.1: A Simple Joint Probability Distribution for Prostate Cancer and High PSA

	<i>High PSA</i>	$\neg$ <i>High PSA</i>
<i>Prostate Cancer</i>	0.25	0.07
$\neg$ <i>Prostate Cancer</i>	0.03	0.65

#### 4.1.2 Bayes’ Theorem and Applications

When we use conditional probabilities to express causal relations, we are relating causes and effects. Seeing the effect, we would like to infer the cause. This is parallel to abductive, or explanatory, reasoning in First Order Logic, as discussed in Chapter 3. This is not a sound inference in First Order Logic. Nevertheless, it can be useful in order to consider possible explanations. In this section we show

how a probability model can shed some light on how believable such explanations or hypotheses about causes can be. Most especially, when multiple effects or observations are made, the probability of one hypothesis or cause may be very high and others very low. In any case, being able to assign probabilities to the possible causes or explanations is helpful in the diagnostic process.

The definition of conditional probability is symmetric, i.e., it does not intrinsically represent a one way relation. However, the relations in equations 4.5 and 4.6 allow us to relate the conditional probabilities in a way that allows a kind of causal reasoning. Solving equation 4.6 for  $p(x \wedge y)$  and substituting into equation 4.5 gives the basic form of Bayes' Rule (equation 4.7).

$$p(x|y) = \frac{p(y|x)p(x)}{p(y)} \quad (4.7)$$

Let  $c$  be a hypothetical cause (the variable  $x$  in equation 4.5), and  $e$  be some observation or evidence (the variable  $y$  in equation 4.5) that is usually caused by  $c$ . For example,  $c$  could be atherosclerosis, and  $e$  could be high blood pressure. This causal relationship can be represented in a simple diagram, as shown in Figure 4.1.

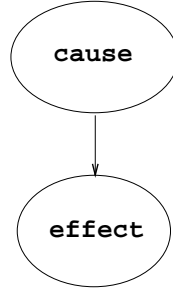


Figure 4.1: A Simple Causal Relationship Diagram

We observe that a patient has high blood pressure. What is the probability that the patient has atherosclerosis? This is the conditional probability  $p(c|e)$ . It is difficult to determine this from patient data since not all patients who have high blood pressure will have come in for diagnosis, and there are other causes of high blood pressure. It is relatively easy to determine the conditional probability  $p(e|c)$  – we take a large sample of atherosclerotic patients and count how many present with high blood pressure. Also the “prior”, or unconditional, probabilities of occurrence of each of these is often known.

Actually, the denominator  $p(e)$  may not be readily measured, as noted earlier, because not all people with high blood pressure report this fact. However, we can compute it from a sample of patients who do *not* have atherosclerosis, by observing the rate of occurrence of high blood pressure in that group. This quantity is  $p(e|\neg c)$ . Then we compute  $p(e)$  from equation 4.8.

$$p(e) = p(e|c)p(c) + p(e|\neg c)p(\neg c) \quad (4.8)$$



which gives

$$p(c|e) = \frac{p(e|c)p(c)}{p(e|c)p(c) + p(e|\neg c)p(\neg c)} \quad (4.9)$$

Implementing mathematical formulas as functions in a computer programming language is pretty straightforward in any language. The Lisp language uses *prefix* notation, i.e., for any expression, the operator is first and the operands follow. This contrasts with most Algol-like languages (FORTRAN, C, Java, BASIC, perl, etc.), which use *infix* notation, more like mathematical expressions. In infix notation, operators appear between operands, and thus need to be repeated to have expressions in which there are multiple operands even though the operator is the same. For example, to add up a bunch of numbers, the mathematical expression might be  $a + b + c + \dots$  while in Lisp it is `(+ a b c ...)`. Although prefix notation seems strange at first, once you get used to it, the simpler syntax and semantics become a real advantage.

Thus, we can represent formula 4.7 as

```
(defun cond-c-e (cond-e-c prior-c prior-e)
  ((/ (* cond-e-c prior-c) prior-e)))
```

and formula 4.8, substituting  $1 - p(c)$  for  $p(\neg c)$ , is

```
(defun prior-e (cond-e-c prior-c cond-e-not-c)
  (+ (* cond-e-c prior-c)
     (* cond-e-not-c (- 1 prior-c))))
```

Then combining these, as in formula 4.9 we can redefine the formula for  $p(c|e)$  as

```
(defun cond-c-e (cond-e-c prior-c cond-e-not-c)
  (/ (* cond-e-c prior-c)
     (prior-e cond-e-c prior-c cond-e-not-c)))
```

or alternately without the extra function call,

```
(defun cond-c-e (cond-e-c prior-c cond-e-not-c)
  (/ (* cond-e-c prior-c)
     (+ (* cond-e-c prior-c) (* cond-e-not-c (- 1 prior-c)))))
```

\*\*\* here give some bio and clinical and ph examples, that the reader can plug into his/her lisp system.

### Multiple Hypotheses or Causes

In some cases there are many possible causal hypotheses, which we denote as  $c_i$ . If these causes represent mutually exclusive alternatives, we can generalize Bayes' Rule to handle this case. Then we have a distribution,  $p(c_i)$ , and the conditional probabilities similarly are represented by a distribution.

\*\*\* need a general graph here

In this case, the prior probability of the evidence can be computed as a sum over all the hypotheses (causes) of the product of the prior and conditional probabilities, just as for the Boolean case where we summed over the two cases, true and false.

$$p(e) = \sum_i p(e|c_i)p(c_i) \quad (4.10)$$

Having a way to compute  $p(e)$ , we can now compute the conditional probability of each hypothesis given a single piece of evidence. This, together with equation 4.10, is one form of a generalized Bayes' Theorem:

$$p(c_i|e) = \frac{p(e|c_i)p(c_i)}{p(e)} \quad (4.11)$$

A simple implementation of formula 4.10 would use the `do` iteration construct, and (to be general) would obtain the array dimension of the `c` array to use as the iteration upper bound.

```
(defun prior-le-nc (cond-e-c prior-c)
  (do ((i 0 (1+ i))
      (stop (1- (array-dimension prior-c 0)))
      (result 0))
    ((> i stop) result)
    (incf result
      (* (aref cond-e-c i) (aref prior-c i)))))
```

Expressions with sums and products occur often in probability and statistics. Rather than having to write out these `do` loops, it would be nice to have `sum` and `product` operators, that can take expressions and transform them into iterative code as above. The Common Lisp macro facility is designed to do exactly that. A brief introduction to macros is in Chapter 1. Following an example in Graham [41, pages 169–171], we can define these operators and then use them as if they were built in. They follow the form of the `for` macro in Graham, but each does the sum or product operation as part of the definition, so that makes their use simpler even than using the `for` macro.

```
(defmacro sum (var start stop &rest body)
  "Evaluates body for each value of var from start to stop,
   inclusive, summing up the results of body as a progn"
  (let ((gstop (gensym)))
```

```

      (gresult (gensym)))
    (do ((,var ,start (1+ ,var))
        (,gstop ,stop)
        (,gresult 0))
        ((> ,var ,gstop) ,gresult)
        (incf ,gresult
          (progn ,@body))))))

```

The `let` form creates some unique local variables using `gensym`, and then uses them in the iteration. The symbols `gstop` and `gresult` are not literally used. Their *values* (the symbols created by `gensym`) are the symbols (variables) actually used in the `do` expression that the macro creates when it is called. Those are initialised to the values passed in with the macro call. This is to avoid the possibility that any particular name we might choose there would be also in the `body` code that the user of this macro provides. That is called variable capture and it is a source of subtle errors. On the other hand, the value of the first parameter, `var`, is exactly the name of a variable used for the iteration, and it is intended that it appear in the body. The `prior-1e-nc` function then looks like this:

```

(defun prior-1e-nc (cond-e-c prior-c)
  (sum i 0 (1- (array-dimension prior-c 0))
    (* (aref cond-e-c i) (aref prior-c i))))

```

When this new definition is used, it will be translated by macro expansion into an expression very similar to the first implementation above. In using this macro the symbol `i` above is the value of `var` in the macro definition, the number `0` becomes the value of `start`, and the `array-dimension` function is used to get the `stop` value (one less than the size of the array). The result of macro expansion of the `sum` expression in the above definition of `prior-1e-nc` can be seen by using `macroexpand-1`. Its input is the expression as we coded it, and the transformed code is shown following, in all upper-case.

```

> (macroexpand-1
  '(sum i 0 (1- (array-dimension prior-c 0))
    (* (aref cond-e-c i) (aref prior-c i))))

(DO ((I 0 (1+ I))
    (#:G16 (1- (ARRAY-DIMENSION PRIOR-C 0)))
    (#:G17 0))
  ((> I #:G16) #:G17)
  (INCF #:G17
    (PROGN
      (* (AREF COND-E-C I) (AREF PRIOR-C I)))))

```

The symbol `G16` corresponds to `stop` in the original code, and `G17` corresponds to `result` in the original code. These symbols are unique symbols that were created automatically by the macro expansion process. The result of the macro expansion

is pretty much the same as the original `do` expression that we wrote. So, having this macro, we can just *use* it instead of writing out complicated `do` forms.

Now formula 4.11 can be coded very simply by utilizing the function just defined for  $p(e)$ . Of course one could also write the body out in the following, but the layered design approach is likely to pay off as the formulas get further generalized.

```
(defun cond-nc-1e (i cond-e-c prior-c)
  (/ (* (aref cond-e-c i) (aref prior-c i))
     (prior-1e-nc cond-e-c prior-c)))
```

An example of such a case would be the problem of identifying a bacterial sample consisting of a single species of bacteria, by making a single observation (e.g. Gram Stain). The problem with this example is that there are thousands of species, so in general a single piece of evidence will not be sufficient to discriminate, and we will examine the problem of generalizing Bayes' Rule to multiple pieces of evidence a little later.

\*\*\* need examples here also, to plug in

### Combining Multiple Pieces of Evidence

Suppose we have two or more *kinds* of evidence. How do we combine them to estimate the conditional probability of a cause? This is analogous to the multiple cause problem, but the mathematics is slightly different. We will first consider the case of just two pieces of evidence, and use the example described earlier of testing for prostate cancer as an illustration. In addition to the High PSA test result, we consider the percent free PSA.

Most of the PSA observed in serum is bound to a protein inhibitor, alpha-1-antichymotrypsin. The percentage of PSA that is unbound (noncomplexed), is referred to as the percentage of “free” PSA, a fraction of the total. In diagnostic testing, two numbers are reported, the total PSA concentration and the percentage of it that is “free”. The percentage of free PSA seems to be lower when prostate cancer is present. When the PSA is between 4 and 10, if the fraction of free PSA is less than 0.10, the probability of having prostate cancer is considerably higher than it is if the free PSA fraction is higher.

Here, rather than distinguishing all the levels and combinations, we just use simple categories. In the formulas, to be concise, we abbreviate “Prostate Cancer” as “PC”, “High PSA”, i.e., greater than 4, as just “PSA” and “abnormally low fraction of free PSA”, i.e., less than 0.10, as “FPSA”. Figure 4.2 shows diagrammatically the relationships. We have filled in the conditional probability tables for each node in the diagram. The Prostate Cancer node just has one entry, the prior, or overall, probability of occurrence of prostate cancer. The High PSA node conditional probability table is calculated from the joint probability distribution in Table 4.1. We have not provided the corresponding joint distribution for FPSA, but only the conditional probabilities of observing FPSA when prostate cancer is present and when prostate cancer is not present. You can of course compute the joint distribution

from just these numbers and the prior probability of prostate cancer, assuming that PSA and FPSA are conditionally independent.

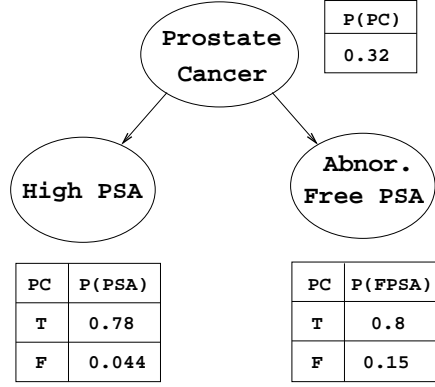


Figure 4.2: A Bayes Net for Prostate Cancer

Essentially the conditional probabilities for each of the tests give the true positive and false positive rates. From these and Bayes' Rule, we can compute what we are really interested in, that is, given a positive test, what is the probability of having prostate cancer.

$$p(PC|PSA) = \frac{(0.78)(0.32)}{(0.78)(0.32) + (0.044)(0.68)} = 0.89 \quad (4.12)$$

$$p(PC|FPSA) = \frac{(0.8)(0.32)}{(0.8)(0.32) + (0.15)(0.68)} = 0.715 \quad (4.13)$$

Now suppose we have high PSA *and* abnormal free PSA? How do we combine the information from both tests? We can use Bayes' Rule here too,

$$p(PC|PSA \wedge FPSA) = \frac{p(PSA \wedge FPSA|PC)p(PC)}{p(PSA \wedge FPSA)} \quad (4.14)$$

but we don't have the complex conditional probabilities required here. Instead, reformulate in two steps. First, the high PSA.

$$p(PC|PSA) = p(PC) \frac{p(PSA|PC)}{p(PSA)} \quad (4.15)$$

Add the knowledge that FPSA is also observed:

$$p(PC|PSA \wedge FPSA) = p(PC|PSA) \frac{p(FPSA|PSA \wedge PC)}{p(FPSA|PSA)} \quad (4.16)$$

$$= p(PC) \frac{p(PSA|PC)}{p(PSA)} \frac{p(FPSA|PSA \wedge PC)}{p(FPSA|PSA)} \quad (4.17)$$

Som simplifying assumptions can be made:

- hypotheses are mutually exclusive and collectively exhaustive, and
- evidence is conditionally independent, i.e.,

$$p(e_i|c) = p(e_i|e_j, c) \quad (4.18)$$

Then,

$$p(e_1, e_2, \dots, e_j|c) = p(e_1|c)p(e_2|c) \dots p(e_j|c) \quad (4.19)$$

Using Conditional Independence we can then simplify our earlier formula for combining evidence. Conditional independence means that

$$p(FPSA|PSA \wedge PC) = p(FPSA|PC) \quad (4.20)$$

and

$$p(PSA|PC \wedge FPSA) = p(PSA|PC) \quad (4.21)$$

This allows us to simplify the update formula:

$$p(PC|PSA \wedge FPSA) = p(PC) \frac{p(PSA|PC)}{p(PSA)} \frac{p(FPSA|PC)}{p(FPSA|PSA)} \quad (4.22)$$

More simplification is possible, by using the definition of conditional probability, formulas 4.5 and 4.6. What to do about  $p(FPSA|PSA)$ ? The product of this and the other denominator,  $p(PSA)$ , is equal to  $p(FPSA \wedge PSA)$ .

Write the equivalent formula for  $p(\neg PC|PSA \wedge FPSA)$ .

$$p(\neg PC|PSA \wedge FPSA) = p(\neg PC) \frac{p(PSA|\neg PC)}{p(PSA)} \frac{p(FPSA|\neg PC)}{p(FPSA|PSA)} \quad (4.23)$$

In this expression, the denominator is equal to the sum of the two numerators. Sum the two, giving on the left hand side exactly 1. Then from known quantities,  $p(FPSA \wedge PSA)$  can be obtained. The final update formula for the prostate cancer problem is:

$$\begin{aligned} p(PC|PSA \wedge FPSA) &= p(PC)p(PSA|PC)p(FPSA|PC)/ \\ &[p(PC)p(PSA|PC)p(FPSA|PC) \\ &+ p(\neg PC)p(PSA|\neg PC)p(FPSA|\neg PC)] \quad (4.24) \end{aligned}$$

In the previous section, we saw how to generalize to a multi-valued hypothesis, or many possible causes, rather than just a Boolean. Now we see how to generalize to many pieces of evidence, when the items of evidence are conditionally independent.

$$p(c|e_i) = \frac{p(c) \prod_i p(e_i|c)}{p(c) \prod_i p(e_i|c) + p(\neg c) \prod_i p(e_i|\neg c)} \quad (4.25)$$

In this formula, we have a *product* expression instead of a sum. We can support this by using a macro implementing a **product** operator, exactly analogous to the **sum** macro. Coding of this macro and implementation of formula 4.25 are left as an exercise for the reader.

Similarly you can extend this formula for many pieces of evidence and a single Boolean causal hypothesis,  $c$ , to incorporate multiple causes,  $c_j$ . This is also left as an exercise for the reader.

## 4.2 Probabilistic Modeling Methods

### 4.2.1 Bayes Nets and Influence Diagrams

\*\*\* Influence diagrams, Belief networks, etc., likelihood ratios as in MYCIN, DXplain, QMR

\*\*\* go into detail about how Bayes net programs work (the core implementation), with simple illustrations from cancer planning. In Part II, application chapters will go into more detail to illustrate issues that come up because of the nature of the medical knowledge.

\*\*\* Bayes Net code to be contributed from PSZ

\*\*\* Modeling time with dynamic (temporal) Bayes nets?

Prostate cancer, like many other kinds of cancers, can usually be treated in a variety of ways, including several strategies involving radiation therapy. It is possible to model many of the effects and ultimately relate them to long term survival, and as we will see later, quality of life. Figure 4.3 shows a decision model representing this idea.<sup>3</sup>

### 4.2.2 Probabilistic Decision Trees

Explain the difference between these and *classification* trees, as described in Section 3.6 of Chapter 3.

Something like the VW green light example, but a medical version. Xref to Shortliffe [120, Chapter 3]

1. Create a decision tree

- Chance nodes, probabilities

---

<sup>3</sup>This model was created for the author by Eric Horvitz, of Microsoft Research.

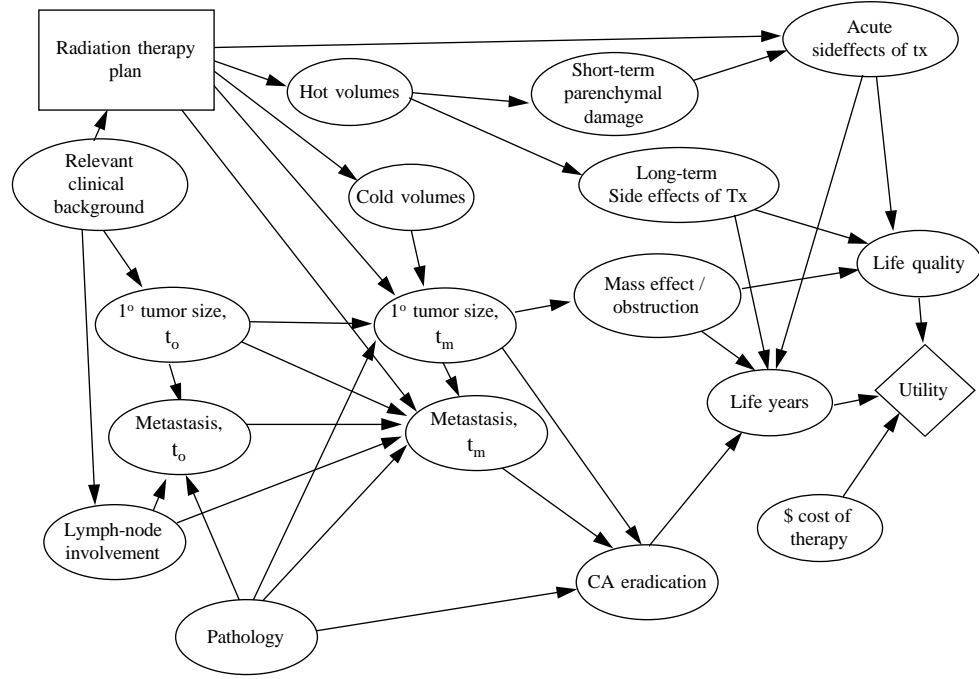


Figure 4.3: A decision model for radiotherapy

- Decision nodes, expected values
- Leaf nodes, outcome measures

2. Calculate expected value of each decision alternative
3. Use sensitivity analysis to test conclusions

At the chance nodes,  $EV = \sum_i p_i(EV_i)$

\*\*\* replace this with an original example, maybe IK fluid balance example, or something from cancer treatment

### 4.2.3 Multiattribute Decision Modeling

Develop machinery needed for the decision model for selecting an optimal prostate cancer treatment [88].



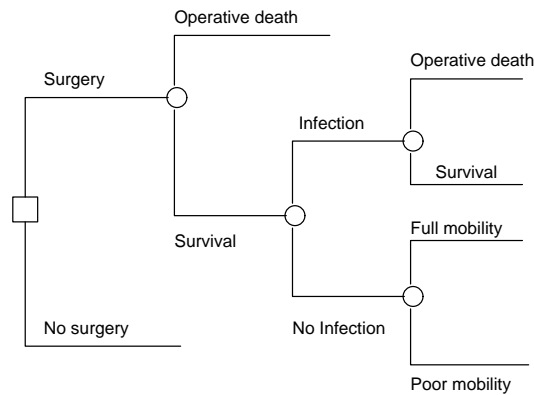


Figure 4.4: An example decision tree, from Shortliffe, et.al. “Medical Informatics”, 2006

#### 4.2.4 Statistics and Statistical Modeling

What is a statistical distribution? Relation between statistics and probability, meaning of an estimator, how to do it, maximum likelihood, fitting models, root-mean-square estimators, etc.

### 4.3 Stochastic Models

Explain Markov models and the machinery for Noah’s tumor spread model.

### 4.4 Hybrid Models

Combining probabilistic and symbolic models, e.g., MYCIN, rules with certainty factors, and also maybe Daphne Koller’s work, combining frames and Bayes nets.

Some example rules from MYCIN

- Rule047
  - If: 1. gram stain +
  - 2. morphology coccus
  - 3. growth in chains
  - Then: identity is streptococcus (.7)
- Rule204
  - If: 1. site of culture is throat
  - 2. identity is streptococcus
  - Then: organism subtype is not group-D (.9)

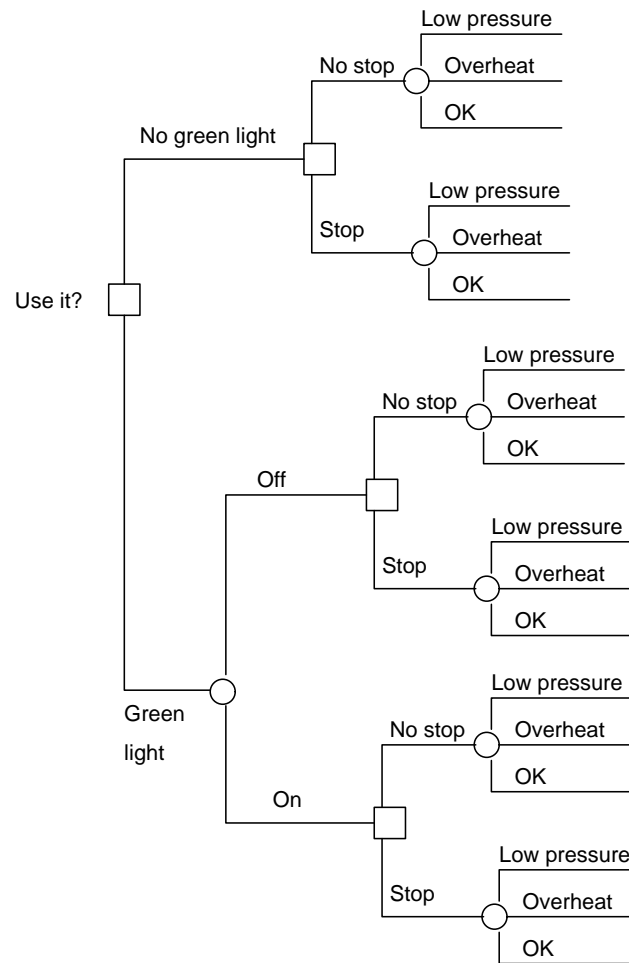


Figure 4.5: The VW bug green light

## 4.5 Information Theory

\*\*\* is this the right place for an intro? alternate is chapter 5

## Chapter 5

# Biomedical Information Access

*Biomedical information access* is concerned with organizing textual records and documents, and searching such repositories to identify relevant and important documents. The key concepts here are: how the structure and organization of information affect the efficiency of search, the construction and use of indexing systems, how a query is matched to items, how to rank order the results, and alternatives to rank ordering.

This chapter will show how such document repositories are constructed, and how search engines work. We will also examine how the problem of searching the World Wide Web (“the Web”) for relevant web pages can be addressed with these ideas. Some special considerations are needed for searching “knowledge bases” as described in Chapter 3, as well as searching image collections for content. Dealing with collections of heterogeneous information resources will be dealt with in Chapter 17 using the Biomediator project [25] as an example.

\*\*\* Outline of principles, search, query languages, etc., maybe something about NLP, pattern matching. Need a more in depth intro here and a chapter guide.

## 5.1 Natural Language Processing

### 5.1.1 Principles

Here talk about grammars, rule based translators, text processing (tokenizers, parsing etc.).

### 5.1.2 Applications

- extracting knowledge from text - PubMed, NCBI, etc.

- processing clinical data - dictations (H+P, radiology reports)
- other?

## 5.2 Information Retrieval Systems

What is an IR system? Distinguish between document repositories, relational databases, object-oriented databases, etc.

Define “search engine”, give overview of IR system architectures and ideas. Define boolean and vector space models, i.e., what constitutes a “match”.

This is (for starters) based on Elizabeth Liddy’s article, “How a Search Engine Works” at <http://www.infoday.com/searcher/may01/liddy.htm>

## 5.3 IR System Design

How is an IR repository created and structured? How do search engines work? Here give example code to do the various parts in the subsections following.

### 5.3.1 Indexing

What is an index? Use of NLP techniques. What about weighting of index entries?

Indexing, page ranking and other stuff from the Google article: Sergey Brin and Lawrence Page, “The Anatomy of a Large-Scale Hypertextual Web Search Engine”, <http://www-db.stanford.edu/backrub/google.html>

### 5.3.2 Processing Queries

Analysing queries and reformulating for optimal use with the index.

### 5.3.3 Searching and Matching

Search algorithms

### 5.3.4 Ranking the Results

## 5.4 Content-based Image Retrieval

Help from Linda Shapiro here.

## 5.5 Electronic Medical Records

Are EMRs searchable? What queries should be supported? What methods from this chapter are applicable?

## 5.6 Querying Knowledge Bases

OKBC, KIF, OWL, etc.

Again, the “curley braces” problem?

What about viewing knowledge bases like FMA and Galen as information storage, with retrieval mechanisms?

Possibly discuss computer programs as views on databases of code fragments (libraries)



## Chapter 6

# Information Use in Biomedical Contexts

This area, *information and technology use in biomedical contexts*, is also one in which computer systems play an important role, but it is not yet clear what kinds of programming examples are appropriate. Maybe instead this is the “editorial soapbox” chapter?

### 6.1 The Social and Political Context of Biomedical Computing

Task analysis, evaluation of systems and software, collaborative work environments.

In the University of Washington graduate biomedical and health informatics curriculum, this topic is treated from a social and political point of view, i.e., addressing the question of how the introduction of information technology impacts organizations and people. We also study how social and organizational frameworks can facilitate, impede or transform the use of computing and information resources. Considering the fact that major projects often fail for these non-technical reasons, it will be valuable to include a discussion of these issues with actual examples known to the author.

Move to a separate chapter in a possible Part III?





## Part II

# Biomedical Ideas and Computational Realizations



## Chapter 7

# Classifying and Organizing Genes and Proteins

This chapter demonstrates the representation and use of classification hierarchies such as the Gene Ontology [5]. The Gene Ontology in its present form needs much more work to be a precise and consistent representation of knowledge about gene products. However, it still appears to be useful, for example, to organize data from microarray experiments, to determine whether changes in expression levels or ratios are associated with classes of functions. Examples from Shrager's tutorial web site illustrate the point that one can discover patterns in such data that uninformed clustering or other statistical methods will not find. It also is an opportunity to demonstrate the difference between information representations that can be used as computable knowledge with which a program can make inferences, and representations (including XML) which can be formatted, searched or browsed but for which interpretation and use is left to the human once the visual display is generated. Some of the issues of consistency and representation will also be addressed.

The main point here is that the action is in the annotations, so make them into computable data and knowledge.

### 7.1 Some Essential Molecular Biology

Describe with references the following:

- The Fundamental Dogma of Molecular Biology: DNA to RNA to Protein

- Proteins regulate genes, are used as building blocks for structure, and also regulate metabolism (forward ref to EcoCyc)

- Other biological molecules?

- Something about conventional computational biology/bioinformatics, and annotations...

- More of Alberto's CL-DNA code and some examples here - e.g. frequency of use of codons for various amino acids, frequency of appearance of amino acids in

proteins, vs. number of codons for them.

## 7.2 The Gene Ontology

The Gene Ontology project, according to its creators, “seeks to provide a set of structured vocabularies...that can be used to describe gene products in any organism.” We will see from some examples that an appropriate representation can be a powerful tool for analysis of experiments, and for other applications. To do this, it needs to be put in a form with which inferences and classification reasoning can be done. The purposes proposed by the authors of the Gene Ontology [5] are pragmatic: to facilitate communication between people and organizations, and to improve interoperability between systems. To these I add a third, to provide a formal theory of gene products at the cellular level.

### 7.2.1 Representations of GO

XML, symbolic trees, description logics,

\*\*\* more about GO and its uses here - add description of OBO and XML formats, the semantics of GO, examples of use, e.g., classifying groups of gene products, others

The Gene Ontology, abbreviated as GO, is a collection of information on gene products (proteins). It is organized into three hierarchical structures, which may (loosely) be called an ontology. The three subdivisions concern molecular function, cellular location, and biological process. Known proteins can be assigned a GO ID, or unique identifier, associating them with entries in GO. The GO hierarchy and the annotations within each related entry can then be used to extract useful information about that protein. This is more than just an annotation that might be added to the protein’s record. It provides multiple levels of classification.

So, properly organized, GO can be used to answer questions like (relate to use examples...)

```
> (is? "nadh dehydrogenase" a "transporter")
T
> (is? "nadh dehydrogenase" an "atpase")
NIL
```

In order to do such queries, the textual information in GO must be converted to a symbolic form. Each GO entry contains a unique identifier (within the GO knowledge base), and several pieces of information identified by tags. It is a good fit to the frame idea described in Chapter 3, Section 3.3.1. In this frame representation, as an example, the GO entry for "nadh dehydrogenase" then looks like this:

```
(0003954
  (NAME "NADH dehydrogenase")
```

```
(SYNONYM "cytochrome c reductase")
(DEFINITION "Catalysis of the reaction: NADH + H+ + acceptor
            = NAD+ + reduced acceptor. definition_")
(isa 0015933)
(isa 0016659))
```

The entry contains links to the classes of GO concepts of which it is a specialization, identified by GO ID's 0015933 and 0016659. These in turn have “isa” links, and others. Entry 0015933 is

```
(0015933
 (NAME "flavin-containing electron transporter")
 (DEFINITION "An oxidoreductase which contains either
              flavin-adenine dinucleotide or flavin mononucleotide
              as a prosthetic group, utilizes either NADH or NADPH
              and transfers electrons to other electron transfer
              proteins. definition_")
 (isa 0005489))
```

We can continue to follow the “isa” links until we find

```
(0005215
 (NAME "transporter")
 (DEFINITION "Enables the directed movement of substances
              (such as macromolecules, small molecules, ions) into,
              out of, or within a cell. definition_")
 (isa 0003674))
```

and thus the answer to the query is “yes.”

A small program to do this would look up the first name in the GO list, extract the GO ID's of its “parent” classes, labeled by the ISA tags, and recursively follow each of those, until it either found the second name in an entry, or reached the top of the GO hierarchy without success. In the first case, the search would be successful and return `t` and in the second case, `nil`. This is exactly the subsumption query that is described in Chapter 3, Section 3.3.1.

So, we need a way to search the GO entries until we find a match to the name we are starting with. Let's start with all 13,318 GO entries in a big list with each entry structured like the above examples. This list is the value of a variable named `*frame-kb*` as in Section 3.3.1. However, the `subsumed-by` function uses the frames themselves, and it would be more convenient to look up frames by name. Looking up entries in GO by name is just a linear search with string matching.

```
(defun find-frame-by-name (name)
  (find name *frame-kb* :key #'name :test #'string-equal))
```

Note that the `name` function here is the one that is defined as a slot accessor in the frame system of Section 3.3.1 in Chapter 3.

Then we can query the Gene Ontology frame list (or any similar frame KB) with

```
(defmacro is? (name1 connector name2)
  '(subsumed-by (find-frame-by-name ,name1)
    (find-frame-by-name ,name2)))
```

The above code is inefficient, but it does logically what we need.<sup>1</sup> Moreover, it is more efficient than pattern matching on strings, which is more typical of how one would approach this problem if programming in Perl. Symbol manipulation, lookup and comparison are very fast, essentially implemented by single machine instructions.

\*\*\* Add getting everything below a given node, i.e. returning a subtree, use functions from Chapter 3

### 7.3 Gene Expression

\*\*\* use data from GEO repository (<http://www.ncbi.nlm.nih.gov/geo>), refer also to Zak Kohane

Using the Gene Ontology to assist in analyzing microarray data provides an opportunity to illustrate the ideas of classification, inheritance (specialization), and knowledge based data analysis. It also illustrates the importance of transforming annotations, or knowledge in the form of human-readable text, into computable form, that represents logical relationships and structure, not just searchable text. This is of growing importance in molecular biology. Until recently, computational biology research has focused on using statistical methods to discover relationships in data, without much a priori knowledge about the genes and gene products being studied. Existing data resources such as GENBANK and others at NCBI then could be used with pattern matching to guess at the function or role of a newly discovered gene or protein, or guess at a new pathway. More recently, a wide range of strategies to use existing knowledge to analyse data have become important.

Something like the use of GO to classify data from gene expression experiments, like Shrager's analysis of the Hihara data.

\*\*\* also add here examples of information theory in molecular biology

---

<sup>1</sup>The second variable input to `is?`, the `connector` variable, is ignored, and is just there to allow typing the word "a" or "an" to make it look impressive when using the function.

## Chapter 8

# Biological Networks

Work done on modeling biological and medical phenomena with networks is a very large field. I intend to take a few examples and develop them to illustrate the principles and methods, rather than try to create a comprehensive survey of this subfield of biomedical informatics. The examples will include:

- the EcoCyc project and related metabolic pathway resources, KEGG, Reactome (Cold Spring Harbor), Lincoln Stein
- the Pathminer project [86], which predicts pathways from basic compilations of biomolecular reactions,
- Petri net models such as the work of Peleg [101] which includes a model of malaria parasite invasion of host erythrocyte cells, and the work of Alex Barke on modeling of radiation treatment machinery [7].

Maybe split this up into several chapters...

### 8.1 EcoCyc - Using a Metabolic Pathway Model

Provide here a GUI view of EcoCyc, as well as examples of actual frames, include description of PathoLogic. Other examples provided by PK, include biological explanations.

```
(in-package :ecocyc)
```

```
;; Given a reaction frame id, returns the balance of the reaction as  
;; an alist that specifies the element difference between the  
;; reactants and products of the reaction. Each tuple in the alist  
;; gives the number of atoms of one element in that difference.
```

```
(defun rxn-balance (rxn)  
  (let (left right diff)
```

```

    (setq left (sum-rxn-side rxn 'left))
    (setq right (sum-rxn-side rxn 'right))
    (setq diff (sum-rxn-side rxn 'right left -1))
    (format t "~%Left: ~A~%Right: ~A~%Diff : ~A~%~%"
      left right diff)
    diff
  ))

;; For a specified side of a reaction, compute the sum of the
;; element counts for all compounds in that side of the reaction.
;; The sum is returned as an alist where each tuple in the alist is
;; of the form (ELEMENT . COUNT). The sum can be computed by itself
;; or it can be folded into an input alist, and can be multiplied by
;; sign (+1 or -1).

(defun sum-rxn-side (rxn side &optional ecounts (sign 1))
  (let ()
    (loop for cpd in (get-slot-values rxn side)
      for coeff = (* sign (coeff-of-substrate rxn side cpd))
      do
        (loop for atom in (get-slot-values cpd 'structure-atoms)
          for tuple = (assoc atom ecounts)
          do
            (if tuple
              (rplacd tuple (+ (cdr tuple) coeff))
              (push (cons atom coeff) ecounts))
            ))
    )
    ecounts
  ))

;; Return the coefficient of cpd in side of rxn.

(defun coeff-of-substrate (rxn side cpd)
  (let ((value (gfp:get-value-annot rxn side cpd 'coefficient)))
    (when (and value (typep value 'double-float))
      (setq value (coerce value 'single-float)))
    (or value 1)
  ))

;; =====
;; Rank substrates by frequency with which they are reaction
;; substrates.

(defun frequent-reaction-substrates ()
  (reset-counts)

```



```

(loop for r in (all-rxns :metab-smm)
  do
(loop for substrate in (append (get-slot-values r 'left)
  (get-slot-values r 'right) )
  do
    (countx (if (coercible-to-frame-p substrate)
      (get-frame-name substrate)
      substrate) )
    ))
  (sort-counts)
  (n-counts :max-count 5)
)

;; Rank substrates by frequency with which they are pathway
;; substrates.

(defun frequent-pathway-substrates ()
  (reset-counts)
  (loop for p in (all-pathways)
    do
(loop for substrate in (substrates-of-pathway p)
  do
    (countx (if (coercible-to-frame-p substrate)
      (get-frame-name substrate)
      substrate) )
    ))
  (sort-counts)
  (n-counts)
)

;; Rank reactions by frequency with which they are pathway members

(defun frequent-pathway-reactions ()
  (reset-counts)
  (loop for p in (all-pathways)
    do
(loop for r in (get-slot-values p 'reaction-list)
  do (countx (get-frame-name r) (get-frame-name p))
  ))

  (sort-counts)
  (n-counts :max-count 2)
)

```

## 8.2 PathMiner - Computing Pathways From Reactions

(also some variation of Shrager's metabolic pathway search)

## 8.3 Petri Net Models

Mor Peleg's work, and Alex Barke's project. Spatio-temporal effects and models, e.g., Cell++, John Parkinson, U of Toronto (talk from Rocky '06).

## Chapter 9

# Modeling Biological Structure

\*\*\* intro to anatomy, physiology

### 9.1 The UW Foundational Model of Anatomy

The University of Washington Foundational Model of Anatomy (FMA) is a formal model of human anatomy. The FMA is built using Protégé, an ontology editor available from the Stanford Protégé Project<sup>1</sup>. It includes entries for every element of human anatomy from the body, progressing in levels of detail down to the cellular and subcellular levels. Most important, the FMA represents *relations* between entities, not only in terms of an **is-a**, or subsumption, hierarchy (class - subclass relationships), but also other relationships such as composition (various **part-of** relations), spatial relations and connectivity (e.g. for the blood vessels and the lymphatic systems).<sup>2</sup>

#### 9.1.1 The Components of the FMA

\*\*\* more description here, from the web site or the papers, maybe a Protégé screen shot, and also do a piece of the FMA in the simple frame language of Chapter 3

#### 9.1.2 FMA Projects

OQAFMA, Emily, the Dynamic scene generator, etc.

---

<sup>1</sup>URL <http://protege.stanford.edu/index.html>

<sup>2</sup>A traditional example of the **connected-to** relation applied to the skeletal system may be found at <http://www.niehs.nih.gov/kids/lyrics/bones.htm>

### 9.1.3 A Simple Network Interface - the FMS

In addition to OQAFMA (actually predating it by a decade or more) there is also a simpler interface to the FMA, for queries from network clients. It is known as the Foundational Model Server (FMS). The FMS accepts queries in a Lisp-like syntax, using a functional interface something like OKBC (see Chapter 3), but simpler. An FMS query consists of a string that looks like the printed representation of a Lisp list starting with a command symbol (analogous to a function name), then zero or more parameters, which, depending on the command, may be strings, numbers, `t`, or `nil`. Here is an example query, asking for the constituent parts of the heart (constitutional parts are those that make up an anatomical entity, while regional parts are spatial subdivisions),

```
(fms-get-children "Heart" "constitutional part")
```

and here is the response from the FMS:

```
( "Coronary sinus" "Interatrial septum" "Interventricular septum"
  "Atrioventricular septum" "Tricuspid valve" "Mitral valve"
  "Aortic valve" "Pulmonary valve" "Wall of heart"
  "Cavity of right ventricle" "Cavity of left atrium"
  "Cavity of left ventricle" "Fibrous skeleton of heart"
  "Cavity of right atrium" "Cardiac vein" "Right coronary artery"
  "Left coronary artery" "Systemic capillary bed of heart"
  "Lymphatic capillary bed of heart" "Neural network of heart" )
```

The query above is not a function call in a Lisp program. It is a string of characters sent to the FMS through an open socket. Similarly, the return value is a long string (without newline characters) that is read back from the socket. However, it is easy to construct strings like this in a Lisp program, and even easier to parse the result string (the Lisp built-in function, `read`, does this).

Our plan is to create a programmatic interface that makes a connection to the FMS, handles arbitrary queries and returns results, so that we can write complex and powerful programs that use the FMA as a knowledge resource, through the FMS. This interface layer will consist of three basic functions. The `term-connect` function will connect to the FMS and return an open stream that can be used to send and receive queries. The `term-query` function will take a list specifying a query according to the FMS query language, convert it to a string, send the string, read back the result and return it as a list of strings (terms from the FMA). Finally, the `term-disconnect` function will close the stream and socket connection when the application no longer needs to refer to the FMA.

Since we are using strings, or terms, to label the entities that are represented in the FMA, we can write some functions that handle the details of making queries from a reasoning program. In particular, we will need to open a socket to the host system of the FMS. First we define some useful constants.

```
(defconstant *data-flag* 4)
```

```
(defconstant *end-flag* 3)
(defconstant *fms-host* "quad.biostr.washington.edu")
(defconstant *fms-port* 8098)
```

As there is no standard function for opening a socket, this will depend on the Lisp implementation. Here is a function to connect to the FMS with particular variants for some Lisp implementations. Note that for *Lispworks<sup>TM</sup>*, a non-standard package must be explicitly loaded. For the others, the package will be “auto-loaded” or already in the running environment by default.

```
#+lispworks (require "comm")

(defun term-connect (&optional (host *fms-host*) (port *fms-port*))
  (let ((stream #+allegro (socket:make-socket :remote-host host
                                              :remote-port port)
          #+lispworks (comm:open-tcp-stream host port)
          #+cmu (system:make-fd-stream
                (extensions:connect-to-inet-socket host port)
                :input t :output t :element-type 'character)
          ))
    (format t "~A~%" (term-readback stream))
    stream))
```

The `term-connect` function opens a text stream connection to `host` at the indicated `port`, outputs the server’s greeting to `*standard-output*`, and returns the open stream. It uses another function, `term-readback`, which is called when expecting output from the FMS. The `term-readback` function is called here because the FMS protocol specifies that when the connection is made, the FMS sends a greeting message, then waits for a query. After that, the sequence is that the client sends a query, and the FMS sends a response.

The protocol for the FMS says that whenever a query is made (or when the connection is first made), the server’s response will be one of the following:

```
<message> <data_char> <sexpression> <end_char>
```

or

```
<message> <end_char>
```

and in either case, our `term-readback` function should return the last non-flag item sent. This will be a list if the result is a valid response, or a symbol corresponding to a message, possibly an error message, if there was an error or end of file. The data flag and end flag characters are defined above.

```
(defun term-readback (stream)
  (let (item data)
    (loop
```

this is actually not the protocol, since it really promises Control-C and Control-D, and the 3 and 4 are just there for human readable noise. Fix this later. In this code, the read-line calls are needed to throw away the real flag characters.

```

(setq item (read stream nil :eof))
(cond ((eql item :eof) (return data))
      ((eql item *data-flag*) (read-line stream)
       (setq data (read stream)))
      ((eql item *end-flag*)
       (read-line stream) (return data))
      (t (setq data item)))))

```

With this, we can then define our function, `term-query`, that takes an expression in the right format, either a list, or a string that looks like a list, sends it to the FMS, gets the result, and returns it. The input, `expr` is either a list in the form

```
(fms-get-children "Brain" "subdivision")
```

or the equivalent string (with embedded " characters).

```

(defun term-query (stream expr)
  (if (listp expr)
      (setq expr ;; convert to string with embedded " around terms
            (format nil "~A~{ \"~A\"~}" (first expr) (rest expr)))
      (write-line expr stream)
      (finish-output stream)
      (term-readback stream))

```

Finally, it will be nice to have a wrapper function, `term-disconnect`, that sends the right command to the FMS to terminate the interaction, and that also closes the open socket stream.

```

(defun term-disconnect (stream)
  (term-query stream "(quit)")
  (close stream)
  (format t "~%FMS Connection closed~%"))

```

The FMS protocol supports many kinds of queries, but we will just describe a few here. The idea is to write wrapper functions that look like the FMS query strings, but are actual functions in Lisp. Then a program that uses the FMA just needs to do these function calls.

For the `fms-get-children` query, we define a wrapper function as follows:

```

(defun get-children (stream term relation)
  "returns a list corresponding to the fms-get-children query for
  anatomical entity <term> using hierarchy <relation>"
  (term-query stream (list 'fms-get-children term relation)))

```

We could use this to define yet higher level functions that can be used to retrieve specific structural information. For example, a function to fetch information about the *parts* of an anatomic entity would take the form

```
(defun parts (stream term)
  (get-children stream term "part"))
```

Here is how it would work. First a connection to the FMA is established, using `term-connect`. Then the `parts` function is used to get the parts of the anatomic entity called "Heart". The whole list returned by the FMA has in all 37 items. In the code below, it is assumed that all the functions are defined in a file and loaded before typing the expression you see at the CL-USER(3): prompt.

```
CL-USER(3): (setq fms (term-connect))
; Fast loading /usr/local/acldns/code/acldns.005
;;; Installing acldns patch, version 5
gc: E=66% N=597904 O+=148080 pfu=0+1955 pfg=0+454
CONNECTED
#<MULTIVALENT stream socket connected from
  pippin.radonc.washington.edu/41319 to
  xiphoid.biostr.washington.edu/8098 @ #x716e8372>
CL-USER(4): (parts fms "Heart")
("Coronary sinus" "Great cardiac vein" "Left marginal vein"
 "Posterior vein of left ventricle" "Middle cardiac vein"
 "Small cardiac vein" "Oblique vein of left atrium"
 "Right marginal vein" "Right atrium" "Left atrium" ...)
```

Some other queries supported by the FMS include queries to find out what attributes are specified for a given term, what attributes or relations are supported by the FMA in general, and what are the parent terms in a given hierarchy (relation tree) for a particular term. Here are wrapper functions to construct these queries. The `get-attributes` function will return all the attributes or relations that are defined for a given term (the names of the attributes, not their values).

```
(defun get-attributes (stream term)
  (term-query stream (list 'fms-get-attributes term)))
```

The `get-parents` function will return all the parents of a given term that are related by a specified relation in the FMA.

```
(defun get-parents (stream term relation)
  (term-query stream (list 'fms-get-parents term relation)))
```

The `get-hierarchies` function will return the hierarchies (relationships) that include the specified anatomic entity, or if no term is supplied, it returns a list of all hierarchies or relationships in the FMA.

```
(defun get-hierarchies (stream &optional term)
  (term-query stream (if term
    (list 'fms-get-hierarchies term)
    (list 'fms-get-hierarchies))))
```

There are 138 relations supported by the FMA, including the built-in relations for classes and instances in the Protégé ontology development system. Here is the output of `get-hierarchies`.

```
CL-USER(9): (setq rels (get-hierarchies fms ))
(" :DIRECT-TYPE" " :DIRECT-SUBSLOTS" " :ASSOCIATED-SLOT" " :TO"
 " :DIRECT-SUBCLASSES" " :FROM" " :SLOT-CONSTRAINTS" " :SLOT-INVERSE"
 " :DIRECT-TEMPLATE-SLOTS" " :DIRECT-SUPERCLASSES" ...)
```

As you can see, at first, the Lisp printer only shows the first few entries, with an ellipsis to show there is more. Using the pretty-printer (with the function `pprint`) will show the entire list.

```
CL-USER(10): (pprint rels)
(" :DIRECT-TYPE" " :DIRECT-SUBSLOTS" " :ASSOCIATED-SLOT" " :TO"
 " :DIRECT-SUBCLASSES" " :FROM" " :SLOT-CONSTRAINTS" " :SLOT-INVERSE"
 " :DIRECT-TEMPLATE-SLOTS" " :DIRECT-SUPERCLASSES" " :ANNOTATED-INSTANCE"
 " :ASSOCIATED-FACET" " :DIRECT-INSTANCES" " :DIRECT-DOMAIN"
 " :DIRECT-SUPERSLOTS" "branch" "part" "tributary" "Preferred name"
 "Synonym" "inherent 3-D shape" "related part" "has derivation"
 "has fate" "has shape" "contained in" "orientation" "has adjacency"
 "anatomical plane" "related to" "has location" "related concept"
 "related object" "part of" "general location" "bounded by"
 "is boundary of" "location" "attributed part" "contains"
 "attributed continuous with" "surrounded by" "surrounds"
 "arterial supply" "venous drainage" "lymphatic drainage" "innervation"
 "sensory nerve supply" "member of" "has wall" "input from" "output to"
 "muscle attachment" "fascicular architecture" "muscle origin"
 "muscle insertion" "branch of" "tributary of" "member" "bounds"
 "arterial supply of" "segmental supply" "nerve supply"
 "nerve supply of" "attaches to" "receives attachment" "union"
 "unites with" "form" "segmental contribution from"
 "anatomical morphology" "venous drainage of"
 "segmental contribution to" "attributed branch" "variation"
 "attributed tributary" "germ origin" "cell shape"
 "attributed cell type" "cell surface specialization"
 "NN abbreviations" "chromosome pair number" "cell connectivity"
 "connection type" "connecting part" "connected to" "regional part"
 "constitutional part" "systemic part" "regional part of"
 "constitutional part of" "systemic part of" "clinical part"
 "clinical part of" "NeuroNames Synonym" "epithelial morphology"
 "2D part" "attributed regional part" "attributed constitutional part"
 "Non-English equivalent" "lymphatic drainage of" "2D part of"
 "segmental composition" "segmental composition of"
 "segmental supply of" "primary segmental supply"
 "secondary segmental supply" "primary segmental supply of"
 "secondary segmental supply of" "origin of" "muscle insertion of"
 "volume" "mass" "viscosity" "temperature" "ion content"
 "continuous with" "branch (continuity)" "homonym for"
 "continuous with distally" "continuous with proximally")
```



```
"tributary (continuity)" "custom partonomy" "custom partonomy of"
"primary site of" "primary site" "Metastatic site of" "efferent to"
"afferent to" "adjacent to" "attached to" "nucleus of origin of"
"receives input from" "nucleus of termination of" "sends output to"
"has related dimensional entity" "gives rise to" "develops from")
```

The relations in upper case preceded by colons, i.e., that look like keywords in Common Lisp, are the Protégé built-in relations. These can be used as relation parameters in the above functions, just as one would use the FMA-specific relations. So, with this interface one can write a program that obtains implementation details of the Protégé classes themselves, and perform computations on them.

\*\*\* add here something about how this maps to OKBC

\*\*\* maybe here provide examples of queries, including recursive functions to compute transitive closures

## 9.2 Galen, Others

### 9.3 Ontologies for Molecular Biology

The FMA primarily focuses on human anatomy down to the level of tissues and cells, but until recently did not deal with subcellular structure. Extending it to cover cellular and molecular structure is a work in progress. Other work on ontologies for molecular biology may be included for comparison.

### 9.4 Protein Structure

Possibly something from Valerie Daggett and others here - classification of protein folds and structure.

Queries of interest are (from Andrew Simms): how does the time variation of a particular residue look, among multiple proteins or variants? Is there a docking point around which everything moves? Are there transition states?



## Chapter 10

# Drug Interactions

A drug is a substance that can chemically affect living organisms, human or other. Drugs are most familiar as therapeutic agents, used to treat infections, heart disease, cancer, diabetes and many other illnesses. Drugs are also used as preventive agents and sometimes as diagnostic tools as well. Some drugs, such as caffeine (as found in coffee and tea) and alcohol (in beverages) have become commonplace in social settings. Some of these (and other more dangerous drugs) have been implicated in antisocial contexts as well. Because of the potency and danger inherent in many useful drugs, some are controlled in that they are available only by prescription from a doctor. Narcotics are even more strictly controlled. Drugs have served in some societies as powerful weapons (for example, so-called “poison darts”). They can also be part of a public health measure, such as fluoridation of drinking water.

The action of biological systems (organisms) on the drugs, i.e., drug absorption, distribution and elimination, is called “pharmacokinetics”. The action of the drugs on the organisms, on the other hand, intended or unintended, is called “pharmacodynamics”. Overall, the study of drugs, their properties and interactions, is called “pharmacology”. Several excellent textbooks cover this subject, most notably the encyclopedic work of Goodman and Gilman [46].

\*\*\* chapter guide and motivation here

also define “pharmaceutics”, a little about drug packaging, labeling, etc., and even more important, methods by which drugs are administered

### 10.1 Background on Drug Interactions

In this chapter we focus on one aspect of pharmacology, that of *drug interactions*. The context is that of use by patients of one or more drugs for preventive, diagnostic or therapeutic purposes. When taking more than one drug at a time, a patient may experience effects that can be ascribed to the actions of one drug on the pharmacokinetics or pharmacodynamics of the other. This phenomenon is called a drug-drug interaction. Multi-way interactions are also possible. When talking about binary drug-drug interactions, we refer to the drug whose serum concentration is being affected as the “object drug”. The drug whose action *causes* the concentration

of the object drug to change is called the “precipitant drug”. This relation between object drug and precipitant drug is *not inherently* symmetric, although for some pairs of drugs, each one does affect the other.

Software that can predict drug interactions is very important in clinical practice. There are many commercial products that purport to address this need, as well as experimental systems developed in biomedical informatics research groups. Nevertheless, a recent study found that some of the systems commonly in use today have an alarming failure rate, so much so that they are routinely ignored or bypassed in practice [48]. The article cited notes, “The software systems failed to detect clinically relevant drug-drug interactions one-third of the time.” There is clearly much work to be done.

\*\*\* more background on pharmacokinetics and pharmacodynamics here or in the individual sections below

\*\*\* add stuff on calculating drug dosages, refs to come from PSZ

## 10.2 A Catalog of Drug Interactions

\*\*\* review here the state of the art, i.e., current programs and dbs are actually just catalogs

\*\*\* also add some notes on data sources, i.e., evidence, anticipating Richard Boyce’s work later in this chapter

A simple (sparse) table can represent a collection of assertions that drug A interacts with drug B, i.e., the tables on which existing systems rely (e.g., [45]) enumerate known, documented pairwise interactions. Such a table is difficult to maintain and voluminous, particularly when new drugs are added. There is an alternative way interactions might be represented in terms of effects and contraindications, so that each drug record is independent of all the others. This, too, has problems and limitations, and it is the subject of further research to elaborate a sophisticated system. Note that the development here does not deal at all with the problems of consistent nomenclature, dosage forms, or routes of administration.

As discussed in Chapter 2, a simple approach with list structure and symbols as tags will work. No arcane programming language syntax is required. Such structures, lists of symbols, form data that can be processed directly by a Lisp function.

An entry from Hansten and Horn [45] could be represented in essence as follows:

```
(carbamaepine (aka (carbatrol tegretol))
  (interacts-with fluvoxamine)
  (abstract "...lots of text")
)
```

This only captures the essential information, but is enough to be able to search a collection of such records. Each record for drug 1 is retrieved to see if drug 2 is in its **interacts-with** field. There may be multiple records for each drug, so that in order to ascertain if two drugs interact, many records may need to be examined. Another approach is to have a single entry for each drug; the **interacts-with** field then would contain a list of all drugs that interact with the drug.

\*\*\* some simple code here

A realistic system augments the above representation with many more fields than alternate names and interactions. Typical systems include textual information similar to that available from, for example, MicroMedex [79], or other compendia on drugs. Thus we could write:

```
(carbamaepine (aka (carbatrol tegretol))
  (classes analgesic
            anticonvulsant
            ...))
  (dosage-forms
    (adult "...lots of text")
    (pediatric "...different text"))
  (contraindications "...")
  (precautions "...")
  (interactions "...")
)
```

\*\*\* discuss differences and similarities among systems?

So far, the problem is mainly one of information organization and retrieval. There are many solutions, among them a web interface to a database. Thus, such a system would emphasize information lookup, presentation and interpretation by the person using it. The main programming job here would be that of indexing and search for appropriate records, followed by formatting the results.

\*\*\* expand to discuss the role of ontologies in organizing and searching this information, with cross reference back to Chapter 5

## 10.3 Reasoning About Pharmacokinetics

The motivation for this part is to predict stuff from more basic data rather than depending on having a complete set of clinical observations. The catalog approach just reports what was seen.

The development in this section is based on work by Richard Boyce, in collaboration with Carol Collins, Tom Hazlet, John Horn and the author [12].

It is possible to identify and model pharmacokinetic origins of drug interactions. These interactions are consequences of mechanisms which are known for many drugs.

A typical course in a pharmaceuticals program would catalog and document a wide range of these mechanisms. Here are a few examples:

- For orally administered drugs whose primary route of absorption into the bloodstream is through the intestine, the serum level (concentration in the bloodstream) can be strongly affected by the condition of the intestine. Low intestinal motility (as caused by some narcotics) will increase the time the drug spends in the intestine and may increase the drug's serum level. Conversely, if there is damage to the gut, or the person is having intestinal problems such as diarrhea, absorption could be dramatically reduced.
- Drugs that are metabolised by enzymes such as those in the Cytochrome P450 (CYP) family may be strongly affected by other agents that interfere with the production of those enzymes. Many drugs are known to inhibit (reduce below normal) or induce (increase above normal) the production of various CYP enzymes, which in turn may increase or decrease the metabolism of the drugs those enzymes metabolize.

$$\begin{aligned} \forall(x, y, z) : \\ & \text{Inhibits}(x, z) \wedge \\ & \text{PrimaryClearanceEnzyme}(z, y) \Rightarrow \\ & \text{ReducesClearance}(x, y) \end{aligned}$$

$$\begin{aligned} \forall(x, y, z) : \\ & \text{Induces}(x, z) \wedge \\ & \text{PrimaryClearanceEnzyme}(z, y) \Rightarrow \\ & \text{IncreasesClearance}(x, y) \end{aligned}$$

explain “prodrug”

- A change in pH can affect the rate of absorption, i.e., if a drug,  $y$ , inhibits the hydrolyzation of an acid-hydrolyzable prodrug,  $x$ , then a reduction in  $y$ 's absorption follows.

$$\begin{aligned} \forall(x, y) : \\ & \text{Prodrug}(x) \wedge \\ & \text{OrallyAdministered}(x) \wedge \\ & \text{AcidHydrolyzable}(x) \wedge \\ & \text{InhibitHydrolyzation}(y) \Rightarrow \\ & \text{ReducesGI Absorption}(y, x) \end{aligned}$$

A drug interaction resource that utilizes these mechanisms would take a very different shape than a simple table of pairwise interaction entries. Instead, for each drug we would catalog (in symbolic form so a program could use it) the above information, i.e., what are its effects on various body systems, what enzymes does it inhibit or induce, what enzymes metabolize it. Then we would use a simple reasoning system to correlate entries and check if two drugs interact according to the data on each. Such a system could predict interactions even if they have not been clinically reported.

The fact that one drug affects the enzyme(s) that metabolize another drug is not sufficient to be certain that an interaction will be significant in a person. But it is the beginning of the process of building a more complex theory that could reliably make such predictions. In the following, we sketch out this rudimentary theory and leave it to the reader to further develop it. In particular, we will only model the CYP enzyme system, and not consider absorption or circulation.

### 10.3.1 Using Knowledge About Enzymatic Metabolism

We will represent the various CYP enzymes by symbols, such as `cyp1a2`, `cyp3a4`, and `cyp2d6`. We use symbols corresponding to drug names as tags to label each drug record. For each drug we have an entry, rather than for each pairwise interaction. Here is how such entries might look:

```
(coumadin (metabolized-by cyp1a2 cyp2c19 cyp3a4)
  ...
)

(fluvoxamine (metabolized-by cyp1a2 cyp2d6)
  (inhibits cyp1a2 cyp2c9 cyp2c19 cyp3a4)
  ...
)
```

Some drugs, such as those above, indeed have multiple enzymatic pathways. One of these may be a primary pathway, but the existence of others makes the problem complicated. If drug A inhibits enzyme 1, which is only one of the metabolic pathways of drug B, will this be significant? It depends on the importance of enzyme 1 compared to others that metabolize drug B. From looking at these entries it is apparent that taking the antidepressant fluvoxamine while also taking coumadin can cause a significant increase in the blood concentration of coumadin since *all* the metabolic routes of coumadin are inhibited by fluvoxamine. So, in that case, we can draw a stronger conclusion than for the case where only one enzyme is inhibited or induced.

There is yet another complication: some people are missing the genes for production of certain CYP enzymes. These people are known as “poor metabolizers”. In those patients, the effect on the remaining enzyme(s) would be much more significant than for normal subjects, since they do not have the alternative path(s) available. This connection with genetics is discussed in Section 10.5 below.

A drug interaction system that relies on pairwise entries cannot represent this information, except to include volumes of text in the abstract. But a reasoning system can compute these cases. If every enzyme in the `metabolized-by` list of drug A is in the `inhibits` list of drug B, that will cause an interaction because no pathways are left for drug A to metabolize. We define a function, `inhibition-interaction`, that compares the `inhibits` properties of the precipitant drug with the `metabolized-by` properties of the object drug, and a similar function `induction-interaction` that checks the `induces` property of the precipitant drug.

```
(defun inhibition-interaction (precip obj)
  (null (set-difference (metabolized-by obj)
                       (inhibits precip))))
```

```
(defun induction-interaction (precip obj)
  (null (set-difference (metabolized-by obj)
                       (induces precip))))
```

\*\*\* the above code is parallel to or equivalent to the corresponding code in the pharmacodynamics section, which uses `intersection` - should be using the same structures and functions wherever possible

Three way or more complex interactions occur when a drug can be metabolized by multiple enzymes (as above) and each of several other drugs inhibits only one or two of them, but the combination eliminates all pathways. This would not be found in the binary drug interaction compilations, and the labor to create a database for it would grow exponentially with the number of enzymes involved. The approach of modeling the underlying mechanisms requires only one entry for each drug, and allows the functions or rules to compute the consequences. For three way interactions, where there are two precipitant drugs and an object drug, we have,

```
(defun inhibition-interaction-3 (precip-1 precip-2 obj)
  (null (set-difference (metabolized-by obj)
                       (append (inhibits precip-1)
                              (inhibits precip-2)))))
```

```
(defun induction-interaction-3 (precip-1 precip-2 obj)
  (null (set-difference (metabolized-by obj)
                       (append (induces precip-1)
                              (induces precip-2)))))
```

and, generally for any number of precipitant drugs, we collect all the `inhibits` or `induces` properties (using `mapcar`), and append them together, before taking the set difference with the `metabolized-by` list. In the following, we reversed the order of the arguments, so that the variable number of precipitants can become a `&rest` argument. We also have to operate on all the precipitants with the `induces`



function and append the whole bunch together. The handy function `mappend` does this. It is so useful, it is worth defining as a utility.

`mappend` is defined in the Radiation Oncology chapter – maybe define here instead, or in some chapter in Part I?

```
(defun inhibition-interaction-n (obj &rest precips)
  (null (set-difference (metabolized-by obj)
    (mappend #'inhibits precips)))))

(defun induction-interaction-n (obj &rest precips)
  (null (set-difference (metabolized-by obj)
    (mappend #'induces precips)))))
```

### 10.3.2 A Rule-based Drug Interaction System

\*\*\* this is verbatim from the AMIA paper - rewrite book style

\*\*\* describe Hansten and Horn “Top 100” and discuss selection of drugs

To better understand the issues of formally representing DDI knowledge, we constructed a simple model involving rules on metabolic mechanisms. A large number of DDIs can be explained by metabolic mechanisms, especially for drugs metabolized by the Cytochrome-P450 enzymes (CYP450) enzymes. Furthermore, for many drugs, data exists on their metabolic mechanisms. FDA guidelines encourage detailed investigations into the metabolic mechanisms of a drug and its potential for drug interactions during that drug’s early development.[83] These investigations are often followed by clinical trials to determine the significance of potential drug interactions.[83, 59]

We first created a FOL representation of metabolic mechanisms of drug-drug interaction from the lectures and class notes of a graduate class on drug-interactions. The following shows the rules pertaining to inhibition of clearance; a similar set of rules was implemented for metabolic induction.

```
(<- (metabolic-inhibit-interact ?precip ?object ?enz)
  (and (inhibits-primary-clearance-enzyme ?precip ?object ?enz)
    (narrow-ther-index ?object yes)))

(<- (inhibits-primary-clearance-enzyme ?precip ?object ?enz)
  (and (inhibits-partial-clearance ?precip ?object ?enz)
    (major-pathway ?object ?enz)))

(<- (inhibits-partial-clearance ?precip ?object ?enz)
  (and (inhibits-effectively ?precip ?enz)
    (substrate-of ?object ?enz)
    (primary-clearance-mechanism ?object metabolic)))

(<- (inhibits-effectively ?drug ?enz)
```

Table 10.1: Mapping between strength of inhibition in Reference A and Reference B and the Drug KB used to test mechanism based drug-drug interaction prediction

<i>Mechanism</i>	<i>Drug KB</i>	<i>Reference A [44]</i>	<i>Reference B [21]</i>
Inhibition	weak	weak	weak, very weak
	moderate	n/a	moderate
	strong	strong	n/a
Induction	weak	n/a	weak
	moderate	n/a	moderate
	strong	strong	n/a

```
(and (inhibits ?drug ?enz)
      (or (inhibit-strength ?drug ?enz strong)
          (inhibit-strength ?drug ?enz moderate))))
```

We then constructed a small drug knowledge-base (KB) containing the necessary drug facts for inference with the selected rules. Facts on the important metabolic enzymes for 249 currently prescribed drugs were input into the KB from a widely used pocket reference on clinically significant drug interactions[44]. This reference (Reference A) also included facts on each drug's potential for inhibition or induction of CYP450 enzymes.

We augmented our KB with information from a Continuing Education Module containing pharmacokinetic information on drugs commonly prescribed to elderly epileptic patients.[21] In addition to facts on potential CYP450 modulation, this reference (Reference B) listed the relative importance of each drug's clearance enzymes. Several drugs not found in Reference A were also added. Since terms regarding the strength of enzyme inhibition and induction varied between the resources, we constructed the mapping shown in Table 10.1. As of the time of this writing, our drug KB contains facts useful for mechanism based inference for 267 currently prescribed drugs.

The implementation uses a simple pattern matching and backward chaining program taken verbatim from chapter 15 of Paul Graham's popular Common Lisp book [41]. Graham's code uses a Prolog-like syntax, where the macro `<-` is analogous to the Prolog `:-` connector, but as usual in Lisp, prefix notation is used. So, the list expressions have `<-` followed by a head expression and optionally a tail expression. Rules that have multiple terms in the tail use combinations of the operators **and**, **or**, and **not** to combine them.

The following are queries on this knowledge base for any drugs that inhibit or induce the primary clearance enzyme for another NTI drug whose clearance is primarily metabolism.

```
(with-answer
  (metabolic-induce-interact ?precip ?object ?enz)
  (format t
    "Drug ~A induces ~A, a primary clearance enzyme, of drug ~A~%"
```

```

                (generic-name ?precip) ?enz (generic-name ?object)))
(with-answer
 (metabolic-inhibit-interact ?precip ?object ?enz)
 (format t
  "Drug ~A inhibits ~A, a primary clearance enzyme, of drug ~A~%"
  (generic-name ?precip) ?enz (generic-name ?object)))

```

\*\*\* flesh out later...

\*\*\* represent drugs as instances of a drug class, then use code from Chapter 3 to convert slot values to assertions.

\*\*\* mention here the drug ontology projects, and why generalizing knowledge about specific drugs to the classes in which they are placed is not applicable to metabolic mechanisms (the drug classes are not according to metabolic properties, and drugs within the same class may differ in their metabolic clearance mechanisms)

### 10.3.3 Representing Levels of Evidence

\*\*\* expand with Richard Boyce's "levels of evidence" idea

Drug metabolism knowledge is evolving, incomplete, and supported by varying kinds of evidence in the scientific literature. A flexible scheme for reasoning with such knowledge would provide a way to dynamically specify which assertions about metabolism should be included and which should not. A way to do this is to use a justification-based TMS. The TMS assertions would be generated from an evidence base which includes an ontology of evidence and a means for generating assertions from evidence specifications.

## 10.4 Reasoning About Pharmacodynamics

Another approach is to use known pharmacological (pharmacodynamic) effects of drugs to predict under what conditions an interaction might occur. Here is an example of a drug entry with such information:

```

(aspirin (effects analgesia anti-inflammatory blood-thinner
              platelet-inhibition heart-attack)
 (contraindications gastritis bleeding-tendency
                    platelet-inhibition nausea))

```

The first list after the drug name is a list of effects. The second list is the effects of other drugs that indicate aspirin is not recommended or to be used with caution. So, for example, if a patient is taking aspirin, a potential problem would exist with

adding ranitidine, because one of ranitidine's effects appears in the aspirin record's contraindications list.

```
(ranitidine (effects gastritis gastroduodenal-ulcer)
             (contraindications cardiac-arrhythmia
              liver-toxicity))
```

Note that “effects” includes all the effects of absorbing the drug in the body, not only the intended therapeutic ones. Also, the representation does not distinguish between the intended effects and the others (“side effects”). However, this distinction could easily be incorporated.

The above representation not only is easy to understand, but it is easily processed by a Lisp program. The code to handle such lookups and comparisons is small. The basic algorithm is to retrieve the entries for each of the two drugs, and match the **effects** of each against the **contraindications** of the other. Each list is a mathematical set (of abstract elements, not numbers or strings!). The built in function **intersection** in Common Lisp takes two sets (lists) and returns the elements that are in both sets.

```
(defun interacts (drug-a drug-b)
  (or
   (intersection (effects drug-a)
                  (contraindications drug-b))
   (intersection (effects drug-b)
                  (contraindications drug-a))))
```

The access functions to get the effects and the contraindications for each entry are simple in this representation. Each is a keyword lookup that returns what follows the keyword in that field. Since the first thing in each drug record is the drug name, we examine the rest of the entry to skip over the name.

```
(defun key-lookup (key record)
  (rest (find key record :key #'first)))

(defun effects (entry)
  (key-lookup 'effects (rest entry)))

(defun contraindications (entry)
  (key-lookup 'contraindications (rest entry)))
```

Retrieving the records for aspirin and ranitidine can be done simply with linear search, or in a more elaborate system with a hash table. Linear search is done with the built in **find** function:

```
(defun drug-lookup (drug drug-list)
  (find drug drug-list :key #'first))
```

While this example is still unrealistically simple, it already has a potential to predict interactions that could be clinically significant but for which there is no published clinical data. This begins to address one of the reported problems of current data, that of false negatives. There is still the problem of false positives, but that will require patient specific data and modeling.

The above is a functional representation of the assertion that drugs interact if any of the effects of one is a contraindication of the other. In Prolog, this same information can be represented in assertions and rules. The representation for knowledge about aspirin is thus rendered:

```
effect(aspirin, analgesia).
effect(aspirin, anti-inflammatory).
...
contraindication(aspirin, gastritis).
contraindication(aspirin, bleeding-tendency).
...
```

and for ranitidine,

```
effect(ranitidine,gastritis).
effect(ranitidine,gastroduodenal-ulcer).
contraindication(ranitidine,cardiac-arrhythmia).
contraindication(ranitidine,liver-toxicity).
```

Then we can represent the interaction concept as a pair of rules with variables:

```
interacts(X,Y,Z) :- effect(X,Z),contraindication(Y,Z).
interacts(X,Y,Z) :- effect(Y,Z),contraindication(X,Z).
```

Then one could query the system as follows:

```
?- interacts(aspirin,ranitidine,M).
M = gastritis
?- interacts(X,ranitidine,Y).
X = aspirin
Y = gastritis
... (possibly other answers)
```

Much more can be done, by leveraging the existence of ontologies, or classification systems, for drugs. Aspirin is one of a class of many drugs that share common effects and contraindications, so lookup by class is potentially more efficient and a more compact knowledge representation scheme, as in e.g. [81]. These authors report a scheme that aggregates drugs into “rollup groups” and “families.” Now, with comprehensive multi-level drug ontologies, a range of granularities for organizing this information is possible.

The important point here is that the usual drug classification systems are in fact based on the effects of drugs, i.e., their therapeutic uses. Thus, in contrast to

pharmacokinetic reasoning, here one *can* use class based knowledge to reason about pharmacodynamics.

How could the above be handled with a drug ontology? Each entry would have an “is-a” link to its immediate parent class (here accessed by a function named **drug-class**). Each drug would need to be located in the hierarchy, and the effects and contraindications would need to be checked at each level against each of the others. A simple algorithm does the checking. The version here is recursive, because it is stepping through a hierarchy. It is of order  $n^2$ , where  $n$  is the depth of the drug class hierarchy. This is the same as an iterative algorithm would be, but the iterative algorithm would be much less transparent.

\*\*\* rewrite the following using the frame system of Chapter 3

For drug A, the algorithm starts with drug B and checks against it and its parent classes until the top of the classification system is reached (i.e., the **drug-class** field of the drug record is empty, or **nil**). We assume in the following code that classes of drugs have the same structure (effects, interactions, etc.) at every level.

```
(defun all-b-interacts (drug-a drug-b)
  (if (drug-class drug-b)
      (cons (interacts drug-a drug-b)
            (all-b-interacts drug-a (drug-class drug-b)))
      (interacts drug-a drug-b)))
```

This is repeated for each successive parent class of drug A.

```
(defun all-interacts (drug-a drug-b)
  (if (drug-class drug-a)
      (append (all-b-interacts drug-a drug-b)
              (all-interacts (drug-class drug-a) drug-b))
      (all-b-interacts drug-a drug-b)))
```

Each function is simple, and follows the standard form of a recursive function. Each builds on the lower level ones, so it illustrates good layered design. The two functions above do not depend in any way on the actual details of the drug class representation, only that the representation is uniform, i.e., that the **interacts** function can be applied to any pair of drug classes.

\*\*\* actually by making **interacts** a generic function, it might be possible to even allow variations in the class representation

## 10.5 Extensions

There is an opportunity to extend this approach to patient specific reasoning, because of the existence of a basic research resource, the Pharmacogenetic Research

Network and Knowledge Base consortium (PharmGKB) [95, 113], which will accumulate and organize emerging knowledge about genetic variation, functional variation and its relation to clinical drug responses. The PharmGKB project is primarily aimed at supporting research, but the knowledge base will likely have tremendous clinical use in the future. It is not hard to see how the representations above can be expanded to include genetic variations and rules or functions to compute the implications. The project is using Protégé [91] to build the knowledge base, but is also using a relational schema to store the same information. An XML schema provides a way to import data from other sources. Protégé provides an application programming interface to ontologies created and maintained with it.





## Chapter 11

# Using Medical Guidelines

This chapter will show how to write programs that actually utilize medical guidelines written in some of the standard formats and representations, such as GLIF [99], Asbru [118], MLM (Arden syntax), etc. This is also an opportunity to work up examples that deal with time-oriented medical data.

### 11.1 GLEE

Notes on conversation with Dongwen Wang, author of GLEE [136].

The GLIF3 guideline model is implemented as a Protégé ontology. Guidelines are instances of the model classes. At <http://www.glif.org> one can download the GLIF model. The web site also has the detailed specification.

GLEE has an interface to the patient record; at the moment this is customized to the ECLIPSIS system at Columbia. The development of a more standardized, non-proprietary interface is an active research area. The “Virtual Medical Record”, derived from (or based on) the HL7 RIM, is a scheme for this. It translates between a standard model and the local EMR system.

GLEE details: a server handles actual logic of the guidelines. The server receives requests from clients, using TCP sockets. Clients are built using a client library, with a defined API. The API is not yet published.

DW envisions two uses in the context of CPOE: alert generation and automatic suggestion of an order list.

GLEE generates a “trace record”, encoded using XML, which augments the clinical data repository. It records all the information about the execution.

Guidelines are written using the Guideline Expression Language (GEL). It encodes the logic of a guideline, and this is one of the slot values in the Protégé frame for a guideline. A GEL interpreter is part of GLEE.

GELLO is an object oriented extension of GEL that connects to the Virtual Medical Record, and provides an object structure for patient data.

GLEE is about one hundred thousand lines of java.

\*\*\* note that here also the “curley braces” problem should be mentioned.

## 11.2 Other

Notes from PSZ discussion, April 2006.

Discrete event simulator with time priority queue. Whenever you do anything, schedule a followup. System in Boston (B.I.D.M.C.) at Brigham and Women’s sends alerts when abnormal lab values occur, schedules when to worry about stuff and follow up. They were able to cut response time to abnormal values by a factor of 3.

References:

- For medical logic modules (Arden Syntax):

<http://www.dbmi.columbia.edu/homepages/wandong/KR/krarden.html>

## Chapter 12

# A Nursing Expert System

Catherine D'Ambrosio, who recently completed a Ph.D. in nursing at the University of Washington, had many years of experience in private practice, providing home nursing care for patients with dementia and other chronic illnesses. She came to her doctoral studies with the idea that individual care plans for her patients should be possible to generate automatically with a computer program. Having already experimented with templates and done considerable review of work on “nursing diagnosis” and controlled vocabularies, she decided that these did not lead to a solution. She proposed that nursing knowledge in this area could be formalized in production rules.

Catherine used the simple mock Prolog inference code from Graham [41, chapter 15], and implemented nursing knowledge about prompted voiding for patients with dementia [24]. Using the rules with the nurse’s observations and data from the patient chart, backward chaining could generate a list of patient-specific procedures for the nurse to follow. To test her representation, she created several artificial test cases and had a panel of experts rate the system’s recommendations for sufficiency, accuracy and appropriateness.

Here is an example rule:

If the patient (?PT) has MMSE score between 10 and 20 (COG)  
and the patient has chronic urinary incontinence (CUI)  
and the patient’s Katz ADL score is less than 5.8 (KATZ)  
Then the patient should be put on prompted voiding (PV)

```
(←- (pv ?pt) (and (cog ?pt) (cui ?pt) (katz ?pt)))
```

and another

If the patient (?PT) has bladder function (HAB)  
and does not have acute urinary incontinence (NOTAUI)  
and urinary incontinence frequency more than 2 per week (FUI)  
Then the patient has chronic urinary incontinence (CUI)

```
(←- (cui ?pt) (and (hab ?pt) (notauai ?pt) (fui ?pt)))
```

The entire system includes 124 such rules, and considerable input/output code. The essential ideas can be expressed in a much smaller version, which will be presented in this chapter.

This project illustrates several common challenges in practical rule-based system design. The toileting decisions analysed by D'Ambrosio involve non-monotonic reasoning. This was not simple to represent in a forward chaining or backward chaining system. The granularity required in the rules was difficult to decide, i.e., how much can be lumped or compiled, and how much to leave as low level details.

## Chapter 13

# DICOM: Medical Image Information Agents

Of the many interesting aspects of medical imaging, the focus in this chapter is on the communication of medical image data between computer systems, including those that produce the images. This communication problem can be solved by thinking of the images and associated data as the subject of a message exchange. One way to implement the message exchange is to represent the grammar and syntax of the messages in a way that can utilize what is known about natural and formal language processing. Here we show how to do it for the medical image data exchange protocol called DICOM.

### 13.1 A Short History of DICOM

In the early 1990's a standard emerged for computer programs that transmit and receive medical image data between computer systems that have medical application software producing or using those images. These systems include diagnostic radiology systems such as CT and MRI scanners and radiation therapy planning systems (discussed in Chapter 14) as well as systems and software used in many other areas of medical practice. Chapter 2, Section 2.2.4 provides some background and history of digital medical imaging.

The first version of a standard for interconnection of digital imaging systems, known as ACR-NEMA version 1 (ACR is the American College of Radiology, which was a co-sponsor with NEMA of the standardization project), proposed a new kind of point-to-point interconnection between computers, modeled on proprietary high speed parallel port interfaces available from some computer vendors. This required invention of software protocols for handling the generic task of getting arbitrary data packets reliably from one computer to another, in addition to the digital imaging specific protocols that described the application data and functions. The idea was never implemented, mainly because of the cost of designing proprietary interface hardware and the fact that it was point-to-point. It was later abandoned in favor of

using emerging world-wide standards for the interconnection hardware and general network software protocols for addressing, routing and insuring reliable message delivery.

The new standard that followed, called Digital Imaging and Communications in Medicine (DICOM)<sup>1</sup>, defines a network protocol for communication between computer systems that produce, store, display and print medical images [3, 10]. The DICOM standard provides a specification of what kinds of image data can be transmitted and received, how they may be encoded in digital form, and transmitted as messages over a computer network. DICOM was developed to relieve software authors of the burden of writing specialized programs to decode proprietary file formats used by manufacturers of imaging equipment. It was also intended to facilitate peer to peer communication between radiological imaging systems and application systems over a local or wide area network, without the need to handle intermediate media storage such as tapes or diskettes. Several implementations of DICOM exist, including demonstration and experimental software from university groups (the most notable being a software package developed at Mallinckrodt Institute of Radiology known as CTN, for Central Test Node [89]), vendor implementations for their products (i.e., bundled with CT scanners and other imaging systems), and proprietary libraries for building DICOM applications, such as the Merge Technologies DICOM software libraries [87]. All the current DICOM standards documents are available via the World Wide Web [3].

Although the adoption of DICOM reduced considerably the programming work for achieving data interchange, the approach most programmers took was still to write traditional procedural code that processed the byte stream explicitly following the complex logic of the standard's description of the messages. An excerpt from one such implementation is shown in Figure 13.3. In this chapter you will see another approach, using the idea of expressing a network protocol as a language with a grammar and a vocabulary. The basic idea is that the knowledge about the structure of the messages and the structure of the content can be encoded in rules. Then the problem of parsing and message generation is one of writing a rule-based translator that could process many different kinds of message streams. There is a further simplification in that the message formats can more easily and directly be transformed into rules than into procedural code. The resulting program takes significantly less code than the conventional alternative and has less complexity.

## 13.2 Networks and Protocols

The DICOM standard is based on the ISO Open Systems Interconnect (OSI) model, although actual implementations use TCP/IP protocols rather than OSI protocols. The ISO Open Systems Interconnect (OSI) model is a layered design, in which the various concerns are separated, to enable interoperability of different kinds of computers and different operating systems. The OSI model is described in many fine books on modern networking concepts, such as [23]. Figure 13.1 illustrates

---

<sup>1</sup>DICOM is the registered trademark of the National Electrical Manufacturers Association (NEMA) for its standards publications relating to digital communications of medical information.

how these concerns are organized into layers, with each layer only depending on the layer immediately below, and providing functions (services) only to the layer immediately above.

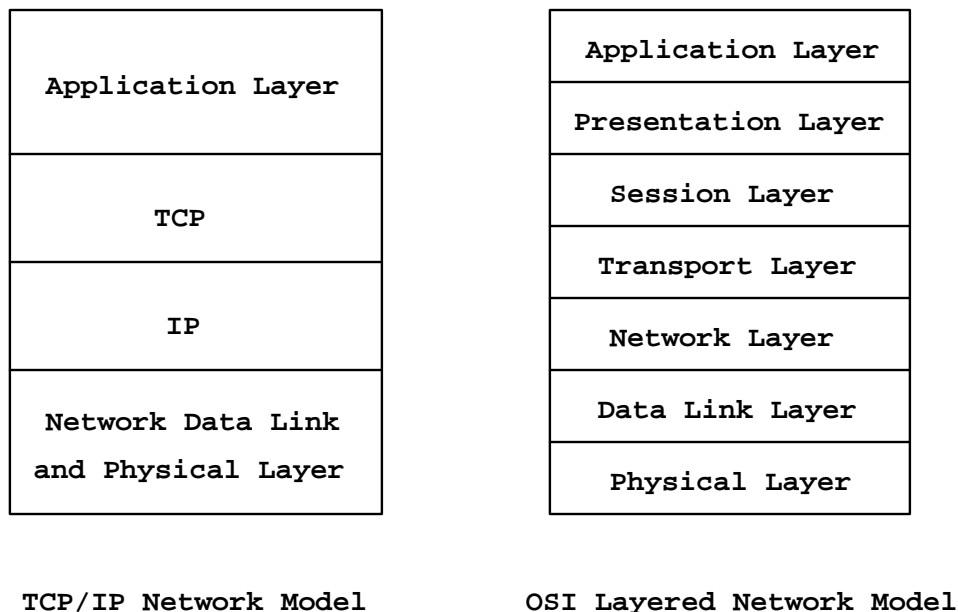


Figure 13.1: TCP/IP and the OSI layered network model

In the OSI model, the data link and physical layers are only concerned with getting packets of data from one computer to another, e.g., the Ethernet standard with all its variants in speed and cable type, fiber optic cable, etc. The network, transport and session layers provide logical pathways between clients and servers, find physical routes through the network, break up long messages into packets, reassembling them at the destination, and perform other functions that should not concern application programmers. The application and presentation layers contain the details of the biomedical application, which might have a fancy user interface, or might just be a server that sends or receives data on request from clients.

In short, the problems of connection wires, signals, insuring error-free transmission, creating and maintaining a logical path from a program on one computer to a program on another computer, are all relegated to a set of standards outside the scope of DICOM. The most common software scheme for this is TCP/IP (also described in [23] and illustrated in Figure 13.1 in relation to the OSI model). The idea of the layered approach is that the upper layers work with interchangeable lower layers and that the upper layer software can be designed without concern for solving the problems addressed by the lower layers.

The TCP/IP layer of the DICOM code presented in this chapter follows the standard designs for network client and server programs. A server program, when

it runs, waits for an incoming connection request and provides services or functions requested by the remote program (the client). A client program is one that initiates a connection to a server, in order to provide the functions of the server to the user of the client program. These programs are generally written using the TCP/IP socket libraries provided for various operating systems. A *socket* is analogous to a file, in that a program opens a socket and performs read and write operations with it. However, the socket represents the input and output of a remote program that similarly has opened a socket for communication over the network. A server opens a “passive” socket, and several additional library procedures ready it to accept incoming connections. The server “listens” on a particular *port*, an abstraction that allows multiple server and client programs to maintain logically independent virtual connections through the same physical interface. A client program opens an “active” socket and specifies the remote entity to connect to by providing an IP address and a remote port number. The TCP/IP socket library functions internally take care of all the details of device drivers, breaking data streams into packets, routing the packets, insuring that the transmitted data are received and reassembled in the correct order, etc. In short, TCP makes each program see the other program as a device with an input and output stream, and neither program has to have any concern about how the data are actually moved from one program to the other. The socket model is well described in standard networking texts such as [23].

What is needed in order to have a working network application? One must be able to open a socket and do reading and writing with it, in order to exchange messages. This is done in different ways in different programming languages but the abstract idea is that a socket is opened analogously to opening a file, with just a little extra detail. In the University of Washington DICOM project [67] we chose for simplicity to just use a Common Lisp vendor’s implementation of a socket layer interface. This is analogous to using high level functions for network access in **java** and other languages. Most Common Lisp systems currently available have such interface functions. Here is the definition for Allegro Common Lisp (ACL)<sup>2</sup> [34]:

add also the CMUCL version  
and a wrapper

```
make-socket ℰrest args ℰkey type connect address-family [Function]
```

Create and returns a socket object with characteristics specified by the arguments.

The **:type** argument can be either **:stream**, which makes a TCP socket, or **:datagram**, which makes a UDP socket. The **:connect** argument can be **:active** (for a client), or **:passive** (for a server). The **address-family** can be either **:internet** or **:file**. The **make-socket** function takes additional keyword arguments, **:local-port**, **:remote-port**, and **:remote-host** depending on whether the socket is active or passive.

Here also add a CMUCL version and a wrapper

The next step for a server is to wait for and accept an incoming connection request. The ACL function for this is:

```
accept-connection sock ℰkey (wait t) [Function]
```

---

<sup>2</sup>Allegro Common Lisp is a software product of Franz, Inc., Oakland, California



If `wait` is `t`, waits for a remote connection request for this socket and returns a new stream object that can be used to do I/O to the connected socket. If `wait` is `nil` and there is no connection request pending it returns `nil` immediately, but if a connection request is pending, it completes the connection and returns a new stream object for input and output on the connected socket.

When the connection is accepted, a function is called (the `applic-fn` parameter here) that reads and writes data according to the specific network application layer protocol, in our case, DICOM. The registered port number for DICOM is port 104, but others could be used. Here is a simple implementation for a generic TCP server program.

```
(defun tcp-server (applic-fn port)
  (let ((socket (make-socket :connect :passive
                             :address-family :internet
                             :type :stream
                             :local-port port)))
    (stream nil))
  (unwind-protect
    (loop
      (setq stream (accept-connection socket :wait t))
      (funcall applic-fn stream)
      (close stream)))
    (close socket))) ;; only here if severe error occurs
```

This server will continue to wait for and accept connections until terminated by an external event such as an operating system intervention or a host computer shutdown. Another possibility is that an unrecoverable error internal to `applic-fn` occurs. In that case, `unwind-protect` catches the error condition, allowing the program to close the socket gracefully and exit.

There are other parameters to `make-socket` which would be used for a fully detailed implementation, but here we just focus on the essential elements. For a network client application, in principle we need a simpler version, since the client does not wait for and repeatedly process connections.

```
(defun tcp-client (applic-fn port)
  (let ((stream (make-socket :connect :active
                             :address-family :internet
                             :type :stream
                             :remote-host host
                             :remote-port port)))
    (unwind-protect
      (funcall applic-fn stream))
    (close stream)))
```

In most applications, the server version of `applic-fn` is somewhat different from the client version. In the case of DICOM, however, it is possible to use the

same function for both. In the above code for a server, the opening and closing of connections is handled outside the implementation of an application protocol. This is an example of what is meant by “layered design”. The protocol simply assumes it has an open channel and deals only with the data that flow through it. Unfortunately, the DICOM protocol does not conform to this design, but mixes up the process of opening and closing connections with the processing of messages. So the DICOM server’s version of `applic-fn` includes the call to `accept-connection` and `close` for the stream. The *server* code then simplifies to:

```
(defun dicom-server (port)
  (let ((socket (make-socket :connect :passive
                             :address-family :internet
                             :type :stream
                             :local-port port)))
    (unwind-protect
      (dicom-state-machine socket))
    (close socket))) ;; only here if severe error occurs
```

The design of the `dicom-state-machine` function will be presented in section 13.4.1.

\*\*\* to be continued – some discussion of wrappers to the socket library itself. Other issues around sockets...

### 13.3 About the DICOM Standard

For this section and the remainder of the chapter, you will find it very helpful to have a copy of certain critical sections of the DICOM standard documents available to look at as we go through the development. All parts can be downloaded as PDF files from the official NEMA DICOM web site, <http://medical.nema.org/dicom/> via the link under “Products” called “The DICOM Standard.” The most immediately useful will be Part 8, “Network Communication Support for Message Exchange.” It is about 50-60 pages.

When a DICOM client program connects with a DICOM server program running on another computer, an exchange of messages takes place. This is the job of `applic-fn` or `dicom-state-machine` in the TCP code. In order to exchange data, the client and server must agree on what operation will be performed and on the data format and syntax of the communication. A DICOM program may be able to send or receive data (or both), respond to queries about the contents of available data sets, or perform other services like printing. The data might be sent as binary numbers in big-endian or little-endian byte order, and may have its data format explicitly specified in the data stream or implicitly inferred from its DICOM identification tag. So, before actual data are transferred, client and server exchange messages to settle which options will be used. Then the data are sent, and the exchange is concluded. A typical sequence of messages is as follows:

1. Client sends a request to server, asking if it (server) can store CT images, using little-endian byte order and implicit value representation.
2. Server responds that it can perform the requested operation.
3. Client sends the data in a sequence of messages.
4. Client requests that the communication be concluded.
5. Server responds that it is OK to close the connection.
6. Client is done (i.e., `applic-fn` returns).
7. Server is done and the loop (inside `dicom-state-machine`) iterates with the server waiting for the next connection request.

DICOM includes other types of messages to handle the possibility that the client requests an operation that the server cannot perform or that something unexpected has happened in the network link.

The collection of message types and conditions under which each is sent are called the “DICOM Upper Layer Protocol,” or DUL. The DUL messages and their sequences are described by a state table and by elaborate detailed byte by byte descriptions. These may be found in Part 8 of the DICOM standard. The states, actions and state transition table are in section 9.2 of Part 8 and the DUL message types and formats are defined in section 9.3 of Part 8.

\*\*\* reiterate some state table and actions stuff here?

The messages of DUL are called “Protocol Data Units,” or PDUs. Some messages are single PDUs, but others, particularly the image data, may be composed of many separate PDUs sent in sequence. This causes implementation difficulties, as we will see later. It appears to have come from the fact that initially the ACR-NEMA committee did not decide to use existing network technology, and therefore had to invent a scheme for dividing long messages into packets, and reassembling them at the receiving end.

An example PDU will illustrate the idea. In the exchange above, the first message that would be sent from the client program is the **A-Associate-RQ** message, requesting the server program to establish an “association,” an agreement about communication details and operations to perform. The byte by byte format of this data stream is diagrammed in the DICOM standard, Figure 9-1 in Part 8.

The sequence of bytes contains a byte specifying the PDU type (a byte is sufficient because there are only 7 types so far), an unused byte, four bytes to be interpreted as a binary integer, the PDU length in bytes, two bytes for the protocol version (currently 1), two more unused bytes, 16 bytes each for character strings (called Application Entity Titles) identifying the client software and the server software (each DICOM program is expected to have a name of some sort). There are 32 more unused bytes and then one or more other items, each with their own format in the byte stream, such as an Application Context Item, one

or more Presentation Context Items and other information. The Application Context Item contains the UID (Unique Identifier) for the DICOM standard, the string 1.2.840.10008.3.1.1.1. This identifies that the sender is using the DICOM protocol. It may seem unnecessary, but the designers anticipated that there might be multiple versions or variants, and this would identify which one is in use. Presentation Context Items specify what data and operations are being requested and/or agreed to, as well as what syntax to use in the data transfer. In this context, “syntax” refers to specification of things like byte order for binary data, whether data values will have their structure (or type) explicitly specified in the data stream, or implicitly specified by their ID tag and the data dictionary, whether image data will be compressed and how, and possibly other issues.

The server typically responds to the **A-Associate-RQ** message by sending an **A-Associate-AC** message, in almost identical format, acknowledging that it can perform the requested operation, and providing some identifying information about itself (the server). An actual byte stream, shown in hexadecimal notation (two hexadecimal digits per byte) on the left, with the corresponding characters on the right (according to the ASCII code, with dots showing non-printing byte codes), is shown in Figure 13.2. This byte stream printout was taken from the log file for an actual interchange between the Prism DICOM Server and a remote client application.

Need to describe Prism or  
Xref to something here

```
SEND-PDU: Sending A-Associate-AC PDU, 17-Jul-2002 18:13:02.
Dumping Outgoing PDU (all 183 bytes):

0200000000b100010000507269736d5f496d6167 .....Prism_Imag
655f5372767250415353504f52545f5251202020 e_SrvrPASSPORT_RQ
202000000000000000000000000000000000 .....
0000000000000000000000000000000000000000 .....1.
322e3834302e31303030382e332e312e312e3100 2.840.10008.3.1.1.1.
2100001a0100000040000012312e322e3834302e !.....@...1.2.840.
31303030382e312e320050000031510000040001 10008.1.2.P..1Q....
00005200001a312e322e3834302e313133393434 ..R...1.2.840.113944
2e3130302e31302e312e3100550000075044535f .100.10.1.1.U...PDS_
312e30                                     1.0
```

Figure 13.2: An actual **A-Associate-AC** packet from an exchange between a DICOM client and the Prism DICOM Server

You can see in Figure 13.2 that this PDU is a response to a request addressed to the server program called **Prism\_Image\_Srvr** which came from a client program calling itself **PASSPORT\_RQ**. The server specifies that it can do the default transfer syntax, which is little endian byte order, and implicit value representation, by providing the UID 1.2.840.10008.1.2, the DICOM standard UID for the default transfer syntax. It assigns this a Presentation Context ID of 1 (you can find the 21 identifying the Presentation Context at the beginning of the sixth row of byte

codes, then count inward one reserved byte, two bytes for the length of the item 1a, then the next byte is the 01 for the ID). The transfer syntax item actually begins where you find a 40 in the same row, then its length 0012 followed by the UID string. The rest of the PDU contains “user information” consisting of the UID of the server program, and its version name string PDS\_1.0.

The seven types of PDUs that are specified by the DICOM standard are (Part 8, section 9.3.1):

1. A-ASSOCIATE-RQ - request an association for the purpose of exchanging data,
2. A-ASSOCIATE-AC - accept a proposed association,
3. A-ASSOCIATE-RJ - reject a proposed association,
4. P-DATA-TF - send/receive commands and data,
5. A-RELEASE-RQ - request termination of an association,
6. A-RELEASE-RP - confirm termination of an association,
7. A-ABORT - abort an association because of some abnormal condition.

The standard specifies a variety of functions that a server or client program may provide, and a variety of image types and other kinds of data with which these functions operate. The functions are called “services” and are given names such as “C-STORE” (for the services that request or perform storage of data from a stream). The combination of a service and a particular kind of object is called a Service-Object-Pair (SOP) Class. Each such SOP class is assigned a UID, such as 1.2.840.10008.5.1.4.1.1.2, which is the UID for CT Image Storage. These are enumerated in Part 6 of the DICOM standard. A requestor of a particular service is known as a Service Class User (SCU), while the application that performs the service is known as the Service Class Provider (SCP).

The kinds of data that DICOM supports, and their representation in message packets, is specified in the form of a data dictionary, based on a hierarchical model of the data elements. This model is not intended to dictate how an application stores data, but only to specify the relations between data items for the purpose of sending meaningful and consistent data. The DICOM Information Model, diagrammed in Figure 7-2 in Part 3 of the standard, illustrates that a DICOM object to be transmitted may consist of many other DICOM objects as component parts. Part 3 of the standard describes what data are included in each information object. For example, Annex C of Part 3, Table C.7-2 lists the attributes of a study. A study in turn may include one or more series, described in Table C.7-4, and a series may contain some number of CT images, where each image has attributes described in Tables C.7-7 and C.8-3. The attributes are each assigned an identifier tag, which references a data dictionary (Part 6 of the standard). The data dictionary specifies what type of datum each element is and how it is encoded.

The commands and data appear in the message stream as Presentation Data Values (PDVs) contained in one or more P-DATA-TF PDUs. A single PDU may

contain multiple PDVs. The structure of a PDV in general is specified in Part 8 of the standard, section 9.3.5.1. The format of the value itself depends on what type of item it is, a patient name, numbers specifying the dimensions of the image, or the image pixels.

\*\*\* diagram PDU with PDVs

Each data element that can be part of a DICOM message has a unique tag, consisting of a group number and an element number, each a 16 bit unsigned integer. These data elements are also associated with a data type or value representation (VR), such as “Application Entity” (a 16 character string), “Date,” “Decimal String,” “Floating Point Single,” “Floating Point Double,” “Integer String,” “Person Name,” and “Sequence of Items.” The details of each of these types are tabulated in Part 5. The data dictionary in Part 6 then lists the tag (in Hexadecimal numbers), data type, and other details for each of the known data elements. An example is “Image Position (Patient),” whose tag is (0020,0032), VR is “Decimal String” (DS), and there are three such values present (for x, y and z). Here is the byte sequence that would appear in a PDU containing this PDV.

\*\*\* put it in...

The DICOM specification is not very formal. It does not use Backus-Naur Form (BNF) notation<sup>3</sup> [92, 138] to describe the protocol. The specification is a narrative together with a series of diagrams describing byte by byte the syntax and contents of messages. The state machine and data dictionary are also described by text and tables. The DICOM specification does include a kind of Entity-Relationship diagram model, which defines the relationships (containment and multiplicity) among the data, but the model provides no information to guide the parsing of messages.

The standard describes many kinds of data and functions that a DICOM conformant program might support. Not all are required to be present. The standard also includes in Part 2 a specification for the format of a *Conformance Statement*. The purpose of the Conformance Statement is to provide a detailed description of exactly which elements of the standard an application claims to be able to perform. This includes which kinds of data objects will be handled, and what operations can be performed on them. By examining a properly written Conformance Statement for each of two DICOM implementations it should be possible to determine whether and under what conditions the two programs can communicate with each other. An example Conformance Statement is the Prism DICOM System Conformance Report [66], available from the author’s web site.

Despite the fact that the standard refers to a state table, includes a data dictionary, and talks about messages and sequences of messages, most DICOM implementations translate all this into procedural code, with the details of the message content and handling of different kinds of conditions represented by formatting operations and conditional expressions in the code. In the CTN implementation [89],

---

<sup>3</sup>BNF notation is a formal syntax for describing a grammar for a language in terms of rules. It is commonly used to provide formal specifications for programming languages.

for example, the structure of each kind of message is coded in a closed subroutine in the DUL library, along with **struct** declarations in the C programming language for parts of the message. Thus a message exchange consists of a series of explicit subroutine calls. Figure 13.3 illustrates this with an excerpt from the CTN code.

```

CONDITION
parseAssociate(unsigned char *buf, unsigned long pduLength,
               PRV_ASSOCIATEPDU * assoc)
{
    CONDITION cond;
    unsigned char type;
    unsigned long itemLength;
    PRV_PRESENTATIONCONTEXTITEM * context;
    ...
    (void) strncpy(assoc->calledAPTitle, (char *) buf, 16);
    ...
    (void) strncpy(assoc->callingAPTitle, (char *) buf, 16);
    ...
    cond = DUL_NORMAL;
    while ((cond == DUL_NORMAL) && (pduLength > 0)) {
        type = *buf;
        ...
        switch (type) {
            case DUL_TYPEAPPLICATIONCONTEXT:
                cond = parseSubItem(&assoc->applicationContext,
                                   buf, &itemLength);
                ...
            case DUL_TYPEPRESENTATIONCONTEXTTRQ:
            case DUL_TYPEPRESENTATIONCONTEXTAC:
                context = CTN_MALLOC(sizeof(*context));
                ...
                cond = parsePresentationContext(type, context,
                                                buf, &itemLength);
                ...
            case DUL_TYPEUSERINFO:
                cond = parseUserInfo(&assoc->userInfo,
                                    buf, &itemLength);
                ...
        }
    }
}

```

Figure 13.3: An excerpt from the CTN DUL library, showing how an A-Associate-RQ message is handled by procedural code (condensed and elided from the actual code, as indicated by ellipses)

The `parseAssociate` subroutine decodes the contents of an A-ASSOCIATE message. It in turn calls a series of other subroutines (for example, `parseSubItem`, `parsePresentationContext`, `parseUserInfo`) which contain explicit code for each of these component parts (`parseSubItem` is used to parse application contexts). The sequence of message exchanges is similarly explicitly coded in other parts of the CTN DUL implementation. This procedural/imperative design style is commonly

practiced and should not be surprising. Commercial software such as the Merge libraries [87] is similarly designed. However, experience with the design of the Prism system [130] suggests that a more declarative approach could provide advantages, including ease of implementation, extensibility, and ease of maintenance.

## 13.4 How to Implement DICOM

Faced with such a huge collection of possible operations and a large variety of data types and formats, it would seem to be a daunting task to write a working DICOM server or client for even a few of the possible functions. Indeed, the CTN core is over 30,000 lines of code. An idea that potentially could simplify the design is to create a new language in which to express these details. Then the task factors into writing an interpreter (a parser and generator) for the language, writing a grammar for the language in the form of rules, and creating a lexicon that describes all the data items.

Because DICOM has two layers of encoding, one for the structure of a PDU, and the other for the data contained in a P-DATA-TF PDU, the design we will develop similarly will have two layers, each with a grammar, a dictionary and an interpreter.

The core code will implement a *finite state machine*, in which the system can be in any of a number of predefined states. As conditions change (messages arrive, results are stored, messages are sent), *events* are generated, which lead to other states via specified *actions*. In this implementation we will need to:

- create a representation of the state transition table and write code for a state machine,
- write the action functions that perform all the different kinds of actions needed depending on the current state and event,
- create a representation of the syntax and semantics of the different PDU types, and implement an interpreter (parser and generator) for PDUs,
- create a representation of the syntax and semantics of the different commands and data contained in a P-DATA-TF PDU, and implement an interpreter (parser and generator) for that.

We will now develop each of these ideas in detail.

### 13.4.1 The DICOM State Machine

The standard defines the *control structure* or sequencing of actions via a *state table* which relates States, Events, and Actions. *States* name situations the system can be in (waiting for a certain PDU, sending data, etc), *events* name conditions triggering certain actions, and *actions* name the operation to be performed in the appropriate situation. A *state table* (or *state transition table*) is simply a table containing an entry for each legal state/event combination. Each entry indicates what action to perform under that condition and what state to enter for the next cycle.



As the transactions progress, information is generated or received, and will be needed in each succeeding state. How will it be maintained? One approach is to define a big and complex data structure, a class definition or a set of class definitions, with slots for all the different variables, etc. However, the DICOM standard is so huge, this would be unwieldy. Also, at any time, only a few of the items are needed. And, as the standard evolves, this data structure would have to be expanded and revised. If there were indeed a slot for each possible DICOM data element, there would be thousands of slots, an unwieldy object to keep around. Another approach is to assign global variables for each possible important datum. This similarly would be unwieldy and almost impossible to maintain, as new kinds of information are supported by the standard.

Fortunately, there is a better way, one that is easy to implement in Lisp. Labeling data with tags is a popular and more flexible approach than predefined structures. An *association list* is a simple way of handling this. We can label items with symbols, e.g., `command` for the ID of a command like `C-STORE`, `calling-AE-title` for the string containing the AE title of the client, and so on.<sup>4</sup> For individual DICOM data elements, each one has a two part tag, consisting of a group number and an element number, as in `(0018,0050)` for image slice thickness. We can represent these as two element lists, and these two element lists can also be used as lookup keys. The association list will grow as data come in, and will shrink as information is used and no longer needed. We call this the *environment*. It is passed as a parameter to all action functions, and they are all expected to return a new environment reflecting any modifications.

Therefore the environment is accessible to any operation that needs values of data fields in the decoded PDUs. In Lisp terms, the environment is represented as a nested association list. That is, the entire environment is a list of components. Each component is a list headed by a keyword symbol naming the component, and the rest of the elements of the component are the values. Since a component contains a list of values, we can represent values of variable multiplicity simply by storing a variable number of elements in a component's value list. Also, this structure is recursive in that a value in a component can itself be another component (that is, another list headed by a keyword symbol naming a field together with its list of values).

Functions which need to retrieve an atomic value (the ultimate number or string contained in a deeply-embedded environment structure) can do so by supplying a list of field names (ie, the keyword symbols naming each field) to a retrieval function that does a recursive search on each value found in a retrieved component, guided by the input list of component names. These lists of field names are like pathnames in a hierarchical file system, or a path specification in a tree-structured data structure. A partial specification results in return of the entire subtree matching the specification provided. For example, a message parsed from a `P-DATA-TF` PDU usually consists of a nested sequence of many data fields. This entire structure, parsed into the syntax tree that represents the environment, can be returned as a single value in

---

<sup>4</sup>“AE Title” is Application Entity Title. Each DICOM software product, or even individual copy of a DICOM program, is identified by a string, such as `UW Prism DICOM Server`. There is no requirement that these be unique, but some kind of name must be specified.

This should eventually point back to a discussion in Chapter 2 about self-describing data and XML

develop further, with an excerpt from a sample environment

an environmental lookup. This nested structure can be passed to other routines (such as a message transmission routine which is formatting a reply) so that all the internal data fields are available using the same small set of environmental accessor functions.

Thus, we will need a function, `item-lookup`, which searches the environment for a specific item labeled by a key. To retrieve such items, we will need to specify a path, or sequence of tags that lead to the desired item in its specific context (the same kind of item might appear in several places in the data). Here is the `item-lookup` function:

```
(defun item-lookup (name env &rest path)
  (let ((pair (if (null path)
                  (assoc name env :test #'eq)
                  (assoc name (rest (item-present? path env))
                          :test #'eq))))
    (if (consp pair) (rest pair) nil)))
```

The `item-present?` function searches recursively down the provided path, returning the last cons referred to by the path.

```
(defun item-present? (path env)
  (if (null (rest path))
      (assoc (first path) env :test #'eq)
      (assoc (first path)
              (rest (item-present? (rest path) env))
              :test #'eq)))
```

In addition to the environment association list, which is used for incoming and outgoing data of DICOM messages, some global variables are needed for the state machine. In particular, some of the action functions will signal an event. The event is represented by a symbol, and we will use a global variable, `*event*`, to hold this value. The `state` variable does not need to be global in the same sense, as it is only set in the body of the state machine main loop. A first sketch of `dicom-state-machine` might then look like this:

```
(defvar *event* nil)

(defun dicom-state-machine (socket)
  (let ((tcp-stream nil)
        (state 'state-01)
        (env nil)
        (trans-data nil)
        (action-fn nil))
    (loop
      (setq trans-data (get-transition-data state *event*))
      action-fn (first trans-data)
      env (funcall action-fn env tcp-stream)
      state (second trans-data))))
```

The system starts in state 1, events are signaled (presumably by the action functions), and the system moves to the next state, processing the next event, and so on. The function `get-transition-data` is not yet defined. It should return a two element list consisting of the name of the action function to call for the state transition, and the name (a symbol) of the next state. The form of this function will depend on how we store information about the states and what to do for each event.

We have left out some important details. How do events actually happen? When (where in the program) are PDUs read from the client, and when are they written? Some of this is easy. Certain action functions will include the writing of a PDU; the action function does not return until the PDU is read by the remote system. An action may also involve storing or retrieving data, or even opening or closing the connection. Action functions in some cases will have to provide an event value, to be used in the next loop iteration. So the operation of the server is expected to be synchronous. We will return to the question of what the actions do and how to code them, after creating the representation of the state table.

Unfortunately, this first attempt at the `dicom-state-machine` function will not work. The action functions write PDUs but do not read them. A look at the definitions of the states suggests that in some states, a PDU read is posted, and the result of reading and processing the incoming PDU also can signal the next event. So, we add code to handle this. In state 1, the system is waiting for a connection, so that is where we will have the call to `accept-connection`. In states 2, 5, 7, 10, 11, and 13, the system is waiting for either an incoming PDU or for the connection to be closed by the remote system. In either case, we post a call to `read-pdu`, and then handle the resulting event. Here is the augmented code:

```
(defvar *event* nil)

(defun dicom-state-machine (socket)
  (let ((tcp-stream nil)
        (state 'state-01)
        (env nil)
        (trans-data nil)
        (action-fn nil))
    (loop
      (case state
        (state-01 (setq tcp-stream (accept-connection socket
                                                             :wait t)
                          *event* 'event-05))
        ((state-02 state-05 state-07 state-10 state-11 state-13)
         (setq env (read-pdu tcp-stream env))))
      (setq trans-data (get-transition-data state *event*)
            action-fn (first trans-data)
            env (funcall action-fn env tcp-stream)
            state (second trans-data))))
```

The `read-pdu` function will use our new (yet to be defined) language for DICOM

messages. We defer this development until section 13.4.3. The function defined above will work well as a server. For a client, the connection is an active one, and in state 1 things will be a little different. Other slight modifications will be needed as well. Later we will augment the code to function as either a client or server, according to a mode parameter.

An important goal in any large software project is to design a modular system, one in which component parts can be understood on their own, without too much (hopefully without any) involvement with the context in which they are used in the system. In our case, we can set as goals the following:

- state transitions (i.e., what will be the next state) should only depend on the current state and the current (most recent) event, and not on details of action functions that signal events,
- action functions should perform well defined operations independent of the state or event that triggered them,
- the reading of a PDU should not depend on the current state, and should only have two effects, to add to the environment and to signal an event.

These goals are consistent with the design of the DICOM Upper Layer Protocol, as specified in Part 8, with a few exceptions that we will address later.

Now we consider how to represent the state transition table (Table 9-10 in Part 8 of the DICOM standard). There are 13 states, and we label them with symbols of the form, **state-*nn*** where the *nn* is a number, like 06 or 11. We similarly label all the possible events with symbols like **event-05**, and for each transition there will be an action function as well. The action functions will get a variety of different labels, depending on the kind of actions they represent, but the idea is the same, each action function is identified by a symbol. For ease of reference to the DICOM standard, we use symbols that match the names in the standard (except that “state” is spelled out instead of being abbreviated “sta”).

A state table then consists of a list of entries, one for each state. Each entry needs to contain the state name, and for each applicable event, the event name and the name of the appropriate action function and next state. For human readability we can add that each state entry will have a string containing lines of text to explain which state it is. We use a string instead of comment lines, because the string will be machine readable, and perhaps can be used later for an on line documentation system.

We can now make a direct translation of the standard’s state table (Table 9-10 in Part 8 of the DICOM standard) into a Lisp data structure consisting of a list of entries as described above and illustrated in Figure 13.4. This state-table representation serves both to document the control structure of the system to the programmer and as machine-readable data to be used directly by the state machine code.

The table entry example shown in Figure 13.4 shows the information associated with the state labeled **STATE-06**. The table entry includes a descriptive comment, the string “Association established, ready for data transfer.” For each event that

may occur in this state, there is a list including the event label, the action function to invoke, and the next state.

```
(state-06
  "Association established, ready for data transfer"
  ((event-09) dt-01 state-06)
  ((event-10) dt-02 state-06)
  ((event-20) dt-03 state-06)
  ((event-21) dt-04 state-06)
  ((event-11) ar-01 state-07)
  ((event-12A event-12B) ar-02 state-08)
  ((event-16) aa-03 nil)
  ((event-17) aa-04 nil)
  ((event-15) aa-01 state-13)
  ((event-03 event-04 event-06 event-13 event-19)
   aa-08 state-13))
```

Figure 13.4: State table entry for Data Transfer state

In some cases, the new state is the same as the original, and in others the new state is different. For example, in state STATE-06, the response to the signaling of EVENT-09, which is a request for data, is to invoke action function DT-01, which constructs and sends the appropriate data packet, and to transfer next to (actually to remain in) STATE-06, i.e., ready to send more data. On the other hand, the response to the signaling of EVENT-11, which is a request to release the communication arrangement, is to invoke action function AR-01, which is the action to release the association, followed by a change to STATE-07, “Awaiting Associate-Release response.”

Where several events are grouped in parentheses (for example, the row containing EVENT-12A and EVENT-12B), the meaning is that any of the listed events signals the invocation of the indicated action function and state transition.

The next state is listed last in each row. A next state of `nil` means that the server exits the DUL interpreter loop, closes the TCP connection to that client, and listens for another connection request. For a client, it means that control will return to the application part of the client code, which can then take appropriate action on behalf of the user.

We create a global variable for the state table, a list of all the entries.

```
(defparameter *state-table*
  '((state-01 "Awaiting establishment of connection"
    ((event-01) ae-01 state-04)
    ((event-05) ae-05 state-02))
    (state-02 "Awaiting A-Associate-RQ"
    ((event-06) ae-06 state-03)
    ...
    ...)))
```

Now we can implement the accessor function, `get-transition-data`.

```
(defun get-transition-data (state event)
  (let* ((state-entry (find state *state-table* :key #'first))
        (evlist (rest (rest state-entry)))
        (ev-entry (find event evlist
                        :test #'member
                        :key #'first)))
    (rest ev-entry)))
```

The first call to `find` gets the entry for `state`, a simple linear search. The first two elements of the entry can be discarded, and the remainder searched for `event`, then the remainder of that is the two element list desired.

The representation as nested lists is easy for a person to read, but obscure for a program to handle, and with two linear searches is not the most efficient. One might be tempted to use hash tables, but there is a simpler way, since the number of entries for each part of the table is small. We can use property lists to make the code much simpler. We store the action function and next state as a two element list, on the property list of the state symbol. The property name is simply the event name and the property value is the list containing the action function name and next state name. So, the `event-05` property of the symbol `state-01` will be the two element list `(ae-05 state-02)`. Then instead of the function `get-transition-data`, in `state-machine`, we just use the built-in Common Lisp function `get`, which handles property lists.

But how will this alternate representation be created? This is the job of a compiler, which we will write later. The important point here is that different representations serve different purposes. A representation that is human readable may not be the most efficient or compact for a computer program, and the purpose of a compiler of any sort (typical programming language compilers are classic examples) is to translate one form into the other.

other topics here: timeouts,  
error handling, the binding  
stack for the state machine,  
other issues

### 13.4.2 Action Functions

Altogether the DICOM standard defines 28 different action functions, including 8 related to the dialogue for establishing an association, two related to data transfer, 10 related to association release, and 8 to handle various abort related conditions. The actions may just involve signaling an event, constructing and sending a PDU, starting or stopping a timer to handle failure or non-response from the remote program, opening and closing the transport (TCP) connection, and some other actions. We will develop the code for each here. Since we have not yet explained how to write the code that parses or generates PDUs or PDVs, we will encapsulate the problem by assuming we can simply call functions to perform those operations.

Both server and client code will start out in `state-01`. A server will at this point be waiting for an incoming TCP connection on port 104 (or another port if the configuration specifies an alternate). When a connection is made, i.e., the call to `accept-connection` returns, this is `event-05`, action function `ae-05` is called

and the next state will be **state-02**. In **state-02** the server should post a read, awaiting a PDU, and then depending on what happens with that process, a new event will be posted, a new action function will be called, and a next state transition will take place. In most cases these action functions will be very simple. As we will see, some merely signal an event and others do nothing at all.

We will implement the action functions in the order listed in Part 8. You will notice that the lists in Part 8 (tables 9-6 through 9-9) also specify the next state. This is not modular, though it is accurate. Instead we represent the action functions as making events happen, which is how the next state is determined.

Since we are developing the server at this point, we will focus on the actions appropriate to server operations and those in common between server and client. The first action function pertaining to server operation is **ae-05**, which is invoked to indicate that the server has received an incoming connection. This function simply returns **nil**, and that means the environment will be reset to an empty one. Nothing else need be done at this point except possibly to write a message to the log file. After a connection is established, the next thing to happen is to read a PDU (in the main state machine loop). That will in turn signal an event and the state machine will proceed to the next iteration.

```
(defun ae-05 (env tcp-stream)
  "Issue connection open message"
  (declare (ignore env tcp-stream))
  ;; write a log entry...
  nil) ;; return a fresh, empty environment
```

The next action function, **ae-06**, does a lot more work. It needs to examine the environment to see if an **A-ASSOCIATE-RQ** PDU came in with a request acceptable to the server. If it is, the action is to signal **event-07**, an “accept” response indication, and if not, then signal **event-08**. Now, if you look carefully in Part 8 at the state table and the association establishment action function table, you will see that the designers of DICOM tried to short circuit what should happen, and made the state table appear non-deterministic. What should be in the state table is that in **state-02** with **event-06** the next state is always **state-03**. The event that is signaled will be handled in **state-03** and the next transition, to either **state-06** or **state-13**, will be determined by which event (7 or 8) was signaled by **ae-06**. And this is how we will implement it.

What determines if the proposed association is acceptable? Here there are many design choices. At a minimum, if the request includes a proposed SOP Class UID, identifying the operation and data that are requested (it is actually not required at this point), it is a good idea for the server to check if it is capable of providing that service. Most DICOM implementations do nothing more than this. It would be better to determine if the requestor was known to the server (i.e., have a configuration file with a list of allowed remote clients), and if the remote is using a DICOM protocol version the server supports. One could check against the IP address of the remote, and the AE Title. We will not include this level of detail here, though we will indicate where it might be placed.

why are they in the spec?  
We should speculate about this...

Another job of this function is to examine the proposed Presentation Contexts, to see if any are supported by the server. Since the DICOM default transfer syntax is required to be supported and always available, this should always succeed. The option to offer (and for the server to accept) other transfer syntaxes is available, and a well designed server should examine what was sent, to check this. The Presentation Context items will be in a list under the `:A-ASSOCIATE-RQ` tag in the environment.

```
(defun ae-06 (env tcp-stream)
  "Determine acceptability of A-ASSOCIATE-RQ"
  (cond ((and (remote-id-ok env tcp-stream)
              (SOP-Class-UID-OK env))
        (setq *event* 'event-07)
        (setq *presentation-contexts*
              (check-presentation-contexts env)))
        (t (setq *event* 'event-08)
            (setq *reject-reasons* (reject-reasons env))))
  env) ;; return env unchanged?
```

could we add this info onto the environment since it will be used to generate the outgoing PDU anyway? Note also `*args*` in many other actions.

The values stored in the global variables will be used by the actions in the next iteration of the state machine. We need action functions that will send association accept or reject messages. These are `ae-07` and `ae-08`, respectively. They each will send the appropriate PDU. There is no need to signal an event, since the next state will require a PDU read operation.

```
(defun ae-07 (env tcp-strm)
  "Issue A-Associate-AC message and send associated PDU"
  ;; Client proposed max PDU size - accept it and cache minimum
  ;; of that value and server's own maximum for all remaining
  ;; PDUs during this association.
  (let ((limit (item-lookup 'Max-DataField-Len env nil
                           :Max-DataField-Len-Item
                           :User-Information-Item
                           :A-Associate-RQ)))
    (setq *max-datafield-len*
          (if (typep limit 'fixnum) (min limit #.PDU-Bufsize)
              #.PDU-Bufsize))
    (apply #'send-pdu :A-Associate-AC env tcp-strm *args*)
    (setq *args* nil)))

(defun ae-08 (env tcp-strm)
  "Issue A-Associate-RJ message and send associated PDU"
  (apply #'send-pdu :A-Associate-RJ env tcp-strm *args*)
  (setq *args* nil))
```

We defer developing the actions for sending and receiving data for now.



The Association Release actions are all straightforward, and require either sending an appropriate PDU (constructed from the current environment), or signaling an event, or closing the TCP connection. The code should match closely with the description in DICOM Part 8, Table 9-8. Note that the “next state” information in this table is redundant with the state transition table, Table 9-10.

```
(defun ar-01 (env tcp-strm)
  "Send A-Release-RQ PDU"
  (send-pdu :A-Release-RQ env tcp-strm)
  nil)

(defun ar-02 (env tcp-strm)
  "Issue A-Release message"
  (declare (ignore env tcp-buffer tcp-strm))
  (setq *event* 'event-14)
  nil)

(defun ar-03 (env tcp-strm)
  "Issue A-Release confirmation message, close connection"
  (declare (ignore env))
  (close-connection tcp-strm)
  nil)

(defun ar-04 (env tcp-strm)
  "Send A-Release-RSP PDU"
  (send-pdu :A-Release-RSP env tcp-strm)
  nil)

(defun ar-06 (env tcp-strm)
  "Issue P-Data message -- P-Data PDU arriving out of order"
  ;; This action is to handle P-Data-TF PDUs that arrive out
  ;; of order, when the client has initiated a release but
  ;; the server has not processed the PDU and is still sending
  ;; Data-Set PDUs. Client must handle data and continue
  ;; waiting for the A-Release-RSP PDU [loop to STATE-07].
  (declare (ignore env tcp-strm))
  nil)

(defun ar-08 (env tcp-strm)
  "Issue A-Release message [release collision]"
  (declare (ignore env tcp-strm))
  (when (eq *mode* :Client)
    (setq *event* 'event-14))
  nil)

(defun ar-09 (env tcp-strm)
```

```

"Send A-Release-RSP PDU"
(send-pdu :A-Release-RSP env tcp-strm)
nil)

(defun ar-10 (env tcp-strm)
  "Issue A-Release confirmation message"
  (declare (ignore env tcp-strm))
  nil)

```

Finally, there are conditions that occur which require termination of communication, and these are handled by the *ABORT* actions. These conditions include:

- receipt of an unexpected PDU, one that is inappropriate for the current state,
- expiration of a timer, indicating that the remote system has become inaccessible or some other remote or network problem,
- the TCP connection is closed unexpectedly by the remote system,

The abort actions are also simple - they either attempt to send an **A-Abort** PDU or simply close the TCP connection on the local end of the communication, or simply ignore the unexpected PDU.

```

(defun aa-01 (env tcp-strm)
  "Error detected -- send A-Abort PDU (Service-User-Initiated)"
  (cond ((consp *args*)
    (apply #'send-pdu :A-Abort env tcp-strm *args*)
    (setq *args* nil))
    ;; Abort-Source = 0: UL Service-User-initiated
    ;; Abort-Diagnostic = 2: Unexpected PDU
    (t (send-pdu :A-Abort env tcp-strm
      'Abort-Source 0 'Abort-Diagnostic 2)
      nil)))

(defun aa-02 (env tcp-strm)
  "ARTIM timeout"
  (declare (ignore env))
  (when (eq *mode* :Client) (close-connection tcp-strm))
  nil)

(defun aa-03 (env tcp-strm)
  "Issue A-Abort/A-P-Abort message, close connection"
  (declare (ignore env))
  (when (eq *mode* :Client) (close-connection tcp-strm))
  nil)

(defun aa-04 (env tcp-strm)

```

```

"Issue A-P-Abort message, close connection"
(declare (ignore env))
(when (and (eq *mode* :Client)
           (streamp tcp-strm))
  (close-connection tcp-strm))
nil)

(defun aa-06 (env tcp-strm)
  "Ignore invalid/unhandleable PDU"
  (declare (ignore env tcp-strm))
  nil)

(defun aa-07A (env tcp-strm)

  "Unexpected PDU received -- Send A-Abort PDU"
  ;; Abort-Source = 2: UL Service-Provider-initiated
  ;; Abort-Diagnostic = 2: Unexpected PDU
  (send-pdu :A-Abort env tcp-strm
            'Abort-Source 2 'Abort-Diagnostic 2)
  nil)

(defun aa-07B (env tcp-strm)
  "Unrecognized PDU received -- Send A-Abort PDU"
  ;; Abort-Source = 2: UL Service-Provider-initiated
  ;; Abort-Diagnostic = 1: Unrecognized PDU
  (send-pdu :A-Abort env tcp-strm
            'Abort-Source 2 'Abort-Diagnostic 1)
  nil)

(defun aa-08 (env tcp-strm)
  "Send A-Abort PDU and issue A-P-Abort message"
  (cond ((consp *args*)
        (apply #'send-pdu :A-Abort env tcp-strm *args*)
        (setq *args* nil))
        ;; Abort-Source = 2: UL Service-Provider-initiated
        ;; Abort-Diagnostic = 2: Unexpected PDU
        (t (send-pdu :A-Abort env tcp-strm
                    'Abort-Source 2 'Abort-Diagnostic 2)
           nil)))

```

\*\*\* the following is draft stuff about the client side action functions.

Function `ae-01` is defined as "Issue transport connect request to local transport service" (this refers to TCP, although in the future other protocols such as OSI could be supported). So, this function (actually used only by a DICOM client) opens an active connection to a specified remote server.

What else to say about actions? Need sketch of `dt-01`, etc.

```
(defun ae-01 (env tcp-strm)
  (declare (ignore tcp-strm))
  (open-connection (item-lookup 'Remote-Hostname env t)
                  (item-lookup 'Remote-Port env t))
  (setq *event* 'event-02)
  env)
```

The remote host name and port should already be in the environment. This is one of the differences between client and server. We will have to revise both the `tcp-client` code and the `dicom-state-machine` code to allow for this. The `open-connection` function will perform the call to `make-socket` that we originally put in the outermost layer.

Function `ae-02` is defined as “Send A-ASSOCIATE-RQ PDU”. It is also a client-only function. So, it is simply:

```
(defun ae-02 (env tcp-buffer tcp-strm)
  (declare (type (simple-array (unsigned-byte 8)
                               (#.TCP-Bufsize)) tcp-buffer))
  (send-pdu :A-Associate-RQ env tcp-buffer tcp-strm)
  env)
```

Note that no event is signaled here, so `env` is returned unchanged.

Function `ae-03` is defined as “Issue A-ASSOCIATE confirmation (accept)”, and is also a client-only action function. This function needs to retrieve and cache some information from the environment, and signal the next event, event 9.

```
(defun ae-03 (env tcp-strm)
  ;; Server either accepted client's proposed max PDU size
  ;; or proposed its own.
  ;; Set variable to cache it for all remaining PDUs
  ;; during this association.
  (let ((limit (item-lookup 'Max-DataField-Len env
                           :Max-DataField-Len-Item
                           :User-Information-Item
                           :A-Associate-AC)))
    (cond ((typep limit 'fixnum)
           (setq *max-datafield-len* (min limit #.PDU-Bufsize)))
          (t (setq *max-datafield-len* #.PDU-Bufsize))))
  (setq *event* 'event-09)
  env)
```

Function `ae-04` is “Issue A-ASSOCIATE confirmation (reject) and close transport connection. This is also a client-only function.

```
(defun ae-04 (env tcp-strm)
  ;; write error message also...
  (close-connection tcp-strm)
  nil)
```

\*\*\* to be continued

### 13.4.3 Parsing and Generation of PDUs

Now we turn to reading, parsing, generating and writing PDUs. We treat each PDU as a sentence in a DICOM language. The DICOM specification may be viewed as defining the grammar and vocabulary for these sentences. A particular DICOM message then can be treated as a sequence of tokens whose content and ordering conforms to this grammar. The vocabulary is defined by the data dictionary. The structure or sequencing of the tokens is expressed in rules, expressions that specify what sequences of tokens constitute valid messages.

Since a PDU can contain a variable number of items and there is no way to determine how many by simply parsing tokens sequentially, every PDU has a length field, that tells the reader how many bytes to read (and subsequently process). The length field is a 32 bit binary quantity encoded in bytes 2–5 (counting the first byte as byte 0). So first, we read the first 6 bytes, then using the length information, read the remaining bytes, all into a buffer that will be passed to a parser. The `decode-fixnum` function defined in Chapter 2 can be used to convert the four byte sequence into an integer data type.

Several things might happen at this point, including error conditions like “connection closed or reset” at the remote end, or an end-of-file condition, or the number of bytes read does not match with the number of bytes expected. In those cases, the data are ignored and the appropriate event signaled. If all is well and the read completes as requested, the `parse-pdu` function is called to translate the contents of the PDU into entries in the environment list. Then depending on what type of PDU it was, the appropriate event is signaled. Here is all that in the `read-pdu` function.

```
(defun read-pdu (tcp-stream env)
  (let ((tail 6) ;; read only 6 bytes to start
        (pdu-end 0) ;; will be set when PDU length is known
        (connection-reset nil)
        (timeout nil) ;; not implemented - add later
        (tcp-buffer (make-array #.TCP-Bufsize
                                :element-type '(unsigned-byte 8)
                                :initial-element #.(char-code #\*))))
    (unless
      (ignore-errors
        (cond ((= tail ;; read the required number of bytes (6)
              (read-sequence tcp-buffer tcp-strm
                            :start 0 :end tail))
              (setq pdu-end ;; encoded PDU length + 6 bytes
                    (+ (decode-fixnum tcp-buffer 2 4)
                      tail))
```

```

        (setq tail (read-sequence tcp-buffer tcp-strm
                                :start tail :end pdu-end))

        t)
      (t (setq eof? t))))
    (setq connection-reset t))
  (setq *event*
    (cond
      (connection-reset 'event-17)
      (timeout 'event-18)
      (eof? 'event-17)
      ((< tail pdu-end) 'event-17)
      (t (let* ((pducode (aref tcp-buffer head))
                 (pdutype (second (assoc pducode
                                         *pdutype-alist*
                                         :test #'=))))
            (if pdutype ;; try to parse the message
              (multiple-value-bind (next-byte new-env)
                (rule-based-parser
                 (get pdutype :Parser-Rule)
                 env tcp-buffer (+ head 6) pdu-end)
                (cond
                  ((eql new-env :Fail) 'event-19)
                  ((= next-byte pdu-end)
                   (setq env new-env)
                   (case pdutype
                     (:A-Associate-AC 'event-03)
                     (:A-Associate-RJ 'event-04)
                     (:A-Associate-RQ 'event-06)
                     (:P-Data-TF 'event-10)
                     (:A-Release-RQ (if (eq *mode* :Client)
                                         'event-12A
                                         'event-12B))
                     (:A-Release-RSP 'event-13)
                     (:A-Abort 'event-16)))
                   (t 'event-15) ;; Inconsistent length PDU
                 ))
              'event-19) ;; bad PDU code
            ))))
    env))

```

After the read is done we test for errors, connection closed, or end of file (EOF). If an error happens in either of the two `read-sequence` calls, it most likely is a *socket-reset* error. This indicates “Stream-Closed by Remote Host,” which the DICOM protocol handles the same as EOF. One matter we do not address here is that the read should be done with provision for timeout, which is referred to in the DICOM specification as the ARTIM timer.

### Parsing of PDUs

We implement parsing of incoming PDU data and generation of outgoing PDU data via rules which define the elements that must be matched in an incoming stream or placed in an outgoing stream. The rules contain variable references, allowing the substitution of data values for named variables during the parsing and generation process.

\*\*\* the following paragraph is not exactly right – redo it.

The rules define an *augmented context-free grammar*. It is *context-free* in that the legality of a structure depends only on its form and not on the context in which it appears, and *augmented* in that rules can contain embedded function calls to test decoded data values for appropriateness. Context-free grammars are a formal way of defining the syntax of a language, including programming languages themselves [138].

\*\*\* NO, we are changing this, redo this paragraph.

The same kind of rules will also describe commands contained in a P-DATA-TF PDU, so we will implement the parser in a function, **parse-group**, that will handle both. However, the context in which the parse takes place is slightly different. For a PDU, the first byte already tells us the type of message, so we can fetch the rule for it and apply it. For a command, the particular type of command is embedded in the byte stream, so we have to try each command rule in turn, until one works or the parse fails (in the case that the message is erroneous). The **parse-pdu** function checks the first byte, skips the length bytes and unused bytes, and passes the remaining data and the appropriate rule to **parse-group**.

\*\*\* here list the kinds of items in PDU's other than PDV items.

For convenience, let's assume each rule is a list whose first term is a keyword symbol naming that rule and whose second term is the numeric code for it in the DICOM standard. The rest will be the details of that PDU's structure.

```
(defun rule-based-parser (rule env buffer head tail)
  (let ((init-head head)
        (init-env env))
    (dolist (term (cdr rule))
      (progn
        (unless (eq env init-env)
          (do ((item env (cdr item))
              (next (cdr env) (cdr next)))
              ((eq next init-env)
               (setf (cdr item) nil)
               (setq env
                     (cons (car rule) (nreverse env)))
               (setq env
```

```

                (if (equal env (first init-env))
                    init-env
                    (cons env init-env))))))
            (values head env)))
    (multiple-value-bind (next-byte new-env)
      (parse-term term env buffer head tail)
      (if (eq new-env :Fail)
          (return (values init-head :Fail))
          (setq head next-byte env new-env))))))

```

To write `parse-term` we need to decide on how the rest of the rule will look. The elements of these grammar rules are *tokens* (symbols naming constants or other rules) or *expressions* (an expression is a parenthesized list of tokens). Expressions can represent *variables* (data values that can be referred to by name in other rules), *function calls* (data values computed from simpler expressions), or *operators* (instructions to generate particular values, such as computing the length of the PDU currently being constructed).

For parsing, a rule describes *conditions* for matching the incoming byte-stream and the *variable bindings* to be added to the environment as a result of such matches. The rule shown in Figure 13.5 would be used to parse an **A-Associate-RQ** message, which has a structure similar to that shown in Figure 13.2. It illustrates several element types, including tags, constants, ignored fields, variables, recursive rule invocations, and optional items.

```

(:A-Associate-RQ
  #x01          ; A-Associate-RQ PDU Type tag
  =ignored-byte ; Reserved field [1 byte]
  (=ignored-bytes 4) ; PDU Length [4 bytes]
  (>decode-var Protocol-Version fixnum 2 :Big-Endian)
  (=ignored-bytes 2) ; Reserved field -- not used
  (>decode-var Called-AE-Title string 16 :Space-Pad)
  (>decode-var Calling-AE-Title string 16 :Space-Pad)
  (=ignored-bytes 32) ; Reserved field -- not used
  :Application-Context-Item
  (:Repeat (1 :No-Limit) :Presentation-Context-Item-RQ)
  :User-Information-Item)

```

Figure 13.5: Parse rule for A-Associate-RQ PDU

The symbol `:A-Associate-RQ` is a *tag* and is returned as the result of decoding this PDU. It also serves to name the rule, allowing rules to be looked up by name in the list of rules.

The hexadecimal value `#x01` is a constant that must be matched by a single byte in the input stream. This value identifies the PDU as an **A-Associate-RQ** PDU.

The entries `=ignore-byte` for a single byte and `(=ignored-bytes 32)` for a



field of 32 bytes (and similar entries) indicate that the parser should account for byte field lengths but should otherwise ignore the content of those fields.

(>decode-var Called-AE-Title string 16 :Space-Pad) indicates that the data in the next 16 bytes should be interpreted as a character string, with space-padding stripped, and stored into a *variable* named **Called-AE-Title**.

:Application-Context-Item means that a rule by that name should be used *recursively* to parse the next sequence of bytes (that is, rules can use other rules recursively to define their own components).

(:Repeat (1 :No-Limit) :Presentation-Context-Item) refers to an item that may appear one or more times in the input data stream. *Which* item is indicated by its tag, and the minimum and maximum number of allowable repeats (including a minimum of zero for *optional* items) is indicated by the limits (1 :No-Limit) (one to infinity in this case).

Since the rule interpreter for PDU parsing works recursively, components can contain other components (nested arbitrarily deeply). The rule for the outer component need simply refer to the name of the sub-rule (the keyword symbol used to look up the sub-rule in the ruleset stored as an association list). Variables in the environment can pass information between sub-rules as well as storing information obtained during parsing of an incoming transmission for use in formatting an outgoing reply.

As an example, the rule for an Abstract Syntax Item is shown in Figure 13.6. This item is contained within a Presentation Context Item which itself is contained within an A-Associate-RQ PDU.

```
(:Abstract-Syntax-Item-RQ
  #x30          ; Abstract Syntax Item type tag
  =ignored-byte ; Reserved field [1 byte]
  ;; Abstract Syntax Name field length [2 bytes]
  (>decode-var ASN-Len fixnum 2 :Big-Endian)
  ;; Abstract Syntax Name string - variable-length
  (>decode-var ASN-Str
    string
    (<lookup-var ASN-Len)
    :No-Pad))
```

Figure 13.6: Parse rule for Abstract Syntax Items

Parsing of the data using this sub-rule works exactly as described above. The additional concept illustrated here is that function calls can be embedded within rules. In the last expression, the (<lookup-var ASN-Len) term tells the parser that the length of the string to be decoded is obtained by calling the function <lookup-var with the variable **ASN-Len** as an input. The function <lookup-var is simply an environment accessor. The reason for this mechanism is that the rule cannot specify a constant length (as does the **Called-AE-Title** slot in the **A-Associate-RQ** PDU rule) because that length is unknown at the time the rule

was written – it is determined at run-time from the message. Essentially, data parsed from one part of the message is used in decoding another part of the same message.

Now we can write the **parse-term** function. The last byte of the PDU in the buffer is known, but the parser may be called recursively so the first byte to process may vary. We call one **head** and the other **tail**. If a parse succeeds, the next byte to read is returned along with the new environment, but if it fails, the parse returns the unchanged start point and the keyword **:fail**.

```
*** parse-term code here
```

New kinds of messages can be supported by just adding rules and state table entries rather than adding to the code. Similarly, modifications to the DUL protocol can be accommodated by making the corresponding alterations in the rules and state table.

### Generation of PDUs

We will use rules in the same format as the parse rules to describe the generation and structure of outgoing messages. Generation rules can contain references to the same variables as do the parse rules, providing a simple mechanism to echo received values on transmission (or to output values computed from previously received data).

Once we have a rule-based PDU parser and a table-driven data-object parser, it is easy to add generation of PDUs (including the transmission of embedded data objects). We simply run the parsers “backwards” as instantiators. We create generation rules rather than using the same rules for both kinds of tasks because there are a few semantic differences in generating messages vs. parsing, which were most easily accommodated with slight variations in the rules.

Figure 13.7 shows the generation rule for the **A-Associate-RQ** PDU. It is a reflection of the corresponding rule for the parser, but instead of creating variables and values to add to the environment, the PDU is constructed from variables and values in the environment. This rule tells the generator to output a byte stream consisting of:

- one byte - the constant value 01 (hex), indicating the PDU type,
- one byte - the constant value 00 (hex),
- four bytes reserved as a place-holder where the length of the entire PDU will be inserted (we don’t know that length until we finish generating the entire PDU, but space must be reserved at the beginning for it),
- a fixnum constant of length 2 bytes encoding the value 0001 (hex),
- a string of length 2 of repeated bytes, each with value 00 (hex),
- the value of the variable “Called-AE-Title” (retrieved from the environment as described above), encoded as a string with padding by Space characters out to 16 bytes,

- the value of the variable “Calling-AE-Title,” likewise encoded as a 16-byte Space-padded string,
- a string of length 32 of repeated bytes, each with value 00 (hex),
- an Application Context Item (here the generator calls itself recursively using the rule indexed by :Application-Context-Item and uses the environment to supply any needed variable bindings),
- a Presentation Context Item, similarly generated by a recursive call,
- finally, a User Information Item, similarly recursively generated.

The “-RQ” in the keyword symbol names in Figure 13.7 refers to rules for items specialized slightly for use inside the **A-Associate-RQ** PDU. The rules might vary slightly for the same item used in other contexts, as in the **A-Associate-AC** PDU.

```
(:A-Associate-RQ
  #x01          ; A-Associate-RQ PDU Type tag
  #x00          ; Reserved field [1 byte]
  :Place-Holder ; PDU Length [4 bytes]
  ;; Protocol Version - fixed value
  (=fixnum-bytes #x0001 2 :Big-Endian)
  (=constant-bytes #x00 2) ; Reserved field
  (<encode-var Called-AE-Title string 16 :Space-Pad)
  (<encode-var Calling-AE-Title string 16 :Space-Pad)
  (=constant-bytes #x00 32) ; Reserved field
  :Application-Context-Item
  :Presentation-Context-Item-RQ ; we only provide one
  :User-Information-Item-RQ)
```

Figure 13.7: Generate rule for **A-Associate-RQ** PDU

Since the rule interpreter for PDU generation works recursively, just as for parsing, generated components can contain other components (nested arbitrarily deeply). The rule for the outer component simply refers to the name of the sub-rule (the keyword symbol used to look up the sub-rule in the ruleset stored as an association list). As for parsing, variables in the environment can pass information between sub-rules, and lookup functions can be used to obtain information for use in formatting an outgoing message.

\*\*\* generator code here

#### 13.4.4 Parsing and Generation of Commands

The parser for PDUs containing commands and data uses the mechanism described above, including a language defined by the parser rules just described and imple-

mented by a recursive-descent parser using these rules. One component of the P-Data-TF PDU is the “message,” that is, the combination of commands and/or data objects that are being communicated between DICOM entities. The PDU parser treats the message as a single entity which is stored in the environment as the value of the variable `PDV-Message`. To the PDU parser, this message is simply an uninterpreted stream of bytes. The DICOM Upper-Layer protocol specifies a header for each message encoding the length of the following data field and some flags indicating whether the data field is a command message or a data object (and additionally whether the data field contains the complete object or is a portion of a long data field that has been fragmented into multiple messages).

If the message is a command, our system gives the PDU parser pointers to the beginning and end (in the TCP data buffer) of the message. The ruleset contains rules for parsing commands using the same conventions and language as for parsing any other type of PDU. So, for commands we have rules with similar structure, and the `parse-command` function will call `parse-group` just as does `parse-pdu`. Figure 13.8 shows a command rule.

```
(:C-Store-RQ
  (=fixnum-bytes #x0000 2 :Little-Endian) ;; Tag (0000,0000)
  (=fixnum-bytes #x0000 2 :Little-Endian)
  (=fixnum-bytes 4 4 :Little-Endian)      ;; Length
  (>decode-var Group-Len fixnum 4 :Little-Endian) ;; Value
  (=fixnum-bytes #x0000 2 :Little-Endian) ;; Tag (0000,0002)
  (=fixnum-bytes #x0002 2 :Little-Endian)
  (>decode-var Store-SOP-Class-UID-Len fixnum 4 :Little-Endian)
  (>decode-var Store-SOP-Class-UID-Str      ;; Value
    string
    (<lookup-var Store-SOP-Class-UID-Len)
    :Null-Pad)
  (=fixnum-bytes #x0000 2 :Little-Endian) ;; Tag (0000,0100)
  (=fixnum-bytes #x0100 2 :Little-Endian)
  (=fixnum-bytes 2 4 :Little-Endian) ;; Length
  (=fixnum-bytes #x0001 2 :Little-Endian) ;; Value
  ... ;; other items in here elided
  (=fixnum-bytes #x0000 2 :Little-Endian) ;; Tag (0000,1000)
  (=fixnum-bytes #x1000 2 :Little-Endian)
  (>decode-var Store-SOP-Instance-UID-Len fixnum 4 :Little-Endian)
  (>decode-var Store-SOP-Instance-UID-Str
    string
    (<lookup-var Store-SOP-Instance-UID-Len)
    :Null-Pad)
  (:Repeat (0 1) :Move-Originator-AE)
  (:Repeat (0 1) :Move-Originator-ID))
```

Figure 13.8: A rule describing the structure of the C-STORE-RQ command

The code for `parse-command` has to be different than for `parse-pdu`. First, in

`parse-pdu` we know which rule to use, because the first byte identifies the PDU type. The commands are not encoded this way. Where the command identifier appears depends on the command so we will have to try each of the supported commands until one matches or we exhaust the supported commands list. This is simple iteration, calling `parse-group` with the same data and each command rule.

check on this, if really true, it is really another dumb move by the DICOM designers

```
(defun parse-command (env tcp-buffer head tail)
  (dolist (msgtype *Message-Type-List* (values :Fail nil))
    (multiple-value-bind (next-byte new-env)
      (parse-group (get msgtype :Parser-Rule)
                   env tcp-buffer head tail)
      (declare (ignore next-byte))
      (unless (eq new-env :Fail)
        (return (values msgtype new-env))))))
```

If a command matches the message data, the function returns the command type and the new environment. If none match the function returns the keyword `:fail` and the second value is `nil`.

\*\*\* need to develop generation code for commands

### 13.4.5 Parsing and Generation of Data

If the message contained in a P-DATA-TF PDU is a data object, the buffer (and begin/end pointers) are passed to a different parser. This parser is also data-driven, using the data dictionary to decode arbitrary data objects.

\*\*\* what is needed about data elements? Tag, type (implies format). In data stream, tag comes first, then data

\*\*\* example

\*\*\* Note - how was the data dictionary first created? maintained for new releases?

The data dictionary is a text file, containing Lisp expressions. It can be read by a human reader and can be written and extended by using a text editor. But it is also readable by the program, and functions as executable code, though in a new declarative language, not Lisp itself. Each entry in our version of the data dictionary is a list containing the tag, value representation, and name. This list maps data-object tags (both fields, group and element numbers) to symbols naming the object's data-type and a string describing the object. An excerpt from this table is shown in Figure 13.9. For example,

```
((#x0020 . #x0032) DS "Image Position Patient")
```

```

(defparameter *group/elemname-alist*
  ;;-----
  ;; Group 0000: CMD "Command"
  '(((#x0000 . #x0000) UL "Group Length")
    ((#x0000 . #x0002) UI "Affected SOP Class UID")
    ((#x0000 . #x0003) UI "Requested SOP Class UID")
    ((#x0000 . #x0100) US "Command Field")
    ((#x0000 . #x0110) US "Message ID")
    ...
  ;;-----
  ;; Group 0010: PAT "Patient Information"
  ((#x0010 . #x0000) UL "Group Length")
  ((#x0010 . #x0010) PN "Patient Name")
  ((#x0010 . #x0020) LO "Patient ID")
  ((#x0010 . #x0021) LO "Issuer of Patient ID")
  ((#x0010 . #x0030) DA "Patient's Birthdate")
  ...
  ;;-----
  ;; Group 0028: IMG "Image"
  ((#x0028 . #x0000) UL "Group Length")
  ((#x0028 . #x0002) US "Samples Per Pixel")
  ((#x0028 . #x0005) RET "Image Dimensions (RET)")
  ((#x0028 . #x0006) US "Planar Configuration")
  ((#x0028 . #x0010) US "Rows")
  ((#x0028 . #x0011) US "Columns")
  ...
  ))

```

Figure 13.9: An excerpt from the DICOM data dictionary table

is the entry for the item mentioned in section 13.3, **Image Position (Patient)**.

Value representations, such as **DS**, are defined in another table, also an association list. An excerpt from this table is shown in Figure 13.10.

This association list maps data types (identified by the first symbol in each sublist) to descriptive information (the string naming it, the second element) and information needed to parse objects of that type (the optional third and fourth elements). If these optional elements are present, they specify the Lisp data type of the object. The types and included information are:

- **STRING** – minimum and maximum length, whether and how the string is padded,
- **FIXNUM** – integer, including number of bytes,
- **DOUBLE-FLOAT** – 8-byte floating-point reals,
- **SINGLE-FLOAT** – 4-byte floating-point reals,

```

(defparameter *datatype-alist*
  '((AE "Application Entity" (string 0 16) :Space-Pad)
    (AS "Age String" (string 4) :No-Pad)
    (AT "Attribute Tag" (fixnum 4))
    (CS "Code String" (string 0 16) :Space-Pad)
    (DA "Date" (string 8) :No-Pad)
    (DS "Decimal String" (string 0 16) :Space-Pad)
    (FD "Floating-Point Double" (double-float 8))
    (FL "Floating-Point Single" (single-float 4))
    (IT "Item in Sequence")
    (ITDL "Item Delimiter")
    (OB "Other Byte" ((unsigned-byte 8) 0 *) :Null-Pad)
    (OW "Other Word" ((unsigned-byte 16) 0 *) :No-Pad)
    (SL "Signed Long" (fixnum 4))
    (SQ "Sequence of Items")
    ...
  ))

```

Figure 13.10: An excerpt from the value representation table, part of the DICOM data dictionary

- (UNSIGNED-BYTE 8) – Uninterpreted string of bytes used to encode 8-bit data values,
- (UNSIGNED-BYTE 16) – Uninterpreted string of bytes used to encode 16-bit data values.

Certain data types are used to delineate nested data objects. These include: IT "Item in Sequence", ITDL "Item Delimiter", SQ "Sequence of Items", and SQDL "Sequence Delimiter". In such cases, the containing object is a "Sequence of Items," whose end is marked by a "Sequence Delimiter." Each element composing the sequence is an "Item in Sequence" with its end marked by an "Item Delimiter." Thus an item in a sequence can itself consist of one or more sequences. Data objects can be nested to arbitrary depth in this manner, and our parser uses a simple recursive-descent algorithm to maintain the state of the parse as it accumulates the sub-elements and packages them into the containers.

\*\*\* Example here

The basic object-parsing algorithm is simple. All data objects are represented as a sequence of elements. Each element (whether it is the outermost container for a large data structure or the innermost scalar element of a larger structure) is represented by a three-component byte-field in the data stream. The first field is the Data Element Tag, an ordered pair of 16-bit unsigned integers, the Group and Element numbers. Indexing by this pair of numbers into the Group/Element-name association list gives the symbol naming the data type of the following data value. (It also yields the descriptive string useful for printing debugging messages and for

dumping arbitrary data-objects.) The data type symbol then serves as an index into the data type association list, giving information needed to parse the data object (number byte-field lengths, string lengths and padding, etc).

\*\*\* code here

The second field for each element is “Value Length,” a fixed-length integer giving the byte-string length of the encoding of the value of the particular element. The third field is the “Value Field,” the string of bytes (of length given by “Value Length”) holding the actual data. Our parser simply reads the appropriate number of bytes and interprets them according to the specifications for the given data type (numerical, string with or without padding, etc).

\*\*\* code here

For data objects whose value field contains a sequence, an alternative specification of “Value Length” is a special code for “Undefined Length.” In that case, the Value Field for that element contains a sequence of items, each represented by its own Tag-Length-Value structure. The parser decodes each element in turn, until detecting an element of type “Sequence Delimiter.” That is the indication that the “Undefined Length” field is now coming to an end.

\*\*\* code here

The first element of any group of data objects (ie, whenever the Group Number portion of the Element Tag increments) is an element giving the length of the entire group, encoded as a fixed-length integer. The value decoded from this element can be used to decode the rest of the elements in the group (i.e., it specifies the length in bytes of the entire group). This element is not strictly necessary, but having it simplifies the reconstruction of data objects whose representations have been fragmented into multiple P-DATA-TF PDUs (images are the prime example of this).

## 13.5 Image Server Design

The Prism DICOM Server (PDS) functions as a DICOM Application Entity (AE). PDS currently provides the C-ECHO (verification) and C-STORE operations. A variety of transverse image types are supported. The server is started at boot time and can also be run manually. It normally runs unattended as a detached process.

The application layer software for the server implements a connection acceptance policy via a configurable set of parameters which define IP addresses from which connections will be accepted (plus a “promiscuous” mode which accepts any connection, for testing purposes - this is sometimes necessary to learn the configuration parameters of a client the first time we set up service for it). Included in this connection-acceptance policy is a mechanism for declaring that output go to particular destinations dependent upon the IP address or AE Title of the client. This enables clients to target images to be stored in particular file system directories,



ie, to distinguish between clinical and research image sets, by sending to a target identified by a different AE Title.

### 13.5.1 Image Data Types Supported

The data dictionary includes all the items defined in the DICOM standard, for both simple and composite objects. The server can therefore parse any data that is sent to it in conjunction with a C-STORE message. The result is a collection of structured objects in the server environment, as described above. What distinguishes the server's ability to handle different types of images and sets of images is therefore the kind of data that can be stored in the Prism RTP system files. The Prism file system can accomodate at present only sets of transverse images that share an axial alignment, in that the images taken together will form a volume image data set. However the server code does not make any requirements. It only checks to see that certain attributes identify the images as transverse images, not overlaid with other data.

Our Conformance Statement [65] details exactly what our system implements (which SOP classes, how it responds to various errors and various other situations, etc). In summary, the SOP service classes we implement are:

- Echo-Verification-Service - 1.2.840.10008.1.1
- Structure-Set-Storage-Service - 1.2.840.10008.5.1.4.1.1.481.3
- RTPPlan-Storage-Service - 1.2.840.10008.5.1.4.1.1.481.5
- CT-Image-Storage-Service - 1.2.840.10008.5.1.4.1.1.2
- MR-Image-Storage-Service - 1.2.840.10008.5.1.4.1.1.4
- US-Image-Storage-Service - 1.2.840.10008.5.1.4.1.1.6.1
- PET-1-Image-Storage-Service - 1.2.840.10008.5.1.4.1.1.128
- PET-2-Image-Storage-Service - 1.2.840.10008.5.1.4.1.1.129
- PET-3-Image-Storage-Service - 1.2.840.10008.5.1.4.1.2.2.1

### 13.5.2 Storage of the Received Data

In order to implement the C-STORE function, the server must accumulate data in the environment until it reaches the end of the C-STORE message (usually a sequence of PDUs). At that point the environment contains one or more images and associated data. All the data written out are in a Prism proprietary file format, for ease of use with the Prism system, rather than in the format of DICOM's media exchange files, described in Part 10 of the DICOM standard. They are collected into a Prism image set object instance and written to the Prism image database using existing Prism file input and output functions. This last step is the only part of the server that has Prism-specific code.

The Prism file system stores data about images, patient anatomy, radiotherapy plans, and other information in a text-based format using keyword-value pairs. Each Prism object is an instance of a class in the Common Lisp Object System. Each data item is a value for a slot in the class. The keywords are the slot names, and the values are either simple data readable by the Common Lisp `read` function, instances of other Prism objects, lists of Prism objects, or large binary arrays such as image arrays. The binary arrays are in separate files from the text descriptions. Thus the issue of parsing “header” files in more conventional data stores is entirely avoided. An excerpt from an image description data file is shown in Figure 2.3. For the pixel data slot, the file contains sufficient information for a binary-array-reading function to read in the relevant binary file and to construct a Common Lisp array data type from it.

In the server, there is a function that selects from the environment the data elements defined by DICOM that correspond to the required elements in the Prism image object definition. The data contained in Prism image objects is a subset of the data provided by a DICOM image source. So, for example, DICOM element (0018,0050) is the source for the Prism image slot `thickness`, and element (7FE0,0010) is one of the sources for the `pixels` slot, along with elements (0028,0010), the number of rows in the image, and (0028,0011), the number of columns. Some slots in the Prism image class are computed from several DICOM elements. An example is the `size` slot, which is computed from the rows, columns and (0028,0030) (pixel spacing) elements. Some Prism slots are generated by the server, not derived from the transmitted data, for example, the Prism `x-orient` slot is always set to the vector `#(1.0 0.0 0.0)`, since only transverse images are stored (and similarly for `y-orient`).

## 13.6 DICOM Client Design

The DICOM protocol has recently been extended to incorporate data used to specify radiation therapy treatments as well, in view of the fact that modern radiation therapy machines are computer-controlled devices. This extension, called DICOM-RT, facilitates the process of planning, delivering, and recording radiation treatments. It is expected to reduce or even eliminate transcription errors in transferring information from a radiation therapy planning computer to a computer controlled treatment machine. New radiation treatment techniques such as Intensity Modulated Radiation Therapy (IMRT) [139] are impossible to set up by manual data entry, so for these cases, a data transfer facility like DICOM-RT is essential to the operation. The DICOM-RT additions were first described in Supplement 11 of the standard, then later incorporated as additions to the appropriate sections of the main standard documents.

Because the server design puts details into rules and a state table, rather than hard-coding them, it can easily be adapted to add other functions and support for additional objects such as radiotherapy plans. This same approach was taken in the design of the DICOM-RT client code, to send radiotherapy treatment plan data to the treatment machines. The same common code used in the image server is reused

in the DICOM-RT client, since it is not server-specific. The server-specific code is not needed for this client application.

The DICOM-RT client follows the standard design of a connection-oriented, single-threaded TCP/IP client [23]. The Prism program includes a control panel that provides a user interface to the DICOM-RT client subsystem. The user picks the plan data that is to be sent, and the data are passed to the DICOM core code to be packaged into a message according to the DICOM Upper Layer Protocol. The client requests a connection to the radiotherapy machine computer (a server program is listening there for incoming C-STORE requests), then it transfers the data and returns control to the Prism application code when done.

The input driver for the client (called Prism DICOM-RT, or PDR) formats complete treatment plans as generated in Prism into a data object and transmits that data to a DICOM server on our linear accelerators. In implementing DICOM-RT, transmitting data according to the protocol was straightforward.

The more difficult issues arise in the area of system integration, because the same prescription information must be represented in three different ways: in Prism, in DICOM-RT, and in the accelerator. Some of the differences are relatively simple ones involving, for example, the numbering of the collimator leaves or the coordinate system for collimator leaf motion. Others involve more complex organizational matters. For example, in Prism a prescription is simply a set of beams, where the description of each beam is complete and self-contained. In DICOM-RT a prescription is a set of partial beam descriptions and also a set of “fraction groups,” where each fraction group contains some (not necessarily all) of the beams and conveys the number of treatment days and the daily dose for each beam in that group (an index number associates each beam in a fraction group with corresponding beam data elsewhere). In the accelerator a prescription is organized in a hierarchical (tree-like) structure including treatment sites, treatment phases, and fraction groups. The DICOM-RT representation is determined by the DICOM standard, the transformation from the DICOM-RT representation to the accelerator representation is determined by the accelerator software, and the transformation from Prism to DICOM-RT is determined by us. We had to acquire a thorough understanding of all three representations and both transformations in order to ensure that every prescription is transmitted correctly through the entire communication path, is re-constituted at the accelerator end in a form that is intelligible to us, and can be readily verified against the original prescription in Prism.

## 13.7 Other

### 13.7.1 What We Discovered About DICOM Itself

We found no inconsistencies in the DICOM specification, and although it is often obscure and arcane, the prose description and tables did in every case provide enough information to decide what we needed and what to do with it. The specification is encyclopedic. It does not provide any guidance about which of the many related data elements is the best to use for any given application. It is very important to

be familiar with the context, i.e., what do radiologists, radiation oncologists and other related medical staff do with the data. Seemingly equivalent data elements may not at all behave the same or represent the same semantics.

The DICOM specification did not, until recently, address security or encryption, although it does include details about methods of image compression. Even with the recent adoption of Part 15, many questions are left up to the implementation. A question we had to address early on was: what is the appropriate response to clients that request connections from unknown IP addresses? How much tolerance should an application provide for matching of Application Entity titles? We found variations on both of these. We decided to make IP address checking configurable, so that our server could be set to accept TCP/IP connections only from known source IP addresses, and initially we required AE titles to match exactly. We log connection requests that are rejected so that we can document possible break-in attempts or other problems. We logged no incidents of break-in attempts, but we did get some transmissions of images for unknown patients from known sources. They were stored in the temporary repository as described above, and later discarded by hand, after verifying that the data were simply sent to the wrong destination. We also logged some port scans done by University of Washington network security staff.

One vendor has implemented their DICOM-RT storage server so that it is even more restrictive. It requires that different IP addresses must have distinct AE titles. This makes it difficult to install a DICOM-RT client in a clustered computer environment and just use any computer in the cluster to send data. We are looking at schemes by which our DICOM-RT client can generate a unique AE title based on the IP address on which it is currently running, in order to solve this problem. We don't believe this restriction adds security, but the vendor has committed to it.

It was unfortunate that the ACR-NEMA committee did not separate the data buffering problem from the definition of messages (PDUs are sometimes complete messages and sometimes message fragments). This is the most significant design flaw in DICOM. It would clean up the design considerably for the standard to remove this. However, it is most likely that existing implementations will undergo a major rewrite to accommodate such a change. There is no obvious way to make this backward compatible. Our design goes to some length to minimize the impact of the buffering scheme, and it would not be difficult to strip it out. The original 50 pin wiring scheme was motivated by a desire for higher bandwidth than 10 Megabit ethernet could provide, but in practice this judgement call turned out to be wrong. Presumably if the ACR-NEMA committee had skipped version 1 entirely and relied on already well-established world-wide network protocols and hardware, the buffering scheme would not have been included.

### 13.7.2 Consistency With Other Implementations

In addition to verification by inspection, we tested our implementation with others, sending and receiving data both locally and long distance (some test nodes were located as far away from Seattle as England and Israel). We discovered errors in our own implementation and also errors in some of the other implementations.

Moreover we found that many implementations are tolerant of violations of the

DICOM specification. The CTN code was never intended to be a validation test suite. Even implementations intended to function as validation tests, such as the AGFA test suite mentioned earlier, cannot be relied on for this purpose. We found errors in our code that were tolerated by some commercial implementations. We also found at least one commercial implementation with an encoding error that has apparently been tolerated by many other implementations for many years, including the AGFA software.



## Chapter 14

# Cancer Radiotherapy Planning

Cancer remains the second highest cause of death in the U.S. and other developed countries. While much is known about cancer etiology and prevention, it is still very important to improve our understanding of cancer prognosis and therapy. No great breakthroughs have happened in cancer treatment, but many very significant improvements have been discovered and developed. In particular, the use of ionizing radiation to treat cancer has made great technical strides in the last few decades. Some of this may be attributed to the development of computational modeling techniques that draw on the foundations of biomedical informatics.

Cancer is in reality hundreds of diseases, all sharing the common property that the cells which have become tumor cells do not follow normal growth and death cycles. Tumors grow in an uncontrolled fashion compared to normal tissue. The great challenge of therapy is that tumor cells come from a person's own tissues; they are not foreign bodies like bacteria or parasites. The tumors that arise from different types of organs and tissues have different responses to drugs and radiation, and different growth patterns. The anatomic location, size and shape will strongly affect the prospect of surgical removal. It is not possible here to present an in depth discussion of cancer biology. A very readable textbook developed for an undergraduate college course [85] will provide much more background on this subject.

Cancer has a strong genetic basis, and this is certainly an area where the application of modern molecular genetics and epidemiology has yielded important insights. This application of informatics in cancer won't be covered here, but further reading includes ...

need cites here

The three main modes of cancer treatment are surgery, chemotherapy and radiation. Surgery has benefited from the availability of digital medical imaging technology, particularly the computer assisted X-ray tomographic imaging (CT) scanner and more recently the magnetic resonance imaging (MRI) scanner. Chemotherapy for cancer is managed typically by following protocols, many of which are experi-

explain each with examples and references

mental. These protocols are series of steps and decision points, which are developed by expert oncologists. One of the most important applications of informatics to this process has been the development of knowledge bases and protocol management tools, notably the ONCOCIN project [121]. Radiation therapy is the most technical of these modes of treatment, and has involved computer modeling for many decades. This chapter develops the medical informatics aspects of planning radiotherapy, and illustrates how the foundational ideas of knowledge representation, reasoning, information access and decision making can be applied to facilitate the planning of more accurate and thus more effective radiation treatments.

## 14.1 Radiation Therapy

Radiation therapy directed at malignant tumors involves aiming and collimating a radiation beam at the tumor in such a way as to deposit a large amount of energy in the tumor and as little energy as possible to surrounding (healthy) tissue. Figure 14.1 shows a typical radiation therapy machine. Radiation, such as X-rays (high energy photons), electrons, neutrons and other particle beams, can kill tumor cells by causing ionization of atoms in or around the cell, and in turn, this can result in molecular bond breakage, i.e., damage to the DNA of the cell, or it can produce free radicals, active chemical species, which then attack the DNA. In either case, the goal of radiation therapy is mainly one of solving an energy delivery problem, to direct a required amount of radiation into the tumor as uniformly as possible consistent with avoiding overdose of the healthy surrounding tissue. An excellent introduction to the biological effects of ionizing radiation, accessible to the non-medical reader, despite its title, is Eric Hall's book, *Radiobiology for the Radiologist* [43]. A more technical treatise on the physics aspects can be found in the popular textbook by Khan [77].

A modern radiation therapy machine consists of a high energy linear accelerator, producing X-ray (photon) beams or electron beams in the range of 4 to 25 million electron volts (MeV). The machine has a lot of flexibility to aim the beam and shape it in arbitrary ways. The radiation oncologist would like to take advantage of this flexibility by designing a plan that achieves a curative dose in the target region while minimizing the dose to surrounding tissues. This is possible because the amount of "beam on" time for a given daily treatment is only a few minutes. Therefore it is practical to plan and deliver a daily treatment consisting of multiple short irradiations, each with its own aperture and direction. The idea is that combining many (or even a few) irradiation steps will bring the dose in the overlap region (the target) to the desired level, while no one irradiation step will deliver an unacceptable dose to the surrounding tissue. Thus the radiation therapy planning problem is not to choose the best single direction from which to irradiate, but to create a combination of irradiations that achieves the goal.

\*\*\* something about clinical efficacy of RT (maybe the Luther Brady article?)

Radiation therapy is probably the earliest example of application of computer programming to the solution of clinical treatment decision problems, dating from





Figure 14.1: A radiation treatment machine, with a member of the UW staff positioned as a patient would be, and a therapist to operate the machine. Once the patient and treatment machine are properly positioned, the therapist leaves the treatment room, then turns on the radiation beam, while monitoring the patient via closed-circuit television. This procedure is repeated for each treatment beam direction.

the mid-1960's. The first few decades of computer modeling focused on the physics of radiation absorption in human tissue relative to the geometry of radiation beams produced by then current machinery, and on the design of interactive systems that could be used for manual computer aided design. These systems are called Radiation Therapy Planning (RTP) systems. The physics problem is this: given a direction, aperture and other settings for a radiation machine to irradiate a particular patient whose body shape is known, how can the radiation dose delivered to any point inside the patient's body be predicted (computed)? Many formulas and algorithms to answer this question have been proposed, implemented, validated and used in the clinical practice of radiation therapy.

With the introduction of Computed Tomography (CT) and Magnetic Resonance (MR) imaging in the 1980's and 1990's, the utility of these computer aided design programs took an enormous jump. Medical physicists were enthusiastic about the possibility of improving the accuracy of the calculation of internal radiation dose, by using CT image data as a proxy for the radiation absorption properties of the internal tissues, rather than assuming the patient's body had uniform density. The

radiation oncologists' attention was drawn to the possibility of more accurately localizing the tumor and estimating its local spread, by using the visual display and computer drawing tools. The next generation of RTP systems provided improved visualization of the beam geometry also, so that the planner could directly apply his/her intuition about anatomy and the capabilities of the machinery.

However, the knowledge of what radiation modalities to choose, and how best to configure them remained largely outside the domain of formal computation. Some of the steps in planning radiation therapy can in fact be formalized in a small scale computational theory, or fragments of a theory. This chapter will develop a few examples of how that can be done.

## 14.2 Radiotherapy Planning Software

need the citation here, also  
perhaps case details

The process of designing good radiation plans requires the use of an RTP system. A significant liability judgement was awarded to a patient who had been paralysed from the neck down by an unsatisfactory radiation treatment. The court found that the practitioner was liable because he did not use an RTP system to plan the treatment but based his plan only on X-ray films. This happened well before RTP systems were in most radiation therapy clinics. The expectation of the court seemed to be that when technology becomes available and is demonstrated to be effective, it should be used, even if it is not yet uniformly or generally in use.

The basic steps in radiation treatment planning are:

1. Gather clinical and physical data
2. Decide general approach
3. Select radiation type(s)
4. Delineate the target volume(s)
5. Configure radiation beams
6. Check acceptability and feasibility

The last two steps may be repeated many times, using the interactive capabilities of the RTP system. A typical screen display from the author's Prism RTP system is shown in figure 14.2. Various control panels and subpanels provide interactive control of the positioning of radiation beams with respect to the patient, as well as the viewpoints and types of visualizations.

Two factors make radiation therapy effective and practical. Understanding these makes clear the role of computer simulation as a step in the design of radiation treatment.

The first factor concerns the physics of radiation beams. For high energy photons, the maximum dose, or energy deposition, is not at the surface, but can be as much as several centimeters below the skin surface. This is illustrated by the graph of dose vs. depth in figure 14.3.

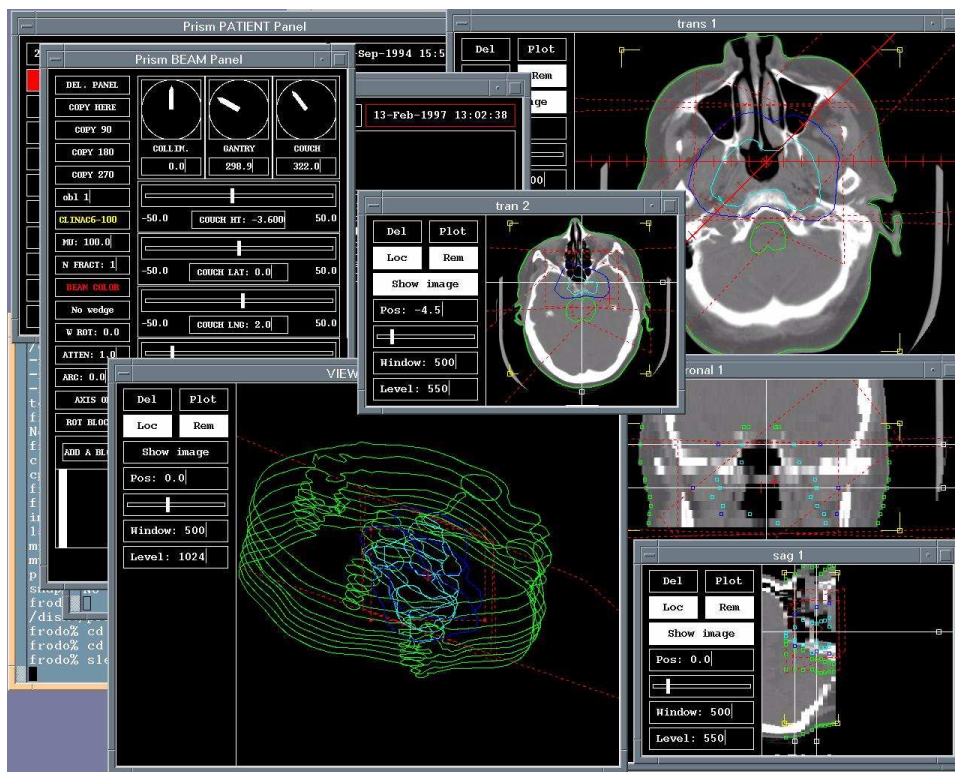


Figure 14.2: An example RTP system display: the Prism system

The second factor concerns the geometric capabilities of the radiation treatment machine. It is possible to deliver the radiation by aiming the machine in each of several directions, turning it on for a short time from each, so that the aggregate dose to the tumor is high, but the dose deposited in the surrounding tissues is spread out, and therefore lower. Figure 14.4 illustrates a two-dimensional cross section of a plan showing two radiation “beams” overlapping to give this effect.

Thus the problem of deciding how to treat the patient is one of choosing directions, apertures, and relative amounts of radiation from some number of radiation beams. A collection of radiation beams thus specified constitutes a plan. Each beam is set up by the therapist, the machine delivers a preset amount of radiation, and the therapist returns to the treatment room to move the machine to the next position. A typical daily treatment takes about a half hour.

In figure 14.4 there are lines showing isolevel contours for various radiation dose levels. These come from interpolation of a grid or array of computed values of radiation dose that would result from the specified arrangement of radiation beams. The formulas used to predict the dose at any point within a patient from

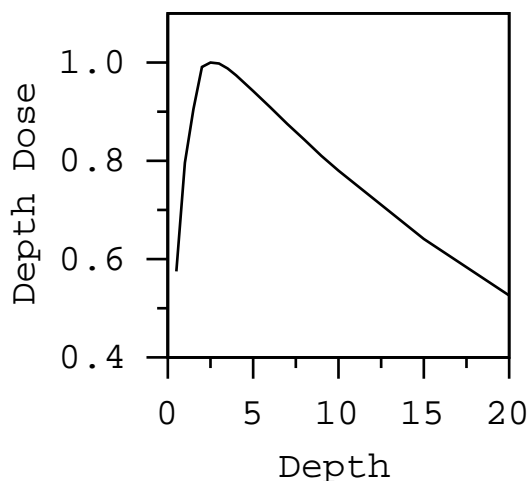


Figure 14.3: Dose from a single radiation beam vs. depth in tissue

any specified radiation beam are partly based on principles of radiation physics and partly empirical, interpolating measured data in standard test conditions. The refinement and testing of such formulas is a large and active area of research in medical physics. A survey of the principles can be found in textbooks, e.g., Khan [77], and the particular formulas used in the Prism RTP system are described in exhaustive detail in a technical report [70].

Although methods are well known for computing the physical dose received anywhere in the body from a given beam configuration, there is no established method for solving the inverse problem of computing a set of beam parameters, given a desired dose distribution. In typical clinical practice a dosimetrist (a person with special training and experience) uses an RTP system to display the geometry and dose distribution. The dosimetrist looks for regions of inadequate dose to the target and excessive doses to sensitive structures, and applies modifications that will correct the problem and thereby improve the plan. Therefore the “configure radiation beams” step above is an interactive generate-test loop.

The importance of having powerful three dimensional radiation treatment planning software tools has been evident for some time [38, 39]. Much work has focused on enhancing the delineation of anatomy [15, 134], and providing visualization tools [102].

Early systems ran as teletype applications on dialup time sharing systems, then on dedicated minicomputers with frame buffer displays. As the computer systems and graphics hardware became faster and cheaper, the medical physicists writing RTP programs became more ambitious about the kind of interactive graphic displays they implemented. Through most of this period, from the late 1970’s through the 1990’s, most of the emphasis was on arcane, clever and practical features and software tools, rather than on innovations in software architecture or on automat-

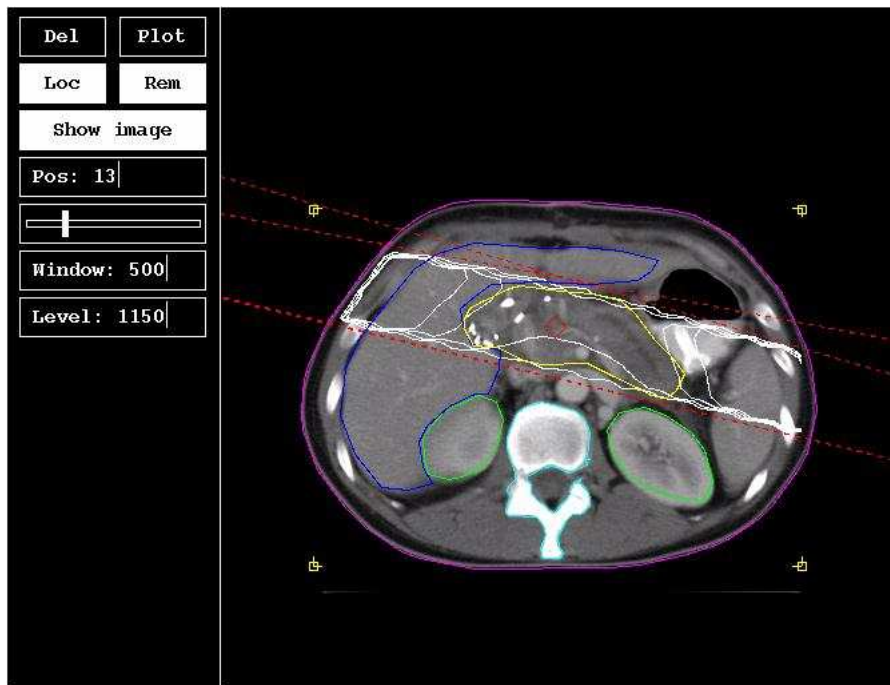


Figure 14.4: A simple radiation treatment plan cross sectional view

ing the logic of the planning process itself. This focus remains unchanged today. The size of such systems ranges from 80,000 to over 500,000 lines of source code. Numerical calculations and interactive graphics have a large role in these systems. The programs are commonly written in programming languages like C or C++ and use graphics libraries like OpenGL [96].

### 14.3 Locating the Target

The elegance, power and effectiveness of radiation therapy is that the radiation machinery can be aimed and adjusted to precisely direct the radiation to any desired location, unlike chemotherapy, where drugs are infused into the patient's bloodstream, in hope that the drugs will be selectively absorbed by the tumor cells. Thus, a basic principle of radiation therapy is that one must know where the target is. In simple terms, you cannot aim precisely at something whose location is unknown. This is a problem involving anatomy, tumor biology, and spatial reasoning.

Radiation oncologists and others, through the International Commission on Radiation Units (ICRU), have created a series of definitions for the entities involved in defining the target for radiation therapy [57, 58]. By defining these concepts, they can more precisely describe how treatments should be designed, and ultimately this

will also add precision to clinical trials.

In delineating the target of radiation therapy for a particular patient, the radiation oncologist must consider what is palpable and/or can be seen in medical image studies such as CT, MRI or PET, as well as the more traditional radiographic X-ray images. This is the starting point. The volume that includes only the visible or palpable tumor mass is called the Gross Tumor Volume, or GTV. The GTV is usually drawn by hand using interactive drawing software on one or more sets of cross sectional image studies. As with organ delineation, the GTV can be represented in the RTP program as a set of planar contours, with each contour being a set of vertices located in the volume including the patient. This is a tedious task, requiring hand drawing, plane by plane. Unlike high contrast organs like bones and lung, tumors are difficult to detect automatically because their tissue density and other properties are close to the surrounding tissues, and the boundaries are ill-defined.

It is well known that for many tumors, there is local and regional spread, to surrounding cavities, lymph nodes, and other structures. This depends on the size of the tumor, the type of cells, and most critically the surrounding anatomy. These migrant tumor cells cannot be made visible with any imaging techniques known today. The volume that includes the local and regional spread is known as the Clinical Target Volume, or CTV.

Pathology findings following surgery have been reported in the literature, usually to address specific questions such as whether or not to include a specific lymph node group in the target volume. Generally tumor spread largely follows anatomic structure and biologic principles, though this is only manifest in the treatment planning process by hand drawing the CTV. As tumors become larger and time elapses, the probability of more distant node groups being positive for microscopic spread increases. Some data are available that quantify this process, and steps have been taken towards a predictive model of tumor metastasis [69]. Section 14.3.2 following develops this idea formally in a small working implementation.

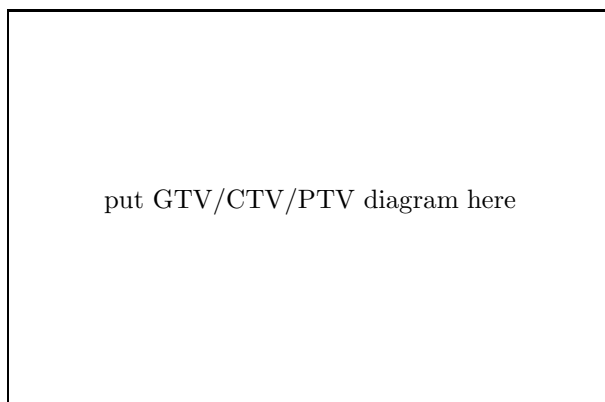


Figure 14.5: A diagrammatic illustration of the idea of GTV, CTV and PTV

Once the CTV is determined, one must allow for the fact that radiation therapy delivery presents a significant physical alignment problem for the radiation therapist. A typical course of treatment includes thirty daily irradiations, each taking only a few minutes. The patient comes into the clinic each day, and the actual setup geometry can vary because the tumor may slightly shift inside the patient's body, the machine is difficult to align consistently to the external landmarks on the patient, and even with the best efforts at immobilizing the patient, the tumor and/or the body will shift during the few minutes of treatment. Thus, the ICRU defined a third volume, the Planning Target Volume, or PTV, which is enlarged from the CTV to allow for these variations. If the treatment plan is designed for the PTV, one would be reasonably confident that the CTV would receive adequate radiation dose, despite the variations. The data available on setup variability and tumor motion make it possible to compute the PTV from the CTV [80, 6]. Evaluation of the accuracy of such a computational model of the PTV is challenging [75, 76]. A PTV model implementation is developed in section 14.3.3.

The ICRU also defined a treatment volume, which represents the volume that was actually irradiated. This volume will depend, of course, on the plan of irradiation that is chosen for actual treatment. Its boundaries depend on the criteria chosen. The volume irradiated to 50 Gray will be larger than the volume irradiated to 60 Gray, and so on. The treatment volume idea is not practical for radiation therapy planning. Instead it is common practice to compute so-called “dose-volume histograms”, which represent information about how much dose a particular treatment plan will deliver to the volumes already defined, such as critical organs (uninvolved tissues such as lung, liver, kidney) and the various target volumes. We won't address either of these problems here.

### 14.3.1 Classification of Tumor Sites

Knowledge about the etiology, pathology, and treatment of tumors depends so much on the location of the primary (and ultimately the metastases) that it is used generally in cancer textbooks as an organizational theme. Also, for the purpose of formalizing knowledge about the computation of the CTV and the PTV, as well as for automating other aspects of radiotherapy planning, it is useful to have a classification system for tumors according to anatomic site. As an example of the importance of site, for tumors in the head, neck and brain, the patient can be held quite rigidly with a custom molded plastic mask, while for lung tumors, body position is much less reproducible. Similarly, tumor motion is highly dependent on site, as some tumors are rigidly fixed to the surrounding anatomy while others move as much as a centimeter or even two. When two pieces of knowledge apply to a specific tumor site, one for that site, and one for the larger region (e.g. nasopharynx and its region, head and neck), the more specific one should apply over the more general one, when there is a conflict.

We focus on the head and neck, which for radiation oncologists includes everything above the shoulders except for the brain and spinal cord. This is a particularly complex anatomic region, where precision in aiming radiation is very important.

Head and neck cancers are a heterogeneous collection of malignancies arising in

check this for more precise definition

multiple anatomic subsites, each with unique clinical, pathological, and prognostic features. The vast majority of these cancers are squamous cell carcinomas that originate in the normal mucosal lining located throughout the head and neck. As these squamous cancers enlarge, they tend to spread from primary location to nearby regional lymph nodes in a predictable sequence. Curative treatment of these cancers, therefore, has historically required attention to the draining regional lymph nodes that could harbor microscopic disease.

Tumor anatomic sites can be represented in a hierarchical structure, a simple directed acyclic graph. Each node in the graph is a tumor site. The nodes can be represented by instances of a CLOS class. The slots should be the **name** of the site (which we represent by a symbol), the immediate larger region to which it belongs (the **part-of** slot) and the subregions or subsites it contains (the **parts** slot). In case the site has no parts, i.e., it is the most fine grained location in common use, its **parts** slot will be empty, i.e., it will have the value **nil**.

Here is the representation of the class whose instances will represent the tumor site hierarchy:

```
(defclass tumor-site ()
  ((name :initarg :name :accessor name)
   (part-of :initarg :part-of :accessor part-of)
   (parts :initarg :parts :accessor parts
          :documentation "A list of symbols naming
                        daughter nodes"))
  (:documentation "Each instance represents a single anatomic
                  site or larger anatomic region. The tree
                  structure implied by the parts and part-of
                  slots describes anatomic relationships of
                  tumor sites and groups of tumor sites.")
  (:default-initargs :name nil :part-of nil :parts nil)
)
```

\*\*\* why not **is-a** here? Because subregion is not the same as subset, and sites are *instances*, not classes, cf. FMA

\*\*\* include **fma-name** slot?

It is useful to be able to refer to each instance of the **tumor-site** class by name, rather than having to search a global list. One convenient way to do this is to arrange for each instance to become the value of the symbol in the **name** slot, when the instance is created. This also allows for the values in the **part-of** and **parts** slots to be symbols rather than instances, which simplifies the representation of the tree in a file, which keeping traversal of the tree straightforward.

To make this happen automatically we just provide a simple augmentation of instance initialization, with the following:

```
(defmethod initialize-instance :after ((site tumor-site)
```



```

                                &rest initargs)
  "This method makes the site instance available by name"
  (set (name site) site))

```

The `set` function rather than the `setf` macro is used because we want to set the value of the symbol that is in the `name` slot, rather than update the contents of the `name` slot. The `&rest` argument is not used, but is required in order to be consistent with the definition of the `initialize-instance` generic function.

Note that this obviates the need for a global list to search for a site (e.g. by name). If you know the name, you have immediate access (in  $O(1)$  or constant time).

Here is some of the tumor site hierarchy in terms of constructor functions:

```

(make-instance 'tumor-site
  :name 'INFERIOR-WALL
  :part-of 'NASOPHARYNX
  :parts NIL
)
(make-instance 'tumor-site
  :name 'NASOPHARYNX
  :part-of 'PHARYNX
  :parts '(POSTERIOR-SUPERIOR-WALL LATERAL-WALL INFERIOR-WALL)
)
(make-instance 'tumor-site
  :name 'PHARYNX
  :part-of 'HEAD-AND-NECK
  :parts '(HYPOPHARYNX OROPHARYNX NASOPHARYNX)
)

```

The full site hierarchy for the head and neck can be constructed by hand from information in a commonly available cancer treatment reference manual [114]. There are altogether 78 sites in the head and neck.

\*\*\* what about `print-tree`? an alternate to `get-object` and `put-object`? why or why not?

Note that these are *tumor classification* categories, *not* anatomical entities according to the UW Foundational Model of Anatomy (FMA). The hierarchy in this model is that of the radiation oncologist's organization of knowledge about tumors. However, it raises an interesting question: can an alignment of the oncologist's understanding of tumors with the anatomist's understanding of structure provide new insights that could be used for designing better therapy? The construction of these formal models is an essential step in the process of answering such questions.

The site hierarchy allows the organization of general knowledge and specific knowledge to be used efficiently. The control strategy can be to use all applicable rules, or the most site-specific available rule, when rules conflict. Rather than having to repeat general information for each specific site, a simple recursive search

This is not strictly true, since access to symbol values is through a hash table or some indirect indexing but we are hiding it in the Lisp implementation instead of the application

check if more recent edition is still appropriate, or maybe one of the big cancer books

from the specific tumor site through all the contained sites, following the **part-of** relation, implemented by the **within** predicate function, collects all applicable rules. This principle underlies the use of many different ontologies that form hierarchies, including subsumption hierarchies as well as those formed by **part-of** relations.

relate this back to Chapter 3

First we must convert the slot values in our ontology to assertions in the inference system. This can be done initially by iterating over the collection of site instances, or individually as each site is created with **make-instance**, as discussed in Chapter 3. Specifically, we augment the **initialize-instance** method above to do this.

```
(defmethod initialize-instance :after ((site tumor-site)
                                       &rest initargs)
  "This method makes the site instance available by name
   and registers a rule for the part-of slot."
  (set (name site) site)
  (assert-value 'part-of (name site) (part-of site)))
```

The **within** function can be defined in the context of mock Prolog (the Graham inference code described in Chapter 3). Location X is within location Y if they are the same, or if X is a part of another location and that location is within the location Y. The function called **same** is also easy to define in this mock Prolog: a thing is the same as itself!

```
(<- (within ?x ?y) (same ?x ?y))

(<- (within ?x ?y) (and (part-of ?x ?z)
                        (within ?z ?y)))

(<- (same ?x ?x))

*** check the above code for termination

*** give examples of use of the within function

*** show how this relates to the FMA - add slot fma-name
```

### 14.3.2 Computing the CTV

The Clinical Target Volume is a region that does not correspond to a visible organ or actual structure, but instead is a region defined by where the radiation oncologist believes the tumor cells have spread. Therefore, it cannot be generated automatically from image data alone. However, many tumors are known to proliferate (metastasize) through the lymphatic system, so a model of the lymphatic system can potentially provide a means for computationally predicting the regional spread of these cancer.

### Anatomic Model of Lymphatic Drainage

Using the Foundational Model of Anatomy (FMA) developed by the University of Washington Structural Informatics Group[112] (see Chapter 9), we can construct a small program that will map the lymphatic drainage pathways from any anatomical site. A query to the FMA can tell us what lymphatic chains drain a given anatomical entity (and its parts). Further queries to the FMA can then tell us what lymphatic chains are downstream, or *efferent to*, the initial drainages. A recursive query can trace these paths all the way back to the final lymph vessels, the thoracic duct and the right lymphatic duct, which then drain into the blood stream. Once tumor cells reach the blood stream, they can be dispersed very great distances from their origins, but that is not going to be modeled here.

Using the `fms-query` code described in Chapter 9, we start with a function to query for the lymphatic drainage of the entity referred to by a particular term. It just provides a thin layer over the general function for querying about slot contents in the FMA. Note that all the functions in this section have as their first argument a variable called `fms`, which is the open socket stream connected to the FMS.

```
(defun lymphatic-drainage (fms term)
  (get-children fms term "lymphatic drainage"))
```

We need a “downstream” function, which we will call `efferent-to`. It returns the nodes or chains that are downstream from the entity specified by its input term. Since all the entries in the FMA are labeled by strings, we must use string comparison, rather than testing if objects are identical. We treat “efferent to” and “tributary of” as synonyms, but have to make sure to remove duplicates where a next node or chain is listed both ways.

later remove this - it is an artifact of the evolution of the FMA

```
(defun efferent-to (fms term)
  (remove-duplicates (append (get-children fms term "tributary of")
                             (get-children fms term "efferent to"))
    :test #'string-equal))
```

To trace the paths, we use the `path-search` function developed in Chapter 3, at the end of Section 3.5. The `lymphatic-drainage` function gets a set of start nodes, and we will apply `path-search` to each. The `efferent-to` function serves as the `extender` function. To get paths rather than just end points, we use `cons` as the `merge` function.

Since we want a list of all drainage patterns for any anatomical site, and each call to `path-search` will return a list of paths starting from one particular start node, we use the utility function, `mappend`, from Norvig, PAIP, page 19,

```
(defun mappend (fn the-list)
  (apply #'append (mapcar fn the-list)))
```

and then we can write `lymphatic-paths` very compactly:

```
(defun lymphatic-paths (site)
  (mappend #'(lambda (d)
    (path-search d #'efferent-to #'cons #'append))
    (lymphatic-drainage site)))
```

\*\*\* fix this paragraph up

If we want to generate paths not in reverse order, change `cons` to something else. Possibly we would need to have some other structure than a list of initial lymphatics, which could be done here or in `lymphatic-drainage`. Might also require slight mods to the `path-search` function.

In some cases, the lymphatic drainage information for a site may not be associated with the anatomical entity but with its parts instead. In this case, it is necessary to query the FMA about the parts of an organ or other anatomic entity, and then find the lymphatic drainage paths for each of the parts. In anatomy, there are several kinds of relations under the general category of parts. The one that is relevant here is the *regional part* relation (refer back to Chapter 9 for the definition).

The `all-nodes` function returns a list of any paths directly from the current site, by calling the `lymphatic-paths` function. It also queries for the regional parts, and then for each part, recursively obtains the same information.

rewrite this as the embedded version, explain, and use `mappend`

```
(defun all-nodes (fms site)
  (list (lymphatic-paths fms site)
    (let ((parts (get-children fms site "regional part")))
      (format t "PARTS OF ~S:~%~S~%" site parts)
      (append (list site 'parts)
        (mapcar #'(lambda (part) (all-nodes fms part))
          parts))))))
```

In this function, the `format` call just provides some output to the command line window to reassure you that it is making progress. It is not necessary, and in an embedded context, you would omit this form.

## Lymphatics and Tumor Spread

The next step in computing the CTV is to determine how far a given tumor is likely to have spread along the pathways computed in the previous section. One way to do this is to assign probabilities based on surgical data from head and neck resections.

The development of standardized surgical approaches to the neck (e.g., radical neck dissection, selective neck dissection) has paralleled development of standardized nomenclature for the various nodal regions (“levels”) in the neck[109, 125]. (See Table 14.1.)

A mapping from the lymphatic drainage entities in the FMA to the clinical region system used by surgeons and radiation oncologists is not readily available. This has to be done by hand. Here is an example table mapping a portion of the lymphatic drainage of the head and neck.

Table 14.1: Current classification of cervical lymph nodes

Level Ia	Submental group
Level Ib	Submandibular group
Level II	Upper jugular group
Level III	Middle jugular group
Level IV	Lower jugular group
Level V	Posterior triangle group

```

(defconstant *clinical-regions*
  '(("Apical axillary lymphatic chain")
    ("Left apical axillary lymphatic chain")
    ("Left central axillary lymphatic chain")
    ("Left subclavian lymphatic trunk")
    ("Left subscapular axillary lymphatic chain")
    ("Right apical axillary lymphatic chain")
    ("Right central axillary lymphatic chain")
    ("Right lymphatic duct")
    ("Right lymphatic duct tree")
    ("Right subscapular axillary lymphatic chain")
    ("Right subclavian lymphatic trunk")
    ("Subclavian lymphatic trunk")
    ("Subscapular axillary lymphatic chain")
    ("Thoracic duct")
    ("Thoracic duct tree")

    ("Basal lingual lymphatic tree" "B")
    ("Central axillary lymphatic chain" "B")
    ("Central lingual lymphatic tree" "B")
    ; I take deep cervical lymphatic chain to be the deep
    ; Posterior C. chain because ther already
    ; is a deep lateral chain (test this with nasopharynx)
    ("Deep cervical lymphatic chain" "Va" "Vb")
    ("Deep parotid lymphatic chain" "P")
    ("Inferior deep lateral cervical lymphatic chain" "IV")
    ("Jugular lymphatic chain" "VI" "IV")
    ("Jugular lymphatic trunk" "VI" "IV")
    ("Jugulo-omohyoid lymphatic chain" "III")
    ("Jugulodigastric lymphatic chain" "IIa")
    ("Left deep cervical lymphatic chain" "Va" "Vb")
    ("Left inferior deep lateral cervical lymphatic chain" "IV")
    ("Left jugular lymphatic tree" "IV") ;VI?
    ("Left jugular lymphatic trunk" "VI" "IV")
    ("Left retropharyngeal lymphatic chain" "RP")
    ("Left submandibular lymphatic chain" "Ib")
    ("Left superficial cervical lymphatic chain" )

```

Table 14.2: Prevalence of subclinical lymph node involvement according to tumor site

Tumor location	Nodal Level				
	I	II	III	IV	V
Oral tongue	14%	19%	16%	3%	0%
Floor of mouth	16%	12%	7%	2%	0%
Alveolar ridge	27%	21%	6%	4%	2%
Retromolar trigone	19%	12%	6%	6%	0%
Buccal mucosa	44%	11%	0%	0%	0%

```

("Left superior deep lateral cervical lymphatic chain"  "Va")
("Marginal lingual lymphatic tree"                    "B")
("Right deep cervical lymphatic chain"                 "Va" "Vb")
("Right inferior deep lateral cervical lymphatic chain" "IV")
("Right jugular lymphatic tree"                        "IV") ;VI?
("Right jugular lymphatic trunk"                      "VI" "IV")
("Right retropharyngeal lymphatic chain"              "RP")
("Right submandibular lymphatic chain"                "Ib")
("Right superficial cervical lymphatic chain"          )
("Right superior deep lateral cervical lymphatic chain" "Va")
("Submandibular lymph node"                          "Ib")
("Submandibular lymphatic chain"                     "Ib")
("Submental lymphatic chain"                         "Ia")
("Superficial cervical lymphatic chain"               )
("Superior deep lateral cervical lymphatic chain"
  "IIa" "III" "IV"))

```

Several published series of surgically-treated patients have provided valuable pathologic data allowing statistical estimation of the risk for subclinical nodal involvement according to these neck subdivisions[16, 17, 18, 117]. The example data shown in Table 14.2 are adapted from Shah, et. al.[117]

\*\*\* remaining sections - probabilistic model from above data, example predictions, from Noah Benson AMIA paper

### 14.3.3 Computing the PTV

The automated generation of a “Planning Target Volume” (PTV) from the Clinical Target Volume, or CTV, takes account of internal and external motions during and between treatments, based on qualitative facts about the tumor’s anatomical location, its histology, and the treatment plan itself. Like the CTV model, it illustrates how a classification system for tumor locations can be used to organize knowledge about their physical behavior. It also ties together numerical and symbolic modeling in a relatively simple example, and illustrates problems inherent in evaluating

the accuracy of such computations.

Margins are distance values from which to compute an expanded volume representation from the GTV or the CTV to make the contours of the planning target volume (PTV). There are several different types of margin allowances to account for different kinds of variability in tumor location. In some circumstances the appropriate margin may be different for each of the three directions,  $x, y, z$ . For example, it may be possible to immobilize the patient well against left-right motion, but the movement up and down due to breathing cannot be controlled. To accommodate this, the margin is represented as a three element list of numbers rather than a single number.

\*\*\* show how used with example from Prism, then the top down `target-volume` code.

One factor is the mechanical motion of the patient during the treatment, due to the patient's breathing, or to the patient not being completely immobilized. In treating head and neck tumors, for example, it is usual to make a custom fitted mask from a plastic mesh that softens and stretches when heated with warm water, then sets once formed to the patient's face shape. The amount of motion allowance in this case is independent of tumor site, and can be expressed in a rule as follows:

```
(<- (pt-movement ?x (0.1 0.1 0.1))
    (and (location ?x ?y)
         (within ?y head-and-neck)
         (immob-dev ?x mask)))
```

This rule states that an allowance of 0.1 cm should be allowed in each direction for movement of the patient during treatment, when a mask is used and the tumor site is in the head or neck.

Another kind of margin allowance is the variation in positioning the patient on the treatment couch, and in aligning the patient with the radiation beam. This is also affected by the kind of immobilization device used, since the device can be marked and laser alignment lights in the treatment room used to position the patient before treatment starts each day. A rule representing day-to-day variability in positioning the patient with respect to the radiation beam might look like this:

```
(<- (setup-error ?x (0.5 0.5 0.5))
    (and (location ?x ?y)
         (within ?y head-and-neck)
         (immob-dev ?x mask)))
```

declares that a margin for patient positioning of 0.5 cm in each of the three directions should be allowed, in general for head and neck tumors, if a mask type of patient immobilization device is used.

If no immobilization device is used (there may be practical reasons why a particular patient may be unsuitable for this technique), the margins will have to be larger. Thus we have rules for this case, similar to above.

```
(<- (pt-movement ?x (0.3 0.3 0.3))
    (and (location ?x ?y)
         (within ?y head-and-neck)
         (immob-dev ?x none)))
```

```
(<- (setup-error ?x (0.8 0.8 0.8))
    (and (location ?x ?y)
         (within ?y head-and-neck)
         (immob-dev ?x none)))
```

For tumors in the lung, the margins are slightly different. A plastic mask will not allow for breathing, so a customized foam filled box-like device is used instead, called an “alpha cradle”. This is not as rigid as the mask.

```
(<- (setup-error ?x (0.6 0.6 0.6))
    (and (location ?x ?y)
         (within ?y lung)
         (immob-dev ?x alpha-cradle)))
```

```
(<- (pt-movement ?x (0.2 0.2 0.2))
    (and (location ?x ?y)
         (within ?y lung)
         (immob-dev ?x alpha-cradle)))
```

Finally, there is a third type of variability that requires aiming at a larger volume and therefore contributes to determining the PTV. During treatment, some tumors can move around internally to the patient’s body surface, especially those that are in body cavities, viscera (the gut), or low density regions like lung. We refer to these as **tumor-movement** margins. For example, nasopharyngeal tumors are usually fixed and rigid, but lung tumors can move slightly internally during normal breathing. In the lung, the extent and direction of motion depends on the detailed location within the lung, so a region has to be specified.

```
(<- (tumor-movement ?x (0.0 0.0 0.0))
    (and (location ?x ?y)
         (within ?y nasopharynx)))
```

```
(<- (tumor-movement ?x (0.8 0.0 0.0))
    (and (location ?x ?y)
         (within ?y lung)
         (region ?x mediastinum)))
```

```
(<- (tumor-movement ?x (0.5 0.5 1.0))
    (and (location ?x ?y)
         (within ?y lung)
         (region ?x lower-lobe)))
```



These rules come from observations utilizing portal imaging techniques.

\*\*\* elaborate a little here, cite sources for data

To solve a particular case, simply assert the particular facts for that case, run a query using the `with-answer` macro (see page 96 in Section 3.2) to get each of the margin values, and then apply the margin values to expand the clinical target volume contours, resulting in a set of contours for the planning target volume.

\*\*\* rewrite the following in a top-down style

The `generate-margins` function just uses backward chaining (the Graham inference code mentioned earlier) to collect the applicable margins. The following code assumes that the value of the symbol `tumor` is an instance of class `tumor`, and the value in the `site` slot is a symbol naming the anatomic site (one of the anatomy tree nodes).

\*\*\* this code is pretty implicit. better to follow the actual Prism code

```
(defun generate-margins (tumor immob)
  (assert-value 'location tumor (site tumor))
  (assert-value 'immob-dev tumor immob)
  (list (list 'setup-m
    (with-answer (setup-error ?y ?x)
      (if (eql ?y tumor) (return ?x))))
    (list 'tumor-m
      (with-answer (tumor-movement ?y ?x)
        (if (eql ?y tumor) (return ?x))))
    (list 'pt-m
      (with-answer (pt-movement ?y ?x)
        (if (eql ?y tumor) (return ?x))))))
```

Once each of the types of margins are determined, they are combined in quadrature (root-mean-square), and the resulting overall values (x, y, and z) are added to the tumor volume contour points to produce the target volume contour points. For convenience we define a function, `rms`, that takes three lists of margin values (in actual use, one for each of the three types of margins) and does the arithmetic for each of the *x*, *y*, and *z* directions, returning a list of the resulting values.

\*\*\* ditto here - really needs explanation of data structures

```
(defvar *chi-sq-factor* 1.88
  "The 75 percentile value of chi-square for 2 degrees of freedom")

(defun rms (l1 l2 l3)
  "returns a list whose elements are each the square root of the
  sums of the squares of each of the elements of l1, l2 and l3,
```

```
scaled by the chi-square factor above."
(mapcar #'(lambda (a b c)
  (* (sqrt (+ (* a a) (* b b) (* c c)))
    *chi-sq-factor*))
  11 12 13))
```

this is what Sharon did, but it would be better to do radial projection - maybe fix this in the book

The tumor volume contours are lists of  $x, y$  coordinate pairs, associated with a particular  $z$  value. Each is projected outward cylindrically.

\*\*\* PSZ suggests including the convex hull code and the expansion code. I haven't decided if this is worth the space.

\*\*\* include some screen shots of this in Prism!

## 14.4 Influence Diagrams in RTP

Even with heuristic methods for search, the tradeoffs in radiation therapy between adequate dose to the tumor to sterilize it and overdose to surrounding structures makes it difficult to decide which of a range of candidate plans is best. The problem is that the tradeoffs are related to the individual's goals and preferences in terms of the treatment outcome and long term (quality of life) outcomes. At the UW we have been working on an influence diagram model that addresses this problem. The source code is available, and will serve to illustrate how a probabilistic reasoning system can be connected with a complex modeling system, the UW Prism radiation therapy planning system.

## 14.5 Automated Radiation Beam Placement

This work is a good example of how to combine numerically intensive and sophisticated mathematical modeling with rule-based search. It represents qualitative knowledge about the physics of radiation beams, anatomy and tumors in terms of rules. The rules are used for a generate-and-test search for an optimal treatment plan. The search space is spanned by numerical dimensions as well as symbolic variables, so one must find a way to constrain the search. This was the subject of Witold Paluszyński's Ph.D. dissertation [97, 98] under the supervision of the author.

## Chapter 15

# Public Health Applications

This chapter will include some examples of time series analysis and data collection for detecting epidemic outbreaks and related stuff.

Also:

- GIS systems in Public Health
- data collection on available resources as well as on disease incidence
- visualization and reporting
- monitoring and analyzing discharge data for ambulatory adverse drug events
- prospective population studies: Iceland, UK Biobank, DECODE, NHGRI (US) (from comments by Don Lindberg (NLM Director) at the NLM Training Program Directors meeting, July 2005)



## Chapter 16

# Intelligent Query Processing

As noted previously, Dr. Wanda Pratt has agreed to make available the programs she wrote as part of the DynaCat project. The chapter on information access will focus on the standard methods for search (indexing) and ranking results. This chapter will develop a more detailed description of Dr. Pratt's categorization system, which uses a medical domain knowledge model to organize the search results according to the domain model.



**Part III**

**Engineering Practical  
Systems**





## Chapter 17

# Biomedical Data Integration

### 17.1 Biowarehouse

Peter Karp's project...

### 17.2 Biomediator

Discussion of the implementation of the Biomediator project [25], which is developing a system for data integration in biomedical applications. The Biomediator idea involves creating a mediated schema, translation methods based on formal descriptions of the data sources, and a new query language, PQL [90].

See also Genesereth work on federated databases.



## Chapter 18

# Building Safe Biomedical Software

\*\*\* expand this description and xref radonc chapter

Radiation therapy machinery has been built with computerised control systems since at least the mid-1970's. The newer machines are designed to provide both high intensity electron beams and X-ray beams. Switching between these modes is complex, requiring coordination between multiple subsystems. One such machine was designed with software that erroneously allowed a sequence of control panel keystrokes that put the machine in an unsafe state. In this state, the electron beam was at the extremely high intensity required for producing X-rays, but it was allowed to exit the machine as if the machine were in electron mode. There were several such malfunctions in different cancer treatment centers, resulting in several injuries and deaths. Some time later the machines were shut down, and the model withdrawn from the market. Dr. Nancy Leveson published a discussion of the issues in this design problem [82].

More recently, the author and Dr. Jonathan Jacky collaborated on the design of a computer control system (hardware and software) for the University of Washington Medical Center Clinical Neutron Therapy Facility. This project incorporated formal methods and specifications as a way to insure safe designs and to ease the checking that the resulting system would be safe for use with patients [63]. A by-product of this was a book by Dr. Jacky [60] on the *Z* formal specification language. In this chapter a similar approach to software design with declarative safety conditions and proofs will be presented in Common Lisp and Prolog, without the need for a separate formal specification language.

\*\*\* Also in this chapter some discussion of *PROforma* [33].

## 18.1 An Example: A Simple Radiotherapy Machine

\*\*\* The following is slide material from MEBI531. Also need to acknowledge Jon Jacky

Radiation Therapy Planning involves aiming radiation beams to:

- irradiate all the visible and invisible tumor cells in the region of the primary tumor,
- *not* irradiate any normal tissues unnecessarily.

The machines can produce electrons or X-rays (photons), selectable at the console. Figure 14.1 in Chapter 14 shows a typical radiotherapy machine setup. The process for planning radiation treatments is also described there. Here we will describe the design of a simplified hypothetical radiation treatment machine, to illustrate the principles of safety in treatment machine design.

Radiation therapy machine safety issues include the following:

- Underdose of tumor  $\Rightarrow$  treatment failure
- Overdose of critical structures  $\Rightarrow$  injury or death
- Machine collision with patient  $\Rightarrow$  injury or death
- Patient falls off couch  $\Rightarrow$  injury or death

We will focus only on the overdose issue in our simplified machine design. Figure 18.1 shows a simplified schematic of how a radiation therapy linac works.

The linac control software will implement a finite state machine. Finite state machines need state variables. For a linac they are (with initial values):

```
(defvar beam 'off "Allowed values: off on")

(defvar current 'high "Allowed values: high low")

(defvar target 'in "Allowed values: in out")
```

These predicate functions define some named states:

```
(defun electron-mode ()
  (and (eq1 current 'low) (eq1 target 'out)))

(defun xray-mode ()
  (and (eq1 current 'high) (eq1 target 'in)))
```

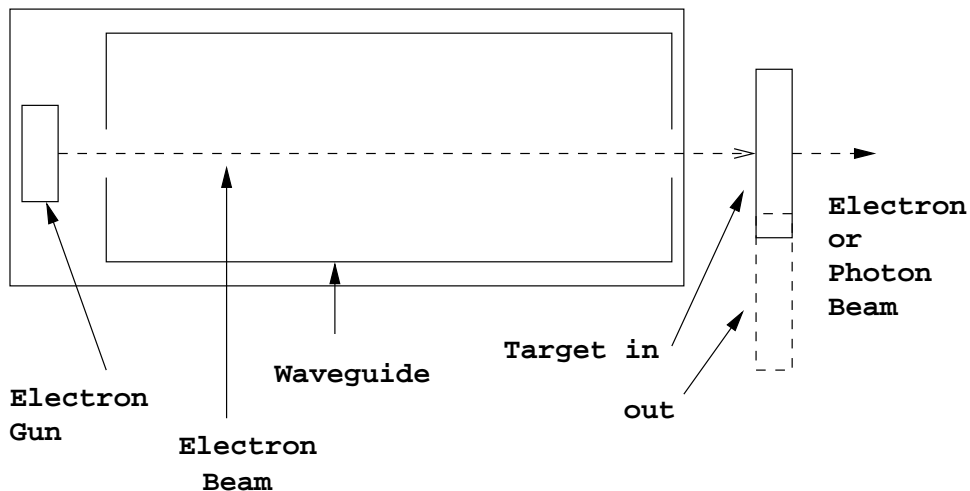


Figure 18.1: A simplified diagram showing how a linear accelerator works.

There are many other states besides these, corresponding to all the other combinations of values for the state variables. Now we can determine which are the safe states. The way we specify the safe states is to define a safety predicate function, **safe**. This function will have the value **t** when the state is safe, and **nil** otherwise.

```
(defun safe ()
  (or (eql beam 'off) ;; safe for sure
      (eql target 'in) ;; safe even with current high
      (eql current 'low) ;; safe even with target out
  ))
```

Next, we have to define functions that implement state transitions, which correspond to actually operating the machine. A state transition is an operation that changes the value of one or more state variables. Each state transition will have preconditions and state variable updates. If the preconditions are not met, the state transition does not take place, i.e., the values of the variables do not change.

The operator will need to press buttons to turn on the beam and turn it off.

```
(defun beam-on ()
  (if (eql beam 'off) (setf beam 'on)))

(defun beam-off ()
  (if (eql beam 'on) (setf beam 'off)))
```

The operator will also need to be able to change from x-ray mode to electron mode and back again. These mode changes should only be done while the beam is off, so that is a precondition.

```
(defun select-electrons ()
  (if (eql beam 'off)
      (setf current 'low target 'out)))

(defun select-xrays ()
  (if (eql beam 'off)
      (setf target 'in current 'high)))
```

We have not said anything about changing the target or the current, except for the two mode settings. But this may be needed as part of the internal design.

```
(defun target-in ()
  (if (eql target 'out) (setf target 'in)))

(defun target-out ()
  (if (eql target 'in) (setf target 'out)))
```

and similarly for the current variable.

Now we can do a safety analysis. Can the system get into an unsafe state? Answer: Yes!

1. Start with beam off, target in, current high. This is safe because the beam is off. It also corresponds to xray mode.
2. Move the target out. This is allowed and the new state is still safe.
3. The **beam-on** state transition is enabled, so do it.
4. Now the system is in an unsafe state, since none of the safety conditions are met.

How can we prevent unsafe transitions? The principal safety concern is that one must not allow the beam to be turned on if that leads to an overdose. We can make this system safe by putting a guard condition on the **beam-on** operation:

```
(defun beam-on ()
  (if (and (eql beam 'off)
          (or (electron-mode) (xray-mode)))
      (setf beam 'on)))
```

If the machine is in any other state, including ones that could become unsafe if the beam were turned on, the beam cannot be turned on.

While this is not always sufficient it illustrates the idea of a safety analysis and how it can lead to an effective remedy.

\*\*\* add dosimetry system

## 18.2 automated analysis

Reformulate the state transitions as links in a network of states (nodes) and apply graph search to determine reachability.

A naive approach would be to do exhaustive search from each safe state that shows that none of the unsafe states are reachable. This is intractable. A better approach is to show that the safe states form a subgraph that has no way out, i.e., that from any safe state, any allowed transition only goes to another safe state.





## Chapter 19

# Using the World Wide Web

While the X window system is widely supported it is also important and sometimes appropriate to use the World Wide Web and its protocols as the base for user interface design. Many books are already available that provide a thorough treatment of writing Web applications. This book will provide an overview of the available resources and a few simple examples to enable the reader to connect the main material of the book to Web front end design. Some of the pitfalls, particularly the problem of maintaining state information, will be discussed.

### 19.1 Computed URLs: CL-HTTP

Although CL-HTTP is not as popular or widely used in bioinformatics as some others, it did pioneer the idea of computed URLs and deserves some discussion from a historical perspective.

This chapter will also provide examples of how to write programs that can leverage the use of a complete Lisp environment within a web server. Two prominent examples are the electronic clinical trials bank project of Ida Sim [124] and the DynaCat project mentioned previously. The chapter on DynaCat will focus on the implementation of search and categorization algorithms. In this chapter the web interface implementation will be presented.

LispWeb, see CHIP web site <http://snpper.chip.org/bio/> and the WWW5 paper [108] and the AIM paper [106].

Also, the Rice, Farquhar et.al. paper from CHI 96 (available in HTML at [http://sigchi.org/chi96/proceedings/papers/Rice/jpr\\_txt.htm](http://sigchi.org/chi96/proceedings/papers/Rice/jpr_txt.htm))

AllegroServe, others?

### 19.2 Common Lisp as a CGI Language

Generating HTML in Lisp, also XML, other functional interfaces (Biolingua again)

### 19.3 Other

Web services, problems and limitations of HTTP and related ideas.

URIs: The extension of the path name idea is the Universal Resource Identifier, or URI, which uniquely locates a file or web page or other datum on the World Wide Web, or the Internet more generally. The generic syntax for URIs is described in an Internet Engineering Task Force (IETF) document, RFC3986 [9].

here too, can include some discussion of how this is used in biomedical applications, e.g., use of URIs in the XML version of the Gene Ontology

## Appendix A

# Software Resources and Current Research

### A.1 Programming Resources for the Reader

Books that provide a thorough introduction to the elements of Common Lisp include: Winston, *Lisp* [140], Graham, *ANSI Common Lisp* [41], and sections of some of the excellent AI books, including Tanimoto, *The Elements of Artificial Intelligence Using Common Lisp* [133] and Norvig, *Paradigms of Artificial Intelligence Programming* [94], mentioned above. Guy Steele's *Common Lisp: the Language, 2nd ed.* [127] is the canonical reference volume for the serious Common Lisp programmer, though for some of the more obscure functions it may also pay to consult the ANSI standard for Common Lisp (available from ANSI). An HTML document derived from the ANSI standard is available on line at

<http://www.lispworks.com/reference/HyperSpec/>

(the Common Lisp HyperSpec). Some commercial Lisp systems include it as well in their software distributions. For a wealth of information about available Common Lisp systems, commercial and freeware, libraries of useful code, tutorials, etc., there are several web sites: ALU, Norvig, Graham, CMUCL repository, and others mentioned later in this book in individual chapters.

Basic introduction to Common Lisp:

- The first three chapters of Norvig, *Paradigms of Artificial Intelligence Programming* [94], and
- The first six chapters of Graham, *ANSI Common Lisp* [41].

For a concise introduction to the Common Lisp Object System (CLOS), see Graham [41], Chapter 11. Norvig [94], Chapter 13 also has a useful treatment of

CLOS and object oriented programming ideas.

\*\*\* add Prolog resources, free Prolog implementations, etc.

\*\*\* other languages??

Also pointer to SLIK, Prism and DICOM. . .

List here sources for Common Lisp, Prolog, bioinformatics software, other relevant software packages.

Where to get Bayes Net code, where to get a simple rule interpreter, other stuff like that (or just present it).

## A.2 Some Research Projects Using Lisp and Prolog

A snapshot of the most notable projects, and web sites like BioLisp and other biomedical informatics resources:

- The Biocyc knowledge library [105, 74], is a collection of pathway databases for over 150 different species, primarily focusing on metabolic processes. It has been and continues to be very useful for answering difficult questions about metabolic processes in microorganisms, and has more recently been extended to include important eukaryotic species.
- BioLingua is a web server that provides programmable access to an integrated collection of biological data [123]. The idea is to provide a higher level “bioinformatics programming language,” in which to express queries and computations on bioinformatics resources, rather than having to learn the low level syntax of the programmer’s interface to NCBI and other data resources. Jeff Shrager, the author of Biolingua, is a Research Fellow at the Carnegie Institution of Washington, Dept. of Plant Biology (Stanford, CA). His lab studies how micro-organisms adapt to environmental change, and spans laboratory molecular biology, microarray technology and computational models. One of their projects, the Chlamydomonas Microarray Project, has resulted in new methods for genome assembly, and for microarray and biological pathway analysis.
- CHIP (Children’s Hospital Informatics Program) Bioinformatics Tools, includes a variety of software tools built on LispWeb, a web server written in Common Lisp by Alberto Riva [106]. They include tools for searching important public databases such as the dbSNP database of single nucleotide polymorphisms, the SwissProt database of proteins, a browser for the Gene Ontology (GO) database of gene products, and a tool for analysing data output by the **phred** program. The **phred** program translates input in “Standard Chromatogram Format” (experimental sequencing data) into a variety of output files listing bases [31].

- PathMiner [86] is a program for inferring or searching for biochemical pathways, based on a representation of basic biochemical principles, rather than creating a curated database of predetermined pathways. This project illustrates the efficiency of Lisp in solving a very computationally intensive problem. It is also designed to be general, in that its basic data can be imported from a variety of sources.
- Antonioti and colleagues at NYU Courant Institute have developed a simulation system that incorporates numerical and symbolic modeling for biological systems [4]. This system provides a flexible approach to modeling systems with large amounts of data, and also for which there are large regulatory networks. The underlying system, `xssys`, is implemented in Common Lisp.

Significant work in bioinformatics and medical informatics has also been done using Prolog. Examples include a system for support of computational analysis using information about expressed genes in erythropoiesis [129], a molecular biology qualitative simulation system [49], and more recently a system for modeling genetic regulation networks [142].

\*\*\* should mention ProForma here also



# Bibliography

- [1] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 2nd edition, 1996.
- [2] Bruce Alberts, Dennis Bray, Julian Lewis, Martin Raff, Keith Roberts, and James D. Watson. *Molecular Biology of the Cell*. Garland Publishing, Incorporated, 4, revised, illustrated edition, 2002.
- [3] American College of Radiology and National Electrical Manufacturers Association. Digital imaging and communications in medicine (DICOM), 2003. the latest version is available from the NEMA DICOM web site, at <http://medical.nema.org/dicom/>.
- [4] M. Antoniotti, F. Park, A. Policriti, N. Ugel, and B. Mishra. Foundations of a query and simulation system for the modeling of biochemical and biological processes. In *Pacific Symposium on Biocomputing*, pages 116–127. World Scientific, 2003.
- [5] M. Ashburner, C. A. Ball, J. A. Blake, D. Botstein, H. Butler, J. M. Cherry, A. P. Davis, K. Dolinski, S. S. Dwight, J. T. Eppig, M. A. Harris, D. P. Hill, L. Issel-Tarver, A. Kasarskis, S. Lewis, J. C. Matese, J. E. Richardson, M. Ringwald, G. M. Rubin, and G. Sherlock. Gene ontology: tool for the unification of biology. the Gene Ontology Consortium. *Nature Genetics*, 25:25–29, May 2000.
- [6] Mary Austin-Seymour, Ira Kalet, John McDonald, Sharon Kromhout-Schiro, Jon Jacky, Sharon Hummel, and Jonathan Unger. Three-dimensional planning target volumes: a model and a software tool. *International Journal of Radiation Oncology Biology Physics*, 33(5):1073–1080, 1995.
- [7] Alexandra Barke. Use of Petri nets to model and test a control system. Master’s thesis, University of Washington, Seattle, Washington, 1990.
- [8] Andreas D. Baxeavanis and B. F. Francis Ouellette, editors. *Bioinformatics: A Practical Guide to the Analysis of Genes and Proteins*. John Wiley & Sons, Inc., 2001.

- [9] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (URI): Generic syntax, 2005.
- [10] W. Dean Bidgood, Jr., Steven C. Horii, Fred W. Prior, and Donald E. Van Syckle. Understanding and using DICOM, the data interchange standard for biomedical imaging. *Journal of the American Medical Informatics Association*, 4(3):199–212, 1997.
- [11] Michael R. Blaha and James R. Rumbaugh. *Object Oriented Modeling and Design With UML*. Prentice Hall, second edition, 2004.
- [12] Richard D. Boyce, Carol Collins, John Horn, and Ira J. Kalet. Qualitative pharmacokinetic modeling of drugs. In *Proceedings of the American Medical Informatics Association Fall Symposium*, 2005.
- [13] Ronald J. Brachman and Hector J. Levesque. *Knowledge Representation and Reasoning*. Elsevier, Inc., 2004.
- [14] Leo Breiman. *Probability and Stochastic Processes: With a View Toward Applications*. Houghton Mifflin Co, 1969.
- [15] James F. Brinkley. A flexible, generic model for anatomic shape: Application to interactive two-dimensional medical image segmentation and matching. *Computers and Biomedical Research*, 26(2):121–142, 1993.
- [16] R. M. Byers, P. F. Wolf, and A. J. Ballantyne. Rationale for elective modified neck dissection. *Head and Neck Surgery*, 10:160–167, 1988.
- [17] F. C. Candela, K. Kothari, and J. P. Shah. Patterns of cervical node metastases from squamous carcinoma of the oropharynx and hypopharynx. *Head and Neck*, 12:197–203, 1990.
- [18] F. C. Candela, J. P. Shah, and D. P. Jaques. Patterns of cervical node metastases from squamous carcinoma of the larynx. *Arch Otolaryngology Head and Neck Surgery*, 116:432–435, 1990.
- [19] Vinay K. Chaudhri, Adam Farquhar, Richard Fikes, Peter D. Karp, and James P. Rice. OKBC: A programmatic foundation for knowledge base interoperability. In *Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98)*, pages 600–607, 1998.
- [20] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.
- [21] Carol Collins and Rene Levy. Drug-drug interaction in the elderly with epilepsy: Focus on antiepileptic, psychiatric, and cardiovascular drugs. *Profiles in Seizure Management*, 3(6), 2004.
- [22] Carlo Combi and Yuval Shahar. Temporal reasoning and temporal data maintenance in medicine: Issues and challenges. *Computers in Biology and Medicine*, 27(5):353–368, September 1997.



- [23] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP: Client-Server Programming and Applications, BSD Socket Version*, volume III. Prentice-Hall, Upper Saddle River, NJ, second edition, 1996.
- [24] Catherine P. D'Ámbrosio. *Computational Representation of Bedside Nursing Decision-Making Process*. Nursing, University of Washington, Seattle, Washington, 2003.
- [25] Loren Donelson, Peter Tarczy-Hornoch, Peter Mork, C. Dolan, J. A. Mitchell, M. Barrier, and Hao Mei. The biomediator system as a data integration tool to answer diverse biologic queries. In *Proceedings of MEDINFO 2004*, pages 768–772. IOS Press, 2004.
- [26] Maureen Donnelly. On parts and holes: The spatial structure of the human body. In *Proceedings of MEDINFO 2004*, pages 351–355. IOS Press, 2004.
- [27] Albert Einstein. Physics and reality. *The Journal of the Franklin Institute*, 221(3), March 1936.
- [28] Albert Einstein. *Ideas and Opinions*. Bonanza Books, 1954.
- [29] Albert Einstein. *Out of My Later Years*. Citadel Press (Carol Publishing Group), 1995. Revised reprint of the original 1950 edition from Philosophical Library, Inc.
- [30] Ramez Elmasri. *Fundamentals of Database Systems*. Addison Wesley, fifth edition, 2006.
- [31] B. Ewing, L. Hillier, M. C. Wendl, and P. Green. Base-calling of automated sequencer traces using phred. I. accuracy assessment. *Genome Research*, 8(3):175–185, 1998.
- [32] Adam Farquhar, Richard Fikes, and James Rice. The ontolingua server: A tool for collaborative ontology construction. *International Journal of Human-Computer Studies*, 46:707–727, 1997.
- [33] John Fox and Subrata Das. *Safe and Sound: Artificial Intelligence in Hazardous Applications*. AAAI Press/The MIT Press, 2000.
- [34] Franz, Inc. Allegro common lisp documentation web site, 2004. <http://www.franz.com/support/documentation/>.
- [35] Douglas B. Fridsma. Representing the work of medical protocols for organizational simulation. In *Proceedings of the American Medical Informatics Association Fall Symposium 1998*, pages 305–309, 1998.
- [36] Douglas B. Fridsma. Error analysis using organizational simulation. In *Proceedings of the American Medical Informatics Association Fall Symposium 2000*, pages 260–264, 2000.

- [37] Erann Gat. Lisp as an alternative to java. *intelligence*, Winter 2000. ACM SIGART quarterly newsletter.
- [38] Michael Goitein and M. Abrams. Multi-dimensional treatment planning: I. delineation of anatomy. *International Journal of Radiation Oncology Biology and Physics*, 9:777–787, 1983.
- [39] Michael Goitein, M. Abrams, D. Rowell, H. Pollari, and J. Wiles. Multi-dimensional treatment planning: II. beam’s eye-view, back projection, and projection through CT sections. *International Journal of Radiation Oncology Biology and Physics*, 9:789–797, 1983.
- [40] Paul Graham. *On Lisp: Advanced Techniques for Common Lisp*. Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1994.
- [41] Paul Graham. *ANSI Common Lisp*. Prentice-Hall, Englewood Cliffs, New Jersey 07632, 1996.
- [42] S. Groothuis and G. G. van Merode. Discrete event simulation in the health policy and management program. *Methods of Information in Medicine*, 39:339–342, December 2000.
- [43] Eric J. Hall. *Radiobiology for the Radiologist*. Lippincott Williams & Wilkins, Philadelphia, PA, 5th edition, 2000.
- [44] Philip D. Hansten and John R. Horn. *The Top 100 Drug Interactions, A Guide to Patient Management*. H&H Publications, 2004.
- [45] Philip D. Hansten and John R. Horn. *Managing Clinically Important Drug Interactions*. Lippincott, Williams & Wilkins, fifth, revised edition, 2006.
- [46] Joel G. Hardman and Lee E. Limbird, editors. *Goodman and Gilman’s The Pharmacological Basis of Therapeutics*. McGraw Hill, New York, 10th edition, 2001. Consulting editor, Alfred Goodman Gilman.
- [47] Elliotte Rusty Harold and W. Scott Means. *XML in a Nutshell*. O’Reilly Media, third edition, 2004.
- [48] Thomas K. Hazlet, Todd A. Lee, Philip D. Hansten, and John R. Horn. Performance of community pharmacy drug interaction software. *Journal of the American Pharmacy Association*, 41(2):200–204, 2001.
- [49] K. R. Heidtke and S. Schulze-Kremer. BioSim—a new qualitative simulation environment for molecular biology. In *Proceedings of the International Conference on Intelligent Systems in Molecular Biology*, volume 6, pages 85–94, 1998.
- [50] William R. Hendee and Russell E. Ritenour. *Medical Imaging Physics*. Wiley-Liss, fourth edition, 2002.

- [51] Lejaren Hiller and Leonard Isaacson. *Experimental music: Composition with an electronic computer*. McGraw-Hill, 1959.
- [52] Douglas R. Hofstadter. *Gödel, Escher, Bach: an Eternal Golden Braid*. Basic Books, Inc., 1979.
- [53] Geoffrey N. Hounsfield. Computed medical imaging. *Science*, 210(4465):22–28, October 1980.
- [54] Lawrence Hunter. Molecular biology for computer scientists. In Lawrence Hunter, editor, *Artificial Intelligence and Molecular Biology*, chapter 1, pages 1–46. out of print, 1993. available on line at <http://compbio.uchsc.edu/hunter/01-Hunter.pdf>.
- [55] Lawrence Hunter. Life and its molecules: A brief introduction. *AI Magazine*, 25(1):9–22, 2004.
- [56] Lawrence Hunter. *(title not yet final)*. MIT Press, 2007.
- [57] International Commission on Radiation Units and Measurements. *Prescribing, Recording and Reporting Photon Beam Therapy*. International Commission on Radiation Units and Measurements, Bethesda, MD, 1993. Report 50.
- [58] International Commission on Radiation Units and Measurements. *Prescribing, Recording and Reporting Photon Beam Therapy (Supplement to ICRU Report 50)*. International Commission on Radiation Units and Measurements, Bethesda, MD, 1999. Report 62.
- [59] Kiyomi Ito, Hayley S. Brown, and J. Brian Houston. Database analyses for the prediction of in vivo drug drug interactions from in vitro data. *British Journal of Clinical Pharmacology*, 57(4):473–486, 2003.
- [60] Jonathan Jacky. *The Way of Z: Practical Programming with Formal Methods*. Cambridge University Press, 1997.
- [61] Jonathan Jacky and Ira Kalet. An object-oriented approach to a large scientific application. In Norman Meyrowitz, editor, *OOPSLA '86 Object Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 368–376. Association for Computing Machinery, 1986. (also *SIGPLAN Notices*, 21(11), Nov. 1986).
- [62] Jonathan Jacky and Ira Kalet. An object-oriented programming discipline for standard Pascal. *Communications of the ACM*, 30(9):772–776, September 1987.
- [63] Jonathan Jacky and Jonathan Unger. Designing software for a radiation therapy machine in a formal notation. In M. Wirsing, editor, *ICSE-17 Workshop on Formal Methods Application in Software Engineering Practice*, pages 14–21, 1995.

- [64] Michael G. Kahn, Larry M. Fagan, and Samson Tu. Extensions to the time-oriented database model to support temporal reasoning in medical expert systems. *Methods of Information in Medicine*, 30(1):4–14, 1991.
- [65] Ira J. Kalet, Drora Avitan, and Robert S. Giansiracusa. DICOM conformance statement, Prism radiation therapy planning system, version 1.2. Technical Report 99-12-01, Radiation Oncology Department, University of Washington, Seattle, Washington, 1999.
- [66] Ira J. Kalet, Drora Avitan, Robert S. Giansiracusa, and Jonathan P. Jacky. DICOM conformance statement, Prism radiation therapy planning system, version 1.3. Technical Report 2002-02-01, Radiation Oncology Department, University of Washington, Seattle, Washington, 2002.
- [67] Ira J. Kalet, Robert S. Giansiracusa, Jonathan P. Jacky, and Drora Avitan. A declarative implementation of the DICOM-3 network protocol. *Journal of Biomedical Informatics*, 36(3):159–176, 2003.
- [68] Ira J. Kalet, Jonathan P. Jacky, Mary M. Austin-Seymour, Sharon M. Hummel, Kevin J. Sullivan, and Jonathan M. Unger. Prism: A new approach to radiotherapy planning software. *International Journal of Radiation Oncology, Biology and Physics*, 36(2):451–461, 1996.
- [69] Ira J. Kalet, Mark Whipple, Silvia Pessah, Jerry Barker, Mary M. Austin-Seymour, and Linda Shapiro. A rule-based model for local and regional tumor spread. In Isaac S. Kohane, editor, *Proceedings of the American Medical Informatics Association Fall Symposium*, pages 360–364. Hanley & Belfus, Inc, 2002.
- [70] Ira J. Kalet, Gavin Young, Robert Giansiracusa, Jonathan Jacky, and Paul Cho. Prism dose computation methods, version 1.2. Technical Report 97-12-01, Radiation Oncology Department, University of Washington, Seattle, Washington, 1997.
- [71] P. D. Karp, M. Riley, S. M. Paley, and A. Pellegrini-Toole. The metacyc database. *Nucleic Acids Research*, 30(1):59–61, 2002.
- [72] P. D. Karp, M. Riley, M. Saier, I. T. Paulsen, J. Collado-Vides, S. M. Paley, A. Pellegrini-Toole, C. Bonavides, and S. Gama-Castro. The ecocyc database. *Nucleic Acids Research*, 30(1):56–58, 2002.
- [73] Peter D. Karp. An ontology for biological function based on molecular interactions. *Bioinformatics*, 16(3):269–285, 2000.
- [74] Peter D. Karp. Pathway databases: A case study in computational symbolic theories. *Science*, 293:2040–2044, 2001.
- [75] Case H. Ketting, Mary M. Austin-Seymour, Ira J. Kalet, Jonathan P. Jacky, Sharon E. Kromhout-Schiro, Sharon M. Hummel, Jonathan M. Unger, and Lawrence M. Fagan. Evaluation of an expert system producing geometric

- solids as output. In Reed M. Gardner, editor, *Proceedings of the Nineteenth Annual Symposium on Computer Applications in Medical Care*, pages 683–687, Philadelphia, PA, 1995. Hanley and Belfus, Inc.
- [76] Case H. Ketting, Mary M. Austin-Seymour, Ira J. Kalet, Jonathan M. Unger, Sharon M. Hummel, and Jonathan P. Jacky. Consistency of three-dimensional planning target volumes across physicians and institutions. *International Journal of Radiation Oncology, Biology and Physics*, 37(2):445–453, 1997.
- [77] Faiz M. Khan. *The Physics of Radiation Therapy*. Lippincott Williams & Wilkins, Philadelphia, PA, third edition, 2003.
- [78] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, Massachusetts, 1991.
- [79] R. K. Klasko. Drugdex system, 2003.
- [80] Sharon Kromhout-Schiro. *Development and Evaluation of a Model of Radiotherapy Planning Target Volumes*. PhD thesis, University of Washington, 1993.
- [81] Gilad J. Kuperman, David W. Bates, Jonathan M. Teich, James R. Schneider, and Dina Cheiman. A new knowledge management structure for drug-drug interactions. In Judy G. Ozbolt, editor, *Proceedings of the Eighteenth Annual Symposium on Computer Application in Medical Care*, pages 836–840. Hanley & Belfus, Inc., 1994.
- [82] Nancy G. Leveson and Clark S. Turner. An investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.
- [83] Jiunn H. Lin. Sense and nonsense in the prediction of drug-drug interactions. *Current Drug Metabolism*, 1(4):305–332, 2000.
- [84] Clement J. McDonald, Stanley M. Huff, Jeffrey G. Suico, Gilbert Hill, Dennis Leavelle, Raymond Aller, Arden Forrey, Kathy Mercer, Georges DeMoor, John Hook, Warren Williams, James Case, and Pat Maloney. LOINC, a universal standard for identifying laboratory observations: A 5-year update. *Clinical Chemistry*, 49(4):624–633, 2003.
- [85] Robert G. McKinnell, Ralph E. Parchment, Alan O. Perantoni, and G. Barry Pierce. *The Biological Basis of Cancer*. Cambridge University Press, 1998.
- [86] D. C. McShan, S. Rao, and I. Shah. PathMiner: predicting metabolic pathways by heuristic search. *Bioinformatics*, 19(13):1692–1698, 2003.
- [87] Merge Technologies, Inc. *MergeCOM-3 Advanced Integrator’s Tool Kit Reference Manual*, 2001.
- [88] Juergen Meyer, Mark H. Phillips, Paul S. Cho, Ira J. Kalet, and Jason N. Doctor. Application of influence diagrams to prostate imrt plan selection. *Physics in Medicine and Biology*, 49(9):1637–1653, 2004.

- [89] S. M. Moore, D. E. Beecher, and S. A. Hoffman. DICOM shareware: a public implementation of the DICOM standard. In R. Gilbert Jost, editor, *Proceedings of the SPIE Conference on Medical Imaging 1994 – PACS: Design and Evaluation*, volume 2165, pages 772–781, Bellingham, Washington, 1994. Society of Photo-optical Instrumentation Engineers (SPIE).
- [90] Peter Mork, Alon Halevy, and Peter Tarczy-Hornoch. A model for data integration systems of biomedical data applied to online genetic databases. In *Proceedings of the American Medical Informatics Association (AMIA) Fall Symposium*, pages 473–477, 2001.
- [91] M. A. Musen, J. H. Gennari, H. Eriksson, S. W. Tu, and A.R. Puerta. PROTÉGÉ-II: Computer support for development of intelligent systems from libraries of components. In *Proceedings of the World Congress on Medical Informatics, MEDINFO’95*, pages 766–770, 1995.
- [92] Peter Naur. Revised report on the algorithmic language Algol-60. *Communications of the ACM*, 6:1–20, 1963.
- [93] James R. Newman. *The World of Mathematics*. Simon and Schuster, 1956. reprinted by Dover Publications, 2000.
- [94] Peter Norvig. *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, San Mateo, California, 1992.
- [95] D. E. Oliver, D.L. Rubin, J. M. Stuart, M. Hewett, T. E. Klein, and R. B. Altman. Ontology development for a pharmacogenetics knowledge base. In *Proceedings of the 2002 Pacific Symposium on Biocomputing*, pages 65–76, 2002.
- [96] OpenGL Architecture Review Board, Mason Woo, Jackie Neider, Tom Davis, and Dave Shreiner. *OpenGL Programming Guide*. Addison-Wesley, third edition, 1999.
- [97] Witold Paluszyński. *Designing radiation therapy for cancer, an approach to knowledge-based optimization*. PhD thesis, University of Washington, 1990.
- [98] Witold Paluszyński and Ira Kalet. Design optimization using dynamic evaluation. In N. S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 1408–1412. Morgan-Kaufmann, 1989.
- [99] Mor Peleg, Aziz A. Boxwala, Omolola Ogunyemi, Qing Zeng, Samson Tu, Ronilda Lacson, Elmer Bernstam, Nachman Ash, Peter Mork, Lucila Ohno-Machado, Edward H. Shortliffe, and Robert A. Greenes. Glif3: The evolution of a guideline representation format. In *Proceedings of the American Medical Informatics Association Annual Symposium*, pages 645–649, 2000.

- [100] Mor Peleg, Samson Tu, Abhijit Manindroo, and Russ B. Altman. Modeling and analyzing biomedical processes using workflow/petri net models and tools. In *Proceedings of MEDINFO 2004*, pages 74–78. IOS Press, 2004.
- [101] Mor Peleg, Iwei Yeh, and Russ B. Altman. Modeling biological processes using workflow and petri net models. *Bioinformatics*, 18(6):825–837, 2002.
- [102] Photon Treatment Planning Collaborative Working Group. Three-dimensional display in planning radiation therapy: A clinical perspective. *International Journal of Radiation Oncology, Biology and Physics*, 21:79–89, 1991.
- [103] John R. Pierce. *An Introduction to Information Theory: Symbols, Signals and Noise*. Dover Publications, Inc., New York, NY, second, revised edition, 1980.
- [104] Raghu Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1998.
- [105] The BioCyc research group. The BioCyc knowledge library. <http://www.biocyc.org/>.
- [106] A. Riva, R. Bellazzi, G. Lanzola, and M. Stefanelli. A development environment for knowledge-based medical applications on the world-wide web. *Artificial Intelligence in Medicine*, 14(3):279–293, 1998.
- [107] Alberto Riva. Implementation and usage of the CL-DNA library, 2002. Presented at the 2002 International Lisp Conference, <http://www.international-lisp-conference.org/2002/>. The CL-DNA code and documentation are available at Alberto Riva’s web site, <http://www.chip.org/~alb/lisp.html>.
- [108] Alberto Riva and Marco Ramoni. LispWeb: a specialized HTTP server for distributed AI applications. In *Proceedings of the Fifth International World Wide Web Conference*, 1996. May 6-10, 1996, Paris, France.
- [109] K. T. Robbins, J. E. Medina, G. T. Wolfe, et al. Standardizing neck dissection terminology. *Arch Otolaryngology Head and Neck Surgery*, 117:601–605, 1991. Official report of the Academy’s committee for head and neck surgery and oncology.
- [110] Wilhelm Conrad Röntgen. On a new kind of rays (preliminary communication). *British Journal of Radiology*, 4:32, 1931. Translation of a paper read before the Physikalische-medicinischen Gesellschaft of Würzburg on December 28, 1895.
- [111] Cornelius Rosse, Anand Kumar, José L. V. Mejino, Jr., Daniel L. Cook, Landon T. Detweiler, and Barry Smith. A strategy for improving and integrating biomedical ontologies. In *Proceedings of the American Medical Informatics Association (AMIA) Fall Symposium*, pages 639–643, October 2005.

- [112] Cornelius Rosse, Linda G. Shapiro, and James F. Brinkley. The digital anatomist foundational model: Principles for defining and structuring its concept domain. In *Proceedings of the American Medical Informatics Association (AMIA) Fall Symposium*, pages 820–824, November 1998.
- [113] D. L. Rubin, F. Shafa, D. E. Oliver, M. Hewett, and R. B. Altman. Representing genetic sequence data for pharmacogenomics: an evolutionary approach using ontological and relational models. In *Proceedings of the 10th International Conference on Intelligent Systems for Molecular Biology*, pages 207–215. Oxford University Press, 2002.
- [114] Philip Rubin, editor. *Clinical Oncology for Medical Students and Physicians, a Multidisciplinary Approach*. American Cancer Society, sixth edition, 1983.
- [115] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.
- [116] Robert W. Scheifler et al. CLX: Common LISP X interface. Technical report, Texas Instruments, Inc., 1989.
- [117] J. P. Shah, F. C. Candela, and A. K. Poddar. The patterns of cervical node metastases from squamous carcinoma of the oral cavity. *Cancer*, 66:109–113, 1990.
- [118] Y. Shahar, S. Miksch, and P. Johnson. The asgaard project: A task-specific framework for the application and critiquing of time-oriented clinical guidelines. *Artificial Intelligence in Medicine*, 14:29–51, 1998.
- [119] Claude E. Shannon and Warren Weaver. *The Mathematical Theory of Communication*. University of Illinois Press, paperback edition, 1963.
- [120] Edward H. Shortliffe and James Cimino, editors. *Biomedical Informatics: Computer Applications in Health Care and Biomedicine*. Springer-Verlag, third edition, 2006.
- [121] Edward H. Shortliffe et al. ONCOCIN: an expert system for oncology protocol management. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 876–881, Los Altos, California, 1981. William Kaufmann, Inc.
- [122] Edward H. Shortliffe, Leslie E. Perreault, Lawrence M. Fagan, and Gio Wiederhold, editors. *Medical Informatics: Computer Applications in Health Care and Biomedicine*. Springer-Verlag, second edition, 2001.
- [123] Jeff Shrager. BioLingua. <http://nostoc.stanford.edu/Docs/intro.html>.
- [124] Ida Sim, Ben Olasov, and Simona Carini. An ontology of randomized controlled trials for evidence-based practice: content specification and evaluation using the competency decomposition method. *Journal of Biomedical Informatics*, 37(2):108–119, April 2004.



- [125] P. M. Som, H. D. Curtin, and A. Mancuso. An imaging-based classification for the cervical nodes designed as an adjunct to recent clinical based nodal classification. *Arch Otolaryngol Head Neck Surg*, 125:388–396, April 1999.
- [126] A. Spokoyny and Yuval Shahar. A knowledge-based time-oriented active database approach for intelligent abstraction, querying and continuous monitoring of clinical data. In *Proceedings of the 2004 World Congress on Medical Informatics (MEDINFO)*, pages 84–88, 2004.
- [127] Guy Steele, Jr. *COMMON LISP, the Language*. Digital Press, Burlington, Massachusetts, second edition, 1990.
- [128] Jess Stein, editor. *The Random House Dictionary of the English Language*. Random House, Inc., New York, NY, unabridged edition, 1971.
- [129] Christian J. Stoeckert, Jr., Fidel Salas, Brian Brunk, and G. Christian Overton. EpoDB: a prototype database for the analysis of genes expressed during vertebrate erythropoiesis. *Nucleic Acids Research*, 27(1):200–203, 1999.
- [130] Kevin J. Sullivan, Ira J. Kalet, and David Notkin. Evaluating the mediator method: Prism as a case study. *IEEE Transactions on Software Engineering*, 22(8):563–579, August 1996.
- [131] Patrick Suppes. *Introduction to Logic*. Van Nostrand Reinhold Company, 1957. Republished as a Dover book, 1999.
- [132] Gerald Jay Sussman and Jack Wisdom with Meinhard E. Mayer. *Structure and Interpretation of Classical Mechanics*. MIT Press, 2001.
- [133] S. Tanimoto. *The Elements of Artificial Intelligence Using Common Lisp*. W. H. Freeman and Co., second edition, 1995.
- [134] Gregg Tracton, Edward Chaney, Julian Rosenman, and Stephen Pizer. MASK: combining 2D and 3D segmentation methods to enhance functionality. In *Mathematical Methods in Medical Imaging III: Proceedings of 1994 International Symposium on Optics, Imaging, and Instrumentation*, volume 2299. SPIE, July 1994.
- [135] Jan H. van Bommel and Mark A. Musen. *Handbook of medical informatics*. Springer Verlag, 1997.
- [136] Dongwen Wang, Mor Peleg, Samson W. Tu, Aziz A. Boxwala, Omolola Ogunyemi, Qing Zeng, Robert A. Greenes, Vimla L. Patel, and Edward H. Shortliffe. Design and implementation of the GLIF3 guideline execution engine. *Journal of Biomedical Informatics*, 37:305–318, 2004.
- [137] Larry Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer, 2004.
- [138] David A. Watt. *Programming Language Syntax and Semantics*. Prentice Hall International, 1991.

- [139] Steve Webb. *Intensity-Modulated Radiation Therapy*. Institute of Physics Publishing, Bristol, United Kingdom, 2001.
- [140] Patrick Henry Winston and Berthold Paul Klaus Horn. *LISP*. Addison-Wesley, third edition, 1989.
- [141] S. T. Wong, H. K. Huang, R. B. Goldstein, R. Arenson, J. C. Cruz, K. Ostrow, and E. Gerson. A paradigm for computer-assisted radiology resident rotation scheduling. *Academic Radiology*, 2(6):533–540, June 1995.
- [142] B. Zupan, I. Bratko, J. Demsar, P. Juvan, T. Curk, U. Borstnik, J. R. Beck, J. Halter, A. Kuspa, and G. Shaulsky. GenePath: a system for inference of genetic networks and proposal of genetic experiments. *Artificial Intelligence in Medicine*, 29:107–130, 2003.