The DARWIN Manual ©

G. Gonnet       M. Hallett

22 July 2000
The Evolution of Darwin

Darwin is an easy to use interpreted computer language especially tailored to research in the biosciences. Its purpose is to serve as a biochemists’ workbench where researchers can explore molecular sequence data quickly and easily. The Darwin project began in 1991 and reflects much of the research done in the CBRG (Computational Biochemistry Research Group) at the ETH-Zurich. Darwin itself shares much in common with the language MAPLE which is designed for symbolic computation. Broadly speaking, it consists of two parts: the libraries and the kernel.

The libraries correspond closely to what one expects from a software package: a pre-defined set of functions offered by the system. The libraries reflect current and past trends in our research efforts but also incorporate many algorithms from the literature, particularly those related to sequence comparison, phylogenetic tree construction, multiple sequence alignment and secondary structure prediction. The libraries themselves are written in the Darwin language and are therefore easy to read even for novice programmers. We briefly describe the current contents of the libraries in the next section.

The kernel of Darwin is responsible for the lower-level operations in the system: executing commands and libraries, memory management, input/output, communication with the operating system, load balancing, etc. Darwin itself is written in C although this source code is not made publicly available. The kernel also contains critical routines, that is, routines which must be performed efficiently due to the number of times they are called or the complexity of the routines themselves. These include (but are not limited too) routines for pairwise alignment, all versus all alignments, and tree construction. Although the kernel is not modifiable, one can execute native code (that is, user designed code written in C, JAVA, etc.) from within Darwin.

Since Darwin is a computer language, it allows one to go beyond the fixed set of routines offered in the kernel and library. The language itself is a high-level interpreted language equipped with lists, sets, general data structures, and a robust collection of basic mathematical functions allowing the user to quickly prototype new ideas. Darwin programs are relatively fast even when compared to optimized C code. A moderately experienced
user of the system will be able to modify existing libraries (written in the Darwin language) when necessary or create new libraries appropriate to whichever problem they are currently exploring. Below is a small example of a Darwin session:

```
unix: darwin
   Darwin: Sequence Searching Facility
Version 2.0, August 1998
   (c) E.T.H. Zurich
DB:=ReadDb('SwissProt38');
Peptide file(SwissProt38(547/14687),
77977 entries, 28268293 aminoacids)
printf("\nIdentification: \%s',
     Entry(1)['ID']);
Identification: 100K_RAT
printf( 'Accession Number: \%s',
     Entry(1)['AC'] );
Accession Number: Q62671;
CreateDayMatrices():
res := AlignOneAll( 1, DB, DM, 120 );
length( res );
19
hisofar := 0: index := 0:
for i from 2 to length( res ) do
   if (res[i,Sim] hisofar) then
      hisofar := res[i,Sim]:
      index := i:
   fi:
   od:
printf( 'Most similar is \%d with
        score \%5.2f', index, hisofar );
Most similar is 11 with score 301.13
```

The above program loads the SwissProt v. 38 dataset (ReadDb), then prints out the identification and accession tags for the first entry. After creating the GCB extended Dayhoff matrices, the first entry is compared against all other entries in SwissProt (AlignOneAll). In this example, the alignment is performed at a PAM distance of 250 (variable DM) and all significant matches are stored in the variable res. A significant match here is
defined as any match with a similarity score greater than or equal to 120.\footnote{This is a maximum likelihood log-odds score which can be interpreted as meaning that it is $10^{120}$ more likely the sequences evolved from a common ancestor than a random alignment.} We search through the 19 such matches for the alignment which induced the highest similarity score.

Although large, the libraries distributed with Darwin are far from complete (computational biology travels simply too fast to make “keeping up” viable). Users are invited to submit new libraries for inclusion in future releases of the system.

Combining algorithms both from the literature and research local to the CBRG, our system allows a flexibility that no previous system has offered. This flexibility is an absolute necessity as we enter an age where the analysis of complete genomes will be commonplace. We believe that the power of Darwin remains largely untapped although over 400 research groups have experimented with our software.

**Basic Mathematical Operations** The system includes operations for sets, lists, trigometric functions, combinatorial graphs, long integers, real and complex numbers, likelihood/probability calculations, matrices (including LLL decompositions, Gaussian eliminations, Givens eliminations, singular value decompositions, eigenvalue/eigenvector computation, Gram-Schmidt decompositions and linear regressions), control of input/output, and interacting with the operating system.

**Pairwise Alignment** Darwin comes equipped with routines to align peptide sequences versus peptide sequences, or nucleotide versus peptide sequences \cite{alignment}. The alignment routines are based on the full dynamic programming approach using the GCB matrices. See \cite{alignment}. The system can also perform parametric alignments which seek to find the PAM distance which maximizes similarity score. Local alignments, global alignments, and cost-free “end gap” alignments are all possible.
Dataset Conversions The system includes routines for converting raw Swiss-Prot \[?] or EMBL \[?] flat-files to the Darwin format. The libraries also include a parser shell which can be easily modified to parse any flat-file.

All versus All Routines One of the most useful features of Darwin is its ability to perform large scale comparisons of genomes; that is, the alignment of every sequence in a dataset with every other sequence in the dataset. To this end, Darwin automatically generates a patricia tree\(^2\) when a sequence dataset is loaded and provides various built-in (i.e. located in the kernel) functions for performing fast alignments. Also, Darwin is capable of distributing a large set of jobs over an intra-net, can control the computation of these jobs on the foreign machines, and collect the results.

Peptide Transition Matrices The following peptide transition matrices are built into Darwin: Dayhoff, GCB \[?\], BLOSUM \[?\], amongst others. New matrices can be computed from sample data. There are also routines for converting PAM distance to and from percent identity.

Protein Identification via Peptide Mass Darwin contains routines for protein identification by aligning the masses of small collections of peptides after N- or C- terminal digestion against either a nucleotide and peptide dataset \[?\].

Phylogenetic Tree and Multiple Sequence Alignment Construction Historically, tree construction in Darwin has been based on distance matrices and the system contains various related routines: tree topology construction algorithms (Neighbour joining \[?\], clustering methods \[?\], amongst others), least squares fits to tree topologies, and local optimization routines. There are now routines for tree construction based on circular orders, a new method developed in \[?\]. Multiple sequence alignments \[?\] are created relative to a phylogenetic tree and the system includes several methods for scoring the quality of the alignment including a novel method developed in \[?\].

Statistics and Visualization This system includes routines for drawing histograms, dot plots, and bar graphs. The system can also draw unrooted trees, rooted trees, split trees, and combinatorial graphs. There are

\(^2\)A patricia tree is a close sibling to the suffix tree, the more common data-structure in the literature.
a large number of routines for producing random permutations, combinations, distributions and specific biological objects such as sequences, trees, and multiple sequence alignments.

This manual describes the Darwin language and all of the basic functionality of the language including the basic commands, constructors, data types, built-in data structures, and descriptors for all library functions. Darwin v. 1 suffered from a somewhat scattered and nonintuitive naming scheme for its predefined functions. In order to make Darwin more usable, we have adopted a standardized naming convention. Furthermore, a substantial subset of the manual is available via on-line help from within Darwin and the remainder is available via the WWW [5]. Lastly, a large number of bugs reported by our user base have been fixed.

This book is divided into three parts. Part I – An Introduction to Darwin has been designed to familiarize even the most computer illiterate amongst us with the basic Darwin environment. We have to tried to write to the biologist/biochemist who perhaps has had a first year Introduction to Computer Science course and who has distant memories of for-loops and recursion stored somewhere deep in the recess of their subconscious. An attempt has been made to use simple terminology, only giving those definitions we deem important in later chapters. For new users, we recommend that Part I be read “in one sitting” beginning at Chapter 1 – Exploring the Basics and following the discussion and examples through to the end of Chapter 12 – A Guide to Debugging. An experienced programmer may find that Part I need only be skimmed in order to familiarize themself with the peculiarities of the Darwin system. Once comfortable with the language, users may find that it acts as a short reference guide for looking up commands “on the fly”. Towards this end, we have attempted to make each chapter self-contained.

Chapter 1 – Exploring the Basics provides a basic session with Darwin designed to give new users a feeling of how to interact with the system. Chapter 2 through to Chapter 7 provides a more in depth tour of the basic Darwin language with a focus on the most commonly used commands and routines built into the kernel. We recommend new users become familiar with the topics covered in these chapters before venturing into Part II.

---

3The naming convention is included in the manual.
vi

The more esoteric and application specific routines are discussed in Chapter 8 – Genetic Databases, Chapter 9 – Randomization, Statistics and Visualization, Chapter 14 – Producing HTML Code, Chapter 15 – Darwin’s Interprocessor Skills, and Chapter 16 – Calling External Functions.

Chapter 8 – Genetic Databases provides an in depth look at how Darwin builds, stores and manipulates genetic databases. In some sense, these data structures are the cornerstone of the system and a fluency with their manipulation will greatly ease the difficulty of programming in Darwin. Chapter 9 – Randomization, Statistics and Visualization contains an overview of the randomization and statistics functions followed with an explanation of the basic primitives available for graphing and plotting information. These are used extensively throughout later chapters, most notably Chapter 18 – Dayhoff Matrices and Mutation Matrices, Chapter 19 – Coping with Insertions and Deletions, Chapter ?? – Generating Random Sequences, and Chapter 24 – Phylogenetic Trees.

An indepth reading of Chapter 10 – Overloading, Polymorphism and Object Orientation, Chapter 13 – Measuring Performance and Chapter 14 – Producing HTML Code should be postponed until the reader feels he/she is particularly comfortable with the system.

Chapter 15 Darwin’s Interprocessor Skills explains the mechanisms built into Darwin for interprocessor communication. These routines allow users to fragment large computationally intensive jobs into smaller pieces which can be distributed automatically to other processors. A complete understanding of this topic is not necessary for one to proceed into later chapters with the exception of the latter half of Chapter ?? – All against All where a program is given which performs an exhaustive matching of a set of amino acid sequences.

Each chapter in Part II Darwin and Problems from Biochemistry examines a different bioinformatic problem. Every chapter contains (1) a statement of the problem, (2) a discussion concerning any biologic assumptions we make about the data, (3) an explanation of how we model the problem mathematically, (4) a description of the algorithm, (5) a Darwin implementation, (6) a discussion about the accuracy and efficiency of our algorithm, and (7) a short guide to the literature. In this manner, we tour the Darwin libraries motivating each routine and data structure with a concrete example. Be-
beyond the understanding of the Darwin libraries, we hope such a presentation gives users

- an understanding of some of the classic problems from bioinformatics,
- an understanding of the underlying biochemistry involved in these problems,
- an understanding of the mathematical model upon which these algorithms are predicated,
- an understanding of how the algorithms works, and
- a conceptual overview of how to structure programs in Darwin.

Part ?? – The Reference Guide provides a complete list of the Darwin built-in functions and libraries. It is structured into sections containing routines related by function (e.g. Mathematical Functions, Input/Output Functions, string Searching Functions, and so forth).

The appendices contain some general material including a short introduction to statistics and dynamic programming. For those readers unfamiliar with the mathematics underlying the models we use, these chapters will provide a deeper understanding of our methods.

All of the examples and programs used throughout this manual are available via the world wide web (WWW) or by ftp (file transfer protocol). The Computational Biochemistry Research Group at ETH–Zürich maintains a web cite at:

http://cbrg.inf.ethz.ch/

The home page located at this address contains links to this manual and the example code files. If you do not have access to a web browsers, the ftp address

    ftp inf.ethz.ch
    user: anonymous

also mirrors these files.
0.1 Availability and Contact

Darwin is available free of charge from our WWW server at inf.ethz.ch or by sending e-mail to darwin@inf.ethz.ch. Interested users are asked to fill out a short form indicating which platform(s)\textsuperscript{4} are desired. The system will be e-mailed shortly after we receive your signed document.

Our group at ETH-Zürich and the University of Florida at Gainsville continue to add code to the Darwin system and we regularly make this new code available via the above web site. Readers are encouraged to submit their code into our algorithms repository. If you feel you have a particularly useful, novel or simply better algorithm for a problem, please send us e-mail at the address below.

\textsuperscript{4} DARWIN is available on the following platforms: DEC Alpha/Digital Unix 4.0, SGI Irix 6.x, Sun Sparc Solaris 2.5 and up, HP-UX 10.x, Linux 2.x, Windows '95, Windows NT.
Contents

0.1 Availability and Contact ................................ viii

List of Tables ...................................................... xviii
List of Figures ..................................................... xxi

The Evolution of Darwin ............................................ 1

I An Introduction to Darwin ....................................... 3
   1.1 Initiating a Darwin Session ................................. 3
   1.2 Basic Terminology and Arithmetic Operators ............. 4
   1.3 Comments ................................................... 9
   1.4 Variables .................................................. 9
   1.5 Simple types .............................................. 11
      1.5.1 Sets ................................................ 13
      1.5.2 Lists and Arrays .................................... 16
      1.5.3 Strings ............................................. 21
      1.5.4 Built-in Functions for Sets, Arrays and string .... 22
   1.6 Simple Output ............................................. 24
   1.7 Boolean Expressions ...................................... 25
   1.8 Decisions ................................................ 26
   1.9 Looping Around ......................................... 29
   1.10 Interrupting Computation ................................ 30
   1.11 Ending a Darwin Session ................................. 31
1.12 Navigating Around the System .................................. 31

2 Adv. String Manipulation ............................................ 33
  2.1 Printing .......................................................... 33
     2.1.1 printf ...................................................... 33
     2.1.2 sprintf ................................................... 37
  2.2 Exact string Search ............................................. 37
     2.2.1 String Format Scan – scanf ................................ 37
     2.2.2 Searching for Strings ...................................... 38
     2.2.3 Searching with SGML Tags .................................. 38
  2.3 Approximate string Search ...................................... 39
     2.3.1 ApproxSearchString ........................................ 40
     2.3.2 BestSearchString ........................................... 40

3 Procedures ................................................................. 41
  3.1 The proc Command ................................................ 42
  3.2 Commenting a Procedure: description ............................ 45
  3.3 Scoping Rules ...................................................... 46
     3.3.1 Independent Scopes ......................................... 48
     3.3.2 Examining Global Variables from within Routines .......... 49
     3.3.3 Modifying Global Variables .................................. 51
     3.3.4 Modifying the Value of a Parameter ......................... 52
  3.4 Functions ........................................................... 53
     3.4.1 The return Command .......................................... 54
  3.5 Nested Routines .................................................... 57
  3.6 Variable Number of Arguments ................................... 58

4 Lists and Arrays Revisited ......................................... 61
  4.1 Zippable Commands ............................................... 61
  4.2 Sorting ............................................................. 61
  4.3 Searching .......................................................... 64

5 Structured Types .......................................................... 67
  5.1 Defining New Structured Types: An Example .................... 70
     5.1.1 Error Checking ............................................... 73
     5.1.2 Simplification ............................................... 75
CONTENTS

5.1.3 Normalization .................................... 77
5.2 Nested Types ...................................... 77
5.3 Selection on Data Structures ......................... 78
5.4 The Types of Structured Types ..................... 81

6 Iteration and Recursion .............................. 85
6.1 Iteration ........................................... 85
   6.1.1 The for Command ............................... 86
   6.1.2 The while Statement ............................ 88
6.2 Recursive Calls ................................... 90

7 Input/Output ........................................ 97
7.1 Standard Files .................................... 98
   7.1.1 The ReadProgram Command .................... 98
   7.1.2 The ReadLibrary command ..................... 99
   7.1.3 The save Command ............................. 100
7.2 Raw Data Files .................................... 101
   7.2.1 Reading Raw Data Files ....................... 102
   7.2.2 Creating Raw Data Files ....................... 104
7.3 Initializing Your Darwin Session ................... 105

8 Genetic Databases .................................. 107
8.1 Sequence Databases ................................ 107
8.2 Darwin Sequence Databases ......................... 108
8.3 Building a Darwin Sequence Database ............... 113
   8.3.1 Map Files and Patricia Trees .................. 119
8.4 Accessing a Darwin Sequence Database .............. 122
   8.4.1 The Entry Structure ............................ 124
   8.4.2 The Offset Structure ........................... 127
   8.4.3 The Sequence Structure ......................... 130
   8.4.4 The String Function ............................ 132
   8.4.5 The SearchDb Function ......................... 133
8.5 Indexing a Darwin Sequence Database ............... 134
8.6 References ....................................... 136
## CONTENTS

9 Randomization, Statistics and Visualization  
9.1 Randomization ........................................... 138  
9.2 Basic Visualization Functions  
  9.2.1 Set(plotoutput)  
  9.2.2 Set(plotdevice)  
  9.2.3 Creating Plots  
  9.2.4 Creating Dot Plots  
  9.2.5 Creating Histograms  
  9.2.6 Drawing Trees  
  9.2.7 Annotating Plots and Graphs  
9.3 Statistics  
  9.3.1 Calculating an Exponential Fit  
  9.3.2 Linear Regressions  
  9.3.3 Single Value Decompositions  
  9.3.4 The stats command  
  9.3.5 The UpdateStat command  
  9.3.6 Generating Distributions  
  9.3.7 Smoothing Data Points  

10 Polymorphism  
  10.1 Object-relational Databases  
  10.2 Using the Database Built into Darwin  
  10.3 Using the Opera Facilities in Darwin  

11 System Commands  
  11.1 Communicating with the Operating System  
    11.1.1 Date and Time  
    11.1.2 Operating System Commands  
    11.1.3 Garbage Collection  
  11.2 Internal Variables  
  11.3 Options  

12 A Guide to Debugging  
  12.1 Tracing and the Print Level  
    12.1.1 printlevel  
    12.1.2 trace  

134  

159  

158  

158  

159  

160  

161  

162  

163  

165
12.2 Trapping Errors ........................................... 168

13 Measuring Performance ........................................... 171
  13.1 option numeric ........................................... 171
  13.2 A Small Example ........................................... 172

14 Producing HTML Code ........................................... 173

15 Darwin’s Interprocessor Skills ........................................... 175

16 Calling External Functions ........................................... 177

II Darwin and Problems from Biochemistry ........................................... 179

17 Modelling Evolution ........................................... 185

18 Point Accepted Mutations and Dayhoff Matrices ......................... 187
  18.1 Modeling Evolution ........................................... 189
    18.1.1 Point Accepted Mutations ........................................... 190
    18.1.2 Dayhoff Matrices ........................................... 191
    18.1.3 Interpreting Scores ........................................... 192
  18.2 The Original Dayhoff Matrices ........................................... 193
  18.3 Better Dayhoff Matrices ........................................... 197
    18.3.1 The CreateDayMatrices Function ......................... 198
    18.3.2 The CreateDayMatrix Function ......................... 203
    18.3.3 Percent Identity and PAM Distance ......................... 206
  18.4 Estimating Mutation Matrices ........................................... 211
  18.5 Other Similarity Matrices ........................................... 217

19 Gap Penalties ........................................... 219
  19.1 Gap Probabilities and Penalties ........................................... 221

20 A General Introduction to Dynamic Programming ......................... 225
  20.1 Longest Common Subsequence ........................................... 226
  20.2 Dynamic Programming with a Weighted Scoring Matrix ......................... 226
  20.3 No Cost for End Gap Alignment ........................................... 226
  20.4 Local Alignment ........................................... 226
20.5 Affine Gap Penalties .............................. 226
20.6 Convex Gap Penalties .............................. 226
20.7 General Gap Penalties .............................. 226
20.8 Aligning DNA versus DNA ......................... 226

21 A Maximum Likelihood Pairwise Alignment of Amino Acid
Sequences ................................. 227
  21.1 Dynamic Programming .......................... 228
      21.1.1 The DynProg Function .................. 229
      21.1.2 The BackDynProg Function .......... 231
      21.1.3 Global Alignments .................... 232
  21.2 The Match Structure ......................... 233
  21.3 Matching Routines ......................... 234
      21.3.1 The GlobalAlign Function .......... 235
      21.3.2 The LocalAlign Function ........ 238
  21.4 The LogDelLocalRefine Function .......... 240
  21.5 Estimating PAM Distances .................. 241
  21.6 The FindBestPam Function ............... 243
  21.7 The LocalAlignBestPam Function ....... 244
  21.8 The DrawSimPam Function ............... 245
  21.9 The GetOffset Function ............... 247

22 Searching for Genes ............................... 249
  22.1 Searching for hypothetical peptides .... 251
  22.2 Direct nucleotide versus peptide alignment 258

23 Datastructures for Genetic Datasets ............ 269

24 Phylogenetic Trees ................................. 271

25 Phylogenetic Trees ................................. 273
  25.1 Introduction .................................. 273
  25.2 Scoring Functions ......................... 273
      25.2.1 Parsimony Based Methods .......... 273
      25.2.2 Maximum Likelihood Methods .... 274
      25.2.3 Distance Based Methods .......... 274
25.3 Searching for Tree Topologies .......................... 274
   25.3.1 Tree Enumeration .............................. 274
   25.3.2 Initial Topologies ............................. 274
   25.3.3 Local Search Techniques ...................... 274

26 Multiple Sequence Alignments ............................. 275
   26.1 Scoring a Multiple Sequence Alignment .............. 276
      26.1.1 Sum of Pairs .................................. 276
      26.1.2 Travelling Salesman Tours .................... 276
      26.1.3 Other Methods ............................... 276
   26.2 Constructing a Multiple Sequence Alignment .......... 276
      26.2.1 Multi-dimensional Dynamic Programming ....... 276
      26.2.2 Bootstraping Techniques ..................... 276
      26.2.3 Building a Multiple Sequence Alignment Using a Phy-
           logenetic Tree ....................................... 276
      26.2.4 The Darwin Technique .......................... 276
      26.2.5 The New Darwin Technique ..................... 276
   26.3 Gap Adjustment ..................................... 276

27 Probabilistic Ancestral Sequences .......................... 279
   27.1 Algorithms for Probabilistic Sequence Creation ....... 280
   27.2 Comparing Two Probabilistic Sequences ............... 280
   27.3 Consensus Sequence Algorithms ....................... 280

28 Recognizing Proteins by Weight ............................. 281
   28.1 The comparison algorithm ........................... 283
   28.2 Probability foundations ............................ 285
   28.3 Example and probability estimates ................... 291

III The Appendices ............................................. 295

A Installation Instructions ................................... 297

B Naming Conventions ......................................... 299
   B.1 Built-in Types ...................................... 301
   B.2 Structured Types .................................... 302
CONTENTS

B.3 Commands .................................. 303
B.4 Built-In Functions .............................. 303
B.5 Library Functions ................................. 304
B.6 Name Conversion List ............................ 307
B.7 Scripts for Updating Code .......................... 317
List of Tables

1.1 Some arithmetic operations offered by Darwin. .............. 6
1.2 The order of operations in Darwin. ......................... 8
1.3 Some basic types in Darwin. .............................. 12
1.4 List of all comparison and boolean operators. .............. 26

2.1 Conversion characters for the printf command. .......... 35
2.2 The cursor control sequences for the printf command. ... 36

5.1 A complete list of built-in Darwin types. .................. 83
5.2 A list of built-in Darwin structured types. ................. 84

8.1 A table showing the different types of Darwin sequence databases and the legal bases for each. ............... 111
8.2 The selectors for objects of type database. ............... 123

9.1 The shape parameters for the DrawPlot function. .......... 146

11.1 Options for the Set function. ......................... 162
11.2 The option command. ................................. 164

18.1 The selectors for the DayMatrix structured type. .......... 195

21.1 Selectors for the Match structured type. ................. 234
22.1 Selectors for the NucPepMatch structured type. ........... 250
26.1 Selectors for the MultiAlign structured type. ............. 277

A.1 The flags for executing Darwin. .......................... 298
xviii

LIST OF TABLES
# List of Figures

1.1 The relationships between the different number types. ........ 14  
1.2 Viewing the list as a chain. ............................. 17  
1.3 A one dimensional **array**. ............................... 18  
1.4 An array of type **matrix**. ............................... 20  
1.5 A three dimensional **array**. ............................... 21  

6.1 A rooted phylogenetic tree formed from the SH2 sequences  
for six species. ........................................... 93  

8.1 A sample Swiss-Prot entry. ............................... 109  
8.2 A sample EMBL entry. ................................. 110  
8.3 The parser for a Swiss-Prot file. ....................... 115  
8.4 A diagram showing the **database** structure **DB**. .......... 128  

9.1 A plot of points in the Cartesian plane. .................. 143  
9.2 A graph of the function \( \sin(x) \) for range \(-5..5\). ...... 145  
9.3 A comparison of two distributions. ....................... 145  
9.4 A figure showing 801 overlapping circles drawn along the sine  
curve. ...................................................... 147  
9.5 Amino acid frequencies in the **Sample/SH2** database. ...... 148  
9.6 An example of a rooted tree in Darwin. ................... 150  
9.7 An example of a unrooted tree in Darwin. .................. 152  

16.1 Darwin’s Tower of Information ........................... 182  

21.1 The **DrawSimPam** Function for the **Match m**. ......... 246
Part I

An Introduction to Darwin
Chapter 1

Exploring the Basics

The aim of this chapter is to help new users initiate a Darwin session, to teach the etiquette of basic conversations with the Darwin kernel, and to provide a general overview of where and how to find help for any questions that may arise during your experimentation.

1.1 Initiating a Darwin Session

Whether you received your Darwin package by electronic mail, ftp, or surface mail, detailed instructions for either you or your system administrator are included in this package for decompressing and installing the necessary files. Appendix A and the Computational Biochemistry Research Group WWW site (see side box below) provide additional information if you should encounter difficulties.

Different operating systems will require different commands for starting Darwin. In the UNIX operating system [26], users would typically be required to type

```
% darwin
```

at the shell prompt. The Darwin prompt is by symbol (>) and the cursor is
placed immediately to the right of it. We are now ready to begin interacting with Darwin.

The Computational Biochemistry Research Group at ETH–Zürich maintains a web site at \url{http://cbrg.inf.ethz.ch/} for the Darwin system and The Darwin Manual (this document). This site contains many sample sessions designed to help you become familiar with the different aspects of the system and provides a regularly updated information page about recent developments. Most major Internet browsers (Explorer, Mosaic, Netscape, Linex and others) are supported.

### 1.2 Basic Terminology and Arithmetic Operators

We start by issuing commands to Darwin which compute simple arithmetic operations. The basic data objects in all programs are symbols and constants. A constant in Darwin is any number. For example,

\[
1.1, 5, -999, 88382932
\]

A symbol is a letter (a…z, A…Z) or an underscore symbol (_). For example,

\[
a, b, m, M, Z, _
\]

An operator specifies what should be done to a set of constants and symbols. The following are all operators in Darwin,

\[
+, -, *, /, =, <, >, <=, >=
\]

Together, symbols and constants written in the correct syntax form expressions. If you type the simple expression,

\[
> 1+1
\]

followed by the return key\(^1\), Darwin responds by supplying a fresh prompt on the line below and then waits for more instructions. Either the user must continue extending this expression or they must enter a semicolon followed by the return key.

---

\(^1\) Depending on the type of system you are using, this may be labelled the enter key.
The expression `1 + 1` combined with the semicolon (`;`) is our first example of a *statement* in Darwin. All Darwin statements are built from commands, expressions and either a terminating semicolon (`;`) or colon (`:`). When Darwin receives the statement, it *evaluates* the expression and echos the result to the terminal. If we had instead chosen to terminate our expression with a colon, Darwin would evaluate the expression but suppresses the (surprising) answer.

```plaintext
> 1 + 1;
2
```

The utility of the colon will be made clearer in later sections when we begin to write routines and loops. The colon allows us to control which statements are echoed to the screen and which are executed silently.

There are a wide range of operators we can use to form expressions in Darwin. Table 1.1 lists some of the more commonly used ones.

```plaintext
> 1 * 2 * 3 * 4;
24
> 4 !;
24
> 2^5 + 2^6;
96
> round(4.9999) * round(4.9999);
25
```

Any number of *return* or *space* keystrokes may be entered by the user before a semicolon (or colon) is entered. These *white space characters*, as they are called, do not affect the calculation in any way. For example,

```plaintext
> 8 + 3
>   -5
>   * 8 / 2
> 
```
<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>+</td>
<td>101 + 27</td>
<td>128</td>
</tr>
<tr>
<td>subtraction</td>
<td>–</td>
<td>15 – 3</td>
<td>12</td>
</tr>
<tr>
<td>multiplication</td>
<td>*</td>
<td>5 * 3</td>
<td>15</td>
</tr>
<tr>
<td>division</td>
<td>/</td>
<td>6/2</td>
<td>3</td>
</tr>
<tr>
<td>exponentiation</td>
<td>^ or **</td>
<td>2^8</td>
<td>256</td>
</tr>
<tr>
<td>natural exponentiation</td>
<td>$exp(\cdot)$</td>
<td>$exp(3)$</td>
<td>20.0855</td>
</tr>
<tr>
<td>absolute value</td>
<td>$abs(\cdot)$</td>
<td>$abs(-6.7)$</td>
<td>6.7</td>
</tr>
<tr>
<td>factorial</td>
<td>!</td>
<td>5!</td>
<td>120</td>
</tr>
<tr>
<td>square root</td>
<td>$sqrt(\cdot)$</td>
<td>$sqrt(16)$</td>
<td>4</td>
</tr>
<tr>
<td>natural logarithm</td>
<td>$log(\cdot)$ or $ln(\cdot)$</td>
<td>$log(1)$</td>
<td>0</td>
</tr>
<tr>
<td>logarithm base 10</td>
<td>$log10(\cdot)$</td>
<td>$log10(100)$</td>
<td>2</td>
</tr>
<tr>
<td>rounding</td>
<td>$round(\cdot)$</td>
<td>$round(4,99999)$</td>
<td>5</td>
</tr>
</tbody>
</table>

Table 1.1: Some arithmetic operations offered by Darwin.

```
> ;
-9
```

produces the same result as

```
> 8+3-5*8/2;
-9
```

With most computers, there is an overall limit on the length of a single line. This is typically is on the order of two or three hundred keystrokes. Entering a statement which exceeds this limit does not pose a problem since the semicolon allows us to split the entry over several lines.

```
> 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 + 12 + 13 +
> 14 + 15 + 16 + 17 + 18 + 19 + 20 + 21 + 22 + 23 + 24 +
> 25 + 26 + 27 + 28 + 29 + 30 + 31 + 32 + 33 + 34 + 35 +
> 36 + 37 + 38 + 39 + 40 + 41 + 42 + 43 + 44 + 45 + 46 +
> 47 + 48 + 49 + 50;
1275
```

Of course, in this case the true mathematicians amongst us would have opted for the following more elegant and fewer keystroke solution anyway.
1.2. BASIC TERMINOLOGY AND ARITHMETIC OPERATORS

\[ (50*(50+1))/2; \]
1275

Beware, it is easy to make errors when splitting entries over several lines. If you enter

\[ > 8 + 3 \]
\[ > 2 + 5; \]
syntax error:
\[ 2 + 5; \]

then you will have officially experienced your first syntax error in Darwin since there is neither a semicolon, colon nor operator between the 3 and the 2. When any such error occurs, Darwin responds with a brief message consisting of your input and a caret symbol (^) at the point in your statement where it first became confused. It then gives you a fresh prompt and ignores your previous entry.

When performing a sequence of statements, making use of the double quote symbol ("" ) will sometimes save you keystrokes. A single double quote symbol refers to the result of the last statement. Two double quote symbols (""") refer to the result of the second last statement and the result preceding this is referred to by ("""). The example below shows how these can be used to generate the elements of the famous Fibonacci sequence 1, 1, 2, 3, 5, 8, 13, . . . .

\[ > 1; \]
1
\[ > 1; \]
1
\[ > " + ""; \]
2
\[ > " + ""; \]
3
\[ > " + ""; \]
> " + ";

When submitting longer arithmetic expressions, users must remember to respect the order of operations. For instance, the expression

\[ 5 \times 8 \times 3 \div 2; \]

does not evaluate to 19.5 but to 17. This is because multiplication and division have higher precedence than addition and subtraction. If we first wanted the addition of 5 and 8 performed before the multiplication and division, we would rewrite the expression using parenthesis as

\[ (5 + 8) \times 3 \div 2; \]

17
\[ 19.5000 \]

Table 1.2 shows the order of operations in Darwin.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>parenthesis</td>
<td>(, )</td>
</tr>
<tr>
<td>exponentiation</td>
<td>^ or **</td>
</tr>
<tr>
<td>negation (unary subtraction)</td>
<td>-</td>
</tr>
<tr>
<td>multiplication/division</td>
<td>*, /</td>
</tr>
<tr>
<td>addition/subtraction</td>
<td>+, -</td>
</tr>
</tbody>
</table>

Table 1.2: Order of operations in Darwin listed from highest to lowest precedence.

Subtle errors can be caused by forgotten parentheses. Beware of the following pitfalls:

\[ -5 \times 2; \]

# Negation has a lower precedence than exponentiation
-25

\[ (-5) \times 2; \]

# The parentheses force the exponentiation to use -5 instead
25

\[ -5 \times 2; \]

# Same thing, but using the other exponentiation operator.
1.3. COMMENTS

-25
> -5 * -2;    # Negative numbers must be parenthesized when following
  syntax error:
> -5 * -2;
  -
> -5 * (-2);    # an operator.
10
> 2 ^ 2 ^ 2;    # Towers of exponents must be parenthesized.
  syntax error:
> 2 ^ 2 ^ 2;
  -
> 2 ^ ( 2 ^ 2 );
16

A complete list of all the mathematical functions built into Darwin is
located in the reference guide under the section §?? Mathematical Functions

1.3 Comments

To help keep programs readable, Darwin allows users to insert comments
into their code. All text to the right of the pound symbol (#) up until the
next return is ignored by the system.

> 2 ^ (2 ^ (2 ^ (2 ^ (2 ^ 2))));    # A comment about the tower of 2s.
  Infinity

1.4 Variables

Variables allow users to assign names to the results of expressions. Suppose
you would like to calculate a long series of arithmetic expressions and some
of the values in these expressions are used more than once. For instance,
instead of the rather laborious typing exercise of
> 5 * 83945773897929834;
> 7 * 83945773897929834;
> 9 * 83945773897929834;

one could instead assign this rather cumbersome number to a variable and use it instead in subsequent multiplications.

> num := 83945773897929834;
> 5 * num;
> 7 * num;
> 9 * num;

We say that the Darwin variable num was *assigned* a value. In the above case, num was assigned a value which has type real. The real type corresponds to numbers which may have a fraction part. In general, variables may have one of several different types.

> year := 1997;  # a variable of type posint
> temperature := -50;  # a variable of type integer
> pie := 3.14;  # a variable of type real
> decision := true;  # a variable of type boolean
> name := 'darwin';  # a variable of type string

Table 1.3 lists some of the basic types in Darwin.

There are rules in Darwin for what constitutes valid variable names. These are as follows:

- Variables must either begin with a letter (a...z, A...Z) or the underscore symbol (\_). Subsequent symbols may be letters, the underscore symbol (\_) or numbers (0...9).

- The length of a variable name must be less than approximately 240 symbols. This bound is system dependent.

- Darwin distinguishes between uppercase and lowercase letters. Thus, variable Result is different from variable result.
1.5. SIMPLE TYPES

- The use of the underscore symbol (_,) to begin variable names is discouraged since the Darwin system uses this convention when creating system variables. If you are unlucky, you may assign a new value to a system variable or the system may assign a new value to your variable. In either case, you probably will not appreciate the resulting havoc.

The following are all legal variable names in Darwin:

> this_is_the_longest_variable_name_in_the_world := 'big';
> R2D2 := true;
> _toying_with_disaster := 'the dangerous underscore';

A word of advice from the computer science community: always use variables names that are meaningful in that they express the function of the variable in the program. This makes your code easier for others to read. (It also makes it easier for you to read when you re-examine your work decades later.)

1.5 Simple types

We touched upon some simple types in Section 1.4 where we saw examples of variables of type real, integer, posint, boolean and string. Types have the following pragmatic advantages:

- they make your programs easier to understand,
- they reduce the number of errors through a mechanism called type checking, and
- they allow one to organize our data in an intuitive natural manner.

All data has a type associated with it and to test the type of an item of data there is a built-in Darwin function

\texttt{type(exp, tn)}
<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>real</td>
<td>Real numbers</td>
<td>-1, -1.01, 99.9, ...</td>
</tr>
<tr>
<td>integer</td>
<td>Integer numbers</td>
<td>..., -1, 0, 1, 2, ...</td>
</tr>
<tr>
<td>posint</td>
<td>Integers greater than zero</td>
<td>1, 2, ...</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean values</td>
<td>true, false</td>
</tr>
<tr>
<td>string</td>
<td>A sequence of symbols surrounded by single quotes</td>
<td>'hello', '12345'</td>
</tr>
<tr>
<td>name</td>
<td>A legal Darwin variable, procedure or structure name</td>
<td>abc, x where x:= proc() end;</td>
</tr>
<tr>
<td>constant</td>
<td>Any real value or variable assigned a value</td>
<td>1, 3.583, -11</td>
</tr>
</tbody>
</table>

Table 1.3: Some basic types in Darwin. See Table 5.1 for a complete list of all built-in types.

where $exp$ is any expression and $tn$ is any Darwin type. This function returns true if the expression $exp$ evaluates to type $tn$ or false otherwise.

For instance,

```plaintext
> type(5, real);          # real numbers are integers and
true
> type(1.3, real);        # numbers with fractional parts
true
> type(5, integer);       # integers positive whole numbers
true
> type(-5, integer);      # and negative whole numbers
false
> type(99, posint);       # no fractions allowed in type integer
false
> type(0, posint);        # positive integers are integers
true
> type(0, posint);        # greater than zero
false
```

# 0 is not a positive integer
false
> type('hello', string);  # strings surrounded with single quotes
true
> type(true, boolean);  # true and false are booleans
true
> type(5.0, integer);  # maybe this one is a bit of a surprise
true

In fact, even variable names have a type associated with them. Any legal
variable name which has not been assigned a value in Darwin has type \texttt{name}.

> type(this_is_the_longest_variable_name_in_the_world, name);
> type(R2D2, name);
> type(_toying_with_disaster, name);

(see § 1.4 for the criteria of what constitutes a legal name).

The type \texttt{boolean} has two values: \texttt{true} and \texttt{false}. The \texttt{real} and \texttt{integer}
types correspond to the mathematical concepts of a real and integer numbers
respectively. The \texttt{posint} type is an abbreviation of \texttt{positive integer}. The
\texttt{posint} type does not include the number zero. The relationships between
these three types is shown in Figure 1.1.

An object of type \texttt{string} is a sequence of symbols surrounded by the
single quote symbols (').

From these simple types, we can build the more complex \textit{structured types}
that we will encounter in later chapters of this book.

1.5.1 Sets

One of the most useful (and most often used) data types in Darwin is the
\texttt{set}. The Darwin type \texttt{set} closely mirrors the mathematical concept: a set
is either empty or has a finite number of distinct unordered items. You can
think of a set as a special bag to keep objects. This bag has the (magic)
ability to discard multiple copies of the same object. When objects are in
the bag, there is no order placed upon them.
Figure 1.1: The relationships between the different number types. Every object of type integer also has type real. Every object of type posint also has type integer (and hence type real).
1.5. SIMPLE TYPES

Sets are surrounded by the brace symbols \{, \}.\footnote{The parenthesis/star combination \((*, *)\) can also be used to define a set although this is rarely used. Example: \texttt{first.set} := \((* 1, 1, 2, 2, 3, 3, 3 *)\);}

\begin{verbatim}
> empty := \{\}; # this set is empty
empty := \{\}
> FirstSet := \{1, 1, 2, 2, 3, 3, 3\};
FirstSet := \{1,2,3\}
> SecondSet := \{1, 2, 3\};
SecondSet := \{1,2,3\}

The sets FirstSet and SecondSet are equal since both contain exactly the elements 1, 2, 3.

> ascend := \{0, 5, 10, 15, 20\};
ascend := \{0,5,10,15,20\}
> descend := \{20, 15, 10, 5, 0\};
descend := \{0,5,10,15,20\}

Again, \texttt{ascend} and \texttt{descend} are equal because sets are unordered.

We can perform several operations on objects of this type including \texttt{union, intersection, member, subset} and \texttt{minus}. All of these operations correspond to the standard mathematical definitions.

\begin{verbatim}
> cdn := \{'beaver', 'moose', 'grizzly', 'loon', 'cow', 'deer'\};
ch := \{'sheep', 'mouse', 'cow', 'deer'\};
> intersect(cdn, ch);
{cow,deer}
> union(cdn, ch);
{beaver,cow,deer,grizzly,loon,moose,mouse,sheep}
> member('grizzly', ch);
false
> minus(cdn, ch);
{beaver,grizzly,loon,moose}
> minus(ch, cdn);
{mouse,sheep}
\end{verbatim}
We are not limited to placing only integer and string items in set structures. We could place any type of data we desire including other set and list objects. When each element of a set has the same type, we say that the set is homogeneous, otherwise we say that it is heterogeneous.

```r
> mixed := {'salmon', {'coho', 'pink'}, 0, 'grizzlies', 1000, 'mosquitos'};
> mixed := {0,1000,grizzlies,mosquitos,salmon,{'coho','pink'}}
```

The set `mixed` contains objects of three types: string, integer, and set.

### 1.5.2 Lists and Arrays

**Lists**

Whereas a set contains distinct unordered items, the Darwin list data type allows the user to create ordered multisets. A multiset is a set which allows more than one instance of the same object. An ordered list is a group of expressions surrounded by square brackets `[, ]`.

```r
> FirstTry := [1, 1, 2, 3];
> FirstTry := [1, 1, 2, 3]
> SecondTry := [1, 2, 3];
> SecondTry := [1, 2, 3]
```

The two lists are not equivalent since `FirstTry` has two copies of the number 1 while `SecondTry` has only a single copy.

```r
> AscendList := [1, 2, 3, 4];
> AscendList := [1, 2, 3, 4]
> DescendList := [4, 3, 2, 1];
> DescendList := [4, 3, 2, 1]
```

Again, `AscendList` and `DescendList` are not equal since the order of the elements in the list matters.

Like a set, a list are allowed to be heterogeneous (elements may have different types). We can also include a list inside of a list.
1.5. SIMPLE TYPES

Figure 1.2: Viewing the list as a chain.

We could define a list of animals as follows:

```haskell
> animals := ['chimpanzee', 'gorilla', 'monkey', 'orangutan'];
> animals := [chimpanzee, gorilla, monkey, orangutan]
```

We can think of the list as a chain with the first element containing the string item 'chimpanzee'. The second element of the chain is 'gorilla' and so forth.

To access the individual elements of `animals`, we provide a number (or range of numbers) indicating which link(s) in the chain we are interested in.

```haskell
> animals[1];  # the first link in the chain
  chimpanzee
> animals[2..4];  # the second through fourth links in the chain
  [gorilla, monkey, orangutan]
```

Arrays

The array data type is identical to the list data type in Darwin as both implement ordered multisets. The difference between the two is solely conceptual: (in computer science terms) lists are typically thought of as finite chains of data while an array is thought of as indexed collection of data. Let us make this a bit more clear with an example.
Figure 1.3: A one dimensional array.

We could define an empty array with four elements and assign to each
element independently. To define an empty array, we use the `CreateArray`
function.

```plaintext
> animals := CreateArray(1..4);
> animals := [0, 0, 0, 0]
```

Each of the four elements of `animals` is initialized to the value 0. To assign
to each element, we index the variable name `animals`.

```plaintext
> animals[1] := 'chimpanzee';
> animals[1] := chimpanzee
> animals[2] := 'gorilla';
> animals[2] := gorilla
> animals[3] := 'monkey';
> animals[3] := monkey
> animals[4] := orangutan
```

To examine the contents of the list we again index the variable name with
the location or the range of locations we are interested in.

```plaintext
> animals[1];
  chimpanzee
> animals[2..4];  # examine only the last 3 elements of the array
  [gorilla, monkey, orangutan]
```

There is absolutely no semantic difference between the two types. The
synonyms are offered because it is sometimes more appropriate to think of
your data as a list than an array and vice versa.
1.5. SIMPLE TYPES

We can define an array of two dimensions by specifying a second range as an argument to the CreateArray command. To access the elements, we specify the index in both dimensions separated by a comma.\(^3\)

\[
\begin{align*}
> \text{square} & := \text{CreateArray}(1..5, 1..7); \\
> \text{square}[1, 3] & := 5; \quad \text{# assign 5 to the element in} \\
& \quad \text{# row 1, column 3} \\
\text{square}[1, 3] & := 5 \\
> \text{square}[5, 7] & := 10; \quad \text{# assign 10 to the element in} \\
& \quad \text{# row 5, column 7} \\
\text{square}[5, 7] & := 10
\end{align*}
\]

Matrices

An array with an equal number of elements in each dimension is called a matrix. We can test whether an array is a matrix in Darwin using the type function and type name matrix.

\[
\begin{align*}
> \text{type}([[1,2],[3,4]], \text{matrix}); & \quad \text{# It is.} \\
\text{true} \\
> \text{type}([1, [2, 3]], \text{matrix}); & \quad \text{# It isn't.} \\
\text{false}
\end{align*}
\]

Matrices are used quite often in numerical and symbolic computation and we will use them in Chapters 18 – Point Accepted Mutations and Dayhoff Matrices and 21 – The Pairwise Comparison of Amino Acid Sequences extensively.

More generally, we can define arrays with any number of dimensions using this CreateArray command.

\[
\begin{align*}
> \text{cube} & := \text{CreateArray}(1..5, 1..5, 1..5, 'a'); \\
\text{cube} & := [[[a, a, a, a, a], [a, a, a, a, a], [a, a, a, a, a]], [[a, a, a, a, a], [a, a, a, a, a], [a, a, a, a, a]]]
\end{align*}
\]

\(^3\)The element at location \(i, j\) may be accessed by \(\text{square}[i,j]\) or, equivalently, \(\text{square}[i][j]\).
Figure 1.4: An array of type \texttt{matrix}. The array has dimension two and there are an equal number of elements in both dimensions. \texttt{square[1,1]} = \texttt{f}, \texttt{square[3,4]} = \texttt{j}, ...

\begin{verbatim}
a, a, a], [a, a, a, a, a], [a, a, a, a, a], [a, a, a, a, a], [a, a, a, a, a], [a, a, a, a, a], [a, a,]
[[a, a, a, a, a], [a, a, a, a, a], [a, a, a, a, a], [a, a, a, a, a], [a, a, a, a, a], [a, a, a, a, a], [a, a, a, a, a],

> cube[5,5,5];
a
\end{verbatim}

If you examine the initialized array that Darwin returns, you will notice that it is an \texttt{array} with five elements each of which consists of an \texttt{array} with five elements each of which consists of an \texttt{array} with five elements. The last argument \texttt{'}a\texttt{'} specifies the initialization value for each element.

As is the case with type \texttt{set}, the data in an array need not be of homogeneous type. Darwin’s allowance of heterogeneous data in these structures makes them extremely flexible and, in later chapters, we will use some form
Figure 1.5: A three dimensional array. \( \text{cube}[3][3][1] = a, \text{cube}[5,5,5] = b, \text{cube}[1,5,3] = c, \ldots \)

of them again and again in almost every routine we build.

There is sometimes confusion between the data types list, array and matrix in Darwin. In actuality, the type list coincides exactly with the type array and the terms may be used interchangeably. Only arrays with an equal number of elements in every dimension are of type matrix.

### 1.5.3 Strings

Any set of characters (including no characters) enclosed by the single quote symbol (') is defined to be of type string in Darwin.

```
> thoughts := '';                      # an empty string
thoughts :=
> MyName := 'mike hallett';           # a string
MyName := mike hallett
> MyAge := 'twenty eight';            # a string
MyAge := twenty eight
```

Each item of type string is indexed and its individual components can be accessed in the same manner as one accesses the individual components
of an array.

> MyName[6];
h
> MyAge[2];
w

The period Operator

The period operator (.) allows us to concatenate two string items together.

> MyName[1..5].MyAge.thoughts;
mike twenty eight

The CreateString Function

We can define a new string object via the command

CreateString(dim : posint , init : symbol)

where dim is the length of string and init is the initial value for each element
of the string. This init symbol must be a letter a...z, A...Z, or the
underscore symbol (_). If CreateString is called with only one argument,
the string is initialized to the space symbol.

> let_there_be_space := CreateString(10000): # but let's not display it

> str := CreateString(100, a);
str := aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
> type(let_there_be_space, string);
true

1.5.4 Built-in Functions for Sets, Arrays and string

Once we have defined a set, an list (array) or a string, we may want to
add or delete data to and from it. To this end, Darwin has several operators
implemented which allow one to easily change the data structure.
The op Operator

The op operator takes either a set, list (array) or string and returns the sequence of expressions formed by removing the outermost braces (set), square brackets (list) or single quotes (string).

```haskell
> weekdays := ['monday', 'tuesday', 'wednesday', 'thursday', 'friday'];
>        # a list
weekdays := [monday, tuesday, wednesday, thursday, friday]
> op(weekdays);
monday, tuesday, wednesday, thursday, friday    # a string
> op(weekdays[1]);
numbers := {1, 2, 3, 4};                      # a set
numbers := {1, 2, 3, 4}
> op(numbers);
1, 2, 3, 4
```

We can exploit this operator to combine two lists into a single new list.

```haskell
> weekdays := ['monday', 'tuesday', 'wednesday', 'thursday', 'friday']:
> weekends := ['saturday', 'sunday']:
> week := [ op(weekdays), op(weekends) ];
week := [monday, tuesday, wednesday, thursday, friday, saturday, sunday]
```

The length Function

The length operator accepts a set, list or string as a parameter and returns the length of the object.

```haskell
> numbers := [1, {2, 3}, [4, {5, 6}]];        # a list with 3 elements
> length(numbers);
1
> length('I have length 16');              # a string
16
```
We can use length function to remove the last element of a list.

> length(weekdays);
5
> shortweek := weekdays[1..length(weekdays)-1];  # taking Friday off
shortweek := [monday, tuesday, wednesday, thursday]

1.6 Simple Output

In order to display the results generated by your programs, Darwin includes several formatting commands designed to give you control over where and how information is sent to the terminal. The simplest of these are the print (pretty print) and lprint (linear print) commands.

> square := [[1,2], [3,4]]:
> print('This is a pretty print of a square matrix', square);
This is a pretty print of a square matrix
1 2
3 4
> lprint('This is a linear print of a square matrix', square);
This is a linear print of a square matrix [[1, 2], [3, 4]]
> lprint(72483734723897348372847382);  # big numbers are printed
7.2483734723897358e+25
    # in exponential notation.
> print('This isn''t difficult to use');  # To print a single quote,
A two for one sale on backslashes \ \ \;
# use the '' sequence.

This isn't difficult to use
> print('A two for one sale on backslashes \\
\');
A two for one sale on backslashes \\
> lprint('A sequence of expressions ', 2, xyz, 2^5);
A sequence of expressions  2 xyz 32

Both commands will accept an arbitrary length sequence of expressions as long as the elements are separated by commas. The lprint does not place a carriage return and newline after displaying each expression whereas the
1.7. BOOLEAN EXPRESSIONS

print command does. The behaviour of the print statement is dependent on the type of the expression. If the expression is of type matrix and it has dimension two, as is the case with variable square above, it prints out the square array in a graphical manner preserving this structure. We will see in Chapter 10 – Overloading, Polymorphism and Object Orientation that print can be extended to “pretty print” any data structure.

1.7 Boolean Expressions

An expression which evaluates to either true or false is called a boolean expression. Boolean expressions are used extensively in programming language constructs such as if-then-fi commands and while loops. Darwin offers and, or and not for building such expressions. The and returns true if and only if both arguments are true. The not operator negates a boolean value.

> u := true; v := false;
false
> (u and v);
false
> (u and (not v));
true
> flag := true:
> not((not flag) = (not not flag));
false

The or operator returns false if and only if both arguments are false.

> u := true; v := false:
> (u or v);  # one must be true for 'or'
> (not u) or v;  # u -> v

We can build more complex boolean expressions using these operators, expressions and the comparison operators. Table 1.4 contains a list of all comparison and boolean operators.

> (6 = 1 + 2 + 3) and (720 = factorial(6));
true
> x := 57;
> (mod(x, 2) = 0) or (mod(x, 3) = 0) or (mod(x, 5) = 0);  # is x divisible
             # by 2, 3, or 5?
false
> 5 < 7;
true
> floor(6.5) <= 6;
true
> 1 <> 2;
true

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>true iff both operands are true</td>
</tr>
<tr>
<td>or</td>
<td>false iff both operands are false</td>
</tr>
<tr>
<td>not</td>
<td>true iff operand is false</td>
</tr>
<tr>
<td>=</td>
<td>true iff operands are equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>true iff left operand is less than right</td>
</tr>
<tr>
<td>&gt;</td>
<td>true iff right operand is greater than right</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>true iff operands are not equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>true iff left is less than or equal to right</td>
</tr>
<tr>
<td>&gt;=</td>
<td>true iff left is greater than or equal to right</td>
</tr>
</tbody>
</table>

Table 1.4: List of all comparison and boolean operators.

1.8 Decisions

The if-then-fi Command

No programming language is useful unless it allows programs to make decisions. The typical construct used for “controlling the flow of execution” is the if-then-fi command:

> x := 5:
> if (x=5) then
define the value 5 to the variable name x. The
else
    print('I'm a number five');
    print('I'm not a number five');
fi:
I'm a number five

The first statement x:=5 assigns the value 5 to the variable name x. The
boolean expression (x = 5) found in the second line is called the conditional
of the if-then-fi command. In the example here, the variable x contains
the integer 5, so the boolean conditional is true and the print command is
executed. If the variable had not been assigned 5, then the boolean conditional would evaluate to false and the program would perform the second
print command located in the else case.

The else command can be omitted from an if-then-fi command.

Nesting if-then-fi Commands

We can nest if-then-fi commands inside of if-then-fi commands when
there are more than two possible outcomes.

> x := 5:  # compare if x<y or x>y or x=y
> y := 7:
> if (x < y) then
  > lprint(x, ' is less than ', y);
> else
  > if (x > y) then
    > lprint(x, ' is greater than ', y);
  > else
    > lprint(x, ' is equal to ', y);
> fi;
> fi:
5 is less than 7
Here the second if-then-fi command is contained within the else case of the outer if-then-fi. It is evaluated only if \( x \geq y \).

**The elif Command**

We can combine the previous two if-then-fi into a single if-then-fi with the addition of the elif (else if) command.

```plaintext
> x := 100:
> y := 34:
> if (x < y) then
>   lprint(x, ' is less than ', y);
> elif (x > y) then
>   lprint(x, ' is greater than ', y);
> else                      # x must equal y
>   lprint(x, ' is equal to ', y);
> fi;
100 is greater than 34
```

We are allowed any number of elif conditions.

**The If Command**

Another if-then-fi variant that offers some convenience from time to time is the If(\( cond, exprtrue, exprfalse \)). This function returns the result of evaluating \( exprtrue \) if the boolean conditional \( cond \) is true or it returns the result of evaluating \( exprfalse \) if \( cond \) is false.

```plaintext
> x := 101:                  # x is just some number
> round_up := If( mod(x, 2)=0 , x/2, (x+1)/2 );
round_up := 51
> x := 44:
> round_up := If( mod(x, 2)=0 , x/2, (x+1)/2 );
round_up := 22
```
1.9 Looping Around

For a programming language to be of any use, it must include some ability to loop through a set of statements. Recall that a statement in Darwin is a command or expression followed by a terminating symbol (the semicolon or colon). A command sequence is a sequence of commands separated by terminating symbols. The terminating symbol after the last command in a command sequence is optional. The for command is our first example of iteration in Darwin.

> X:=CreateArray(1..100):
> for i from 1 to length(X) do
>     X[i]:=i: # a command sequence
>     lprint(`The value of X[`, i, `] is `, X[i])  # no terminating semicolon
>     od: # or colon needed
>
The value of X[ 1 ] is 1
The value of X[ 2 ] is 2
The value of X[ 3 ] is 3
The value of X[ 4 ] is 4

After the for loop terminates, the \(i\)th element of \(X\) contains the number \(i\).

Of course, we could have accomplished the same by writing one hundred different statements of the form \(X[1]:=1, X[2]:=2, \ldots, X[100]:=100\). The body of the loop consists of the command sequence between the \texttt{do}..\texttt{od} commands.

Now suppose we only want to print those elements of \(X\) which are odd and divisible by both 3 and 5. Since all elements of \(X\) which have an even index are assigned an even integer, we need only check the odd elements of \(X\) to see if they are divisible by these numbers.

> for i from 1 to length(X) by 2 do
CHAPTER 1. EXPLORING THE BASICS

> if ((mod(X[i], 3)=0) and (mod(X[i], 5)=0)) then
> lprint(quad(\text{'X['}{1..i}\text{(','i,')}\text{='}\text{'X[i],'})\text{ is divisible by both 3 and 5''});
> fi;
> od;
X[ 15 ]= 15 is divisible by both 3 and 5
X[ 45 ]= 45 is divisible by both 3 and 5
X[ 75 ]= 75 is divisible by both 3 and 5

Here \texttt{mod(x,y)} denotes the modulus function; this returns the integer remainder after dividing \texttt{y} into \texttt{x}. Note the use of the \texttt{by 2} clause in the \texttt{for} command. In general, you may count in steps by any value you would like. We may also specify a negative increment to count backwards. If we change the first line of the about \texttt{for} loop to

> for \texttt{i} from \texttt{length(X)-1} to \texttt{1} by \texttt{-2} do
then the exact same is accomplished but in the reverse order.

X[ 75 ]= 75 is divisible by both 3 and 5
X[ 45 ]= 45 is divisible by both 3 and 5
X[ 15 ]= 15 is divisible by both 3 and 5

Chapter 6 describes several other means of looping in Darwin.

1.10 Interrupting Computation

At one time or another, you will get caught in an infinite loop. Here are some particularly simple examples of programs which never terminate.

> do od;

> for \texttt{i} from \texttt{1} to \texttt{10} do
> if \texttt{(i=10)} then
> \texttt{i:=1;}
> fi;
> lprint(quad(\text{''i = '}{1..i}\text{,'});
> od;
To interrupt the computation, type the <control-c> sequence (press the c key while simultaneously holding down the control (or ctrl) key).

1.11 Ending a Darwin Session

We can end a session by typing one of the commands quit, done or stop. The total amount of CPU usage is indicated and control is returned to your operating system.

> quit;

Alternatively, we may terminate a session by pressing the <control-c> sequence twice before terminating a command with either a semicolon or colon.

1.12 Navigating Around the System

There are many built-in routines and data types in Darwin that we have not yet encountered. Later chapters contain a more detailed expository of the common but more complex functions available. Part ?? – The Reference Guide provides a complete list of all built-in routines and types available to the user. Beyond this, there are several fast on-line ways to look for information regarding built-in tools. The commands

? topic
help(topic) or
print(topic)

display information about keywords in Darwin which match topic or approximately match topic. (Usage of print results in only exact matches to topic.)

Suppose you would like to know if there is a function available which calculates the square root of a number. To find the appropriate function, one might type
> ? square root;

or

> help('square root');

This leads one directly to the function \texttt{sqrt()}.

\textbf{Function \texttt{sqrt}( x:numeric )}

Compute the square root of a number.

A list of topics available in the system help is found as follows:

> ? index;

Querying the help utility with an item from this list gives more specific information.

A good way to get to know the complement of Darwin tools is to browse through the output produced by the \texttt{names} routine.

> names();

This generates a list of all assigned names in your current session. This is especially useful if, during one of your Darwin sessions, you should forget what you have called a variable, data structure or procedure. The forgotten name will appear somewhere in this list. If you accompany the command \texttt{names} with the name of a built-in type, you receive the list of names for routines using that type.

> names(real);
Chapter 2

Advanced String Manipulation

The printf command, presented in Chapter 1 – Exploring the Basics, was our first tool for sending information to the terminal. This section explores a second printing feature in Darwin modeled closely after the printf (format print) command from the language C [22].

The ability to perform searching and matching operations is a necessity when dealing with a large amount of textual data (like an annotated sequence database). Darwin offers both exact and approximate matching routines.

2.1 Printing

2.1.1 printf

The printf command (format printing) translates internal values into characters for sending to the standard output. The simplest form of this command echos back the text pattern it is passed.

> printf('The simplest form of this command echos');
CHAPTER 2. ADVANCED STRING MANIPULATION

However, we can use the text pattern to control the exact format we would like for the output through the use of conversion patterns and cursor control characters. For example,

```c
> printf(\nFirst a carriage return & new line\n');
First a carriage return & new line
> printf(‘then print an integer %d’, 101);
then print an integer 101
> printf(\nstep\nstep\nstep\nstep\n');
step
step
step
step
```

A text pattern is an ordinary name with the addition of zero or more conversion patterns. Each conversion pattern begins with a percent symbol (%) and is followed by

- a minus symbol (-) indicating the argument should be left justified,
- a number that specifies the minimum field width,
- a period which separates the field width from the precision,
- a number indicating the precision for a real value (this is the number of digits after the decimal point),
- a character from the set {a, c, d, e, f, g, l, o, s, u, x, %}.

(Table 2.1 contains a description of the function for each.)

```c
> printf(‘%a, %a’, ['L', 'I', 'S', 'T'], ’a means any structure’);
[L, I, S, T], a means any structure
```
### 2.1. PRINTING

<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>prints any Darwin value including lists and sets</td>
</tr>
<tr>
<td>c</td>
<td>prints a single character</td>
</tr>
<tr>
<td>d</td>
<td>prints an integer</td>
</tr>
<tr>
<td>e</td>
<td>prints a number in exponential notation</td>
</tr>
<tr>
<td>f</td>
<td>prints a real number</td>
</tr>
<tr>
<td>g</td>
<td>prints a real number in the same manner as the e character if the exponent is greater than 8. Otherwise equivalent to the f or d characters.</td>
</tr>
<tr>
<td>o</td>
<td>prints the octal conversion of an integer</td>
</tr>
<tr>
<td>s</td>
<td>prints out an item of type name</td>
</tr>
<tr>
<td>u</td>
<td>prints an unsigned integer</td>
</tr>
<tr>
<td>x</td>
<td>prints the hexadecimal conversion of an integer</td>
</tr>
<tr>
<td>%</td>
<td>prints a percent sign %</td>
</tr>
</tbody>
</table>

Table 2.1: Conversion characters for the printf command.

```c
> int := 1234;
> printf(‘\n|\%d|\%10d|\%-10d|’, int, int, int, int); # integers
1234   1234   1234   1234 |
> printf(‘\n|\%11s|\%12s|\%12s|\%12s|’, ’normal’, ’field of 12’, ’5 decimal’, ’left flush’);
| normal| field of 12| 5 decimal| left flush|
> r := 1234.567;
> printf(‘\n|\%lf|\%12.5f|\%12.5f|’, r, r, r, r);
1234.567000   1234.567000   1234.567000   1234.567000 |
```

The backslash symbol (\) is treated as a special symbol in a *text pattern*. Followed by a letter from Table 2.2, it can be used to control the position of the cursor.

```
> printf(’    <- The number five goes here\r 5\n’);
5  <- The number five goes here
> printf(’A newline\nfollowed by a single quote ’’’);
A tab
    followed by a single quote ’
```
<table>
<thead>
<tr>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>\b</code></td>
<td>backspace</td>
</tr>
<tr>
<td><code>\n</code></td>
<td>carriage return and newline</td>
</tr>
<tr>
<td><code>\t</code></td>
<td>tab</td>
</tr>
<tr>
<td><code>\v</code></td>
<td>newline</td>
</tr>
<tr>
<td><code>\r</code></td>
<td>carriage return</td>
</tr>
<tr>
<td><code>\</code></td>
<td>\</td>
</tr>
<tr>
<td><code>\'</code></td>
<td>'</td>
</tr>
</tbody>
</table>

Table 2.2: The cursor control sequences for the `printf` command.

The syntax for a `printf` command is as follows:

```
printf(textpattern, exp_1, exp_2, ...)
```

The value of expression `exp_i` is placed in the `i`th conversion pattern found in the `textpattern` according to the type of this conversion pattern. If the number of expressions is greater than the number of conversion patterns, the extraneous ones are ignored. If the number of conversion patterns is greater than the number of expressions, Darwin responds with an error. One must take care to match correct conversion patterns with the type of the corresponding expression. In some cases, Darwin will not respond with an error and the resulting text may appear nonsensical.

The best way to become familiar with the `printf` command in Darwin is through examples and experimentation.

```
> printf(\n%e \b %g \b %g\n', 1234567, 1234567, 1234567);
1.234567e+06 1.23457e+06 1234567
> printf(\n0ctal: %o, Hexadecimal: %x',888, 888);
Octal: 1570, Hexadecimal: 378
> printf(\nA character %c followed by a string %s', 'z', 'hello');
A character z followed by a string hello
> printf(\nA percent sign % followed by a \" symbol.\');
A percent sign % followed by a ' symbol.
```
> printf('\nfirst a new line and carriage return\nnow just a newline');
first a new line and carriage return
now just a newline

2.1.2 sprintf

The sprintf command is identical to the printf command except that instead of sending the result of evaluating the text pattern to the terminal, it returns it as a name.

sprintf(text pattern, exp\_1, exp\_2, ...)

The \%a can not be included as a conversion pattern in text pattern.

> res := sprintf('\%s \%d.', 'ACCAGACCATG', 11);
res := ACCAGACCATG 11.

2.2 Exact string Search

2.2.1 String Format Scan – sscanf

The sscanf function provides the same service as the printf does except in the opposite direction.

sscanf(text : name, pattern : name)

The text is the item of type name you would like to scan (any name which can be produced by a printf command is valid). The second is the pattern which may contain conversion patterns (see §2.1.1). The pattern name specifies how to interpret text. Starting at the leftmost character of text, this function attempts to match the first conversion pattern in pattern to text. If it succeeds, it stores this as the first element of a list. It then tries to match the second element of the conversion pattern at the point in text where it left off. When either (1) all conversion patterns have been exhausted, (2) text is entirely parsed, or (3) it fails to match a conversion pattern to text, sscanf finishes and returns the list of successful matches.

> matches := sscanf('MMAA21336 standard; RNA; EST; 410 NToInt',
>                   '\%s \%s \%s \%d \%s');
In general, the \textit{pattern name} may contain:

- Spaces. These are ignored.
- Ordinary characters (except the \% symbol).
- Any cursor control characters (see Table 2.2). These are ignored.
- The slash character (\textbackslash{}) followed by a conversion character. The character (any structure) is not supported. Table 2.1 contains a complete list of such options.

### 2.2.2 Searching for Strings

\texttt{SearchString(pattern, target : name)}

\texttt{CaseSearchString(pattern, target : name)}

Darwin offers two high level functions – \texttt{CaseSearchString} and \texttt{SearchString} – to find a pattern in an item of type \texttt{string}. The first is case sensitive (meaning the strings \texttt{hello} and \texttt{Hello} are not equivalent); the second is not. Both commands take two \texttt{string} parameters; the first parameter is the \texttt{pattern} and the second is the \texttt{target}. If the \texttt{pattern} is found in \texttt{target}, the position before the character where \texttt{pattern} was found in \texttt{target} is returned. If the \texttt{pattern} is not found, the value -1 is returned.

```plaintext
> SearchString('Waldo', 'can you find waldo in here'); # case insensitive
10
> CaseSearchString('it', 'It is in here'); # case sensitive
-1
> SearchString('am', 'Here I am'); # not found
0
```

### 2.2.3 Searching with SGML Tags

In recent years, the ISO-SGML (Standardized Generalized Markup Language) tagging convention has begun to be used extensively, most notably
2.3. **APPROXIMATE STRING SEARCH**

with HTML and the World Wide Web [21]. An SGML tag surrounds a body of information with an *opening* and *closing label*. The opening label is of the form `<lb1>` and the closing label is of the form `</lb1>` where the type label `lb1` can be almost any sequence of symbols. SGML is extremely flexible since it allows for an infinite number of different labels, it allows for an arbitrary depth of nested tags, it is relatively economical in terms of storage, and, because of the angle brackets `<` and `>`, it easy to recognize the beginning and end of a body of information. For these reasons, the ISO-SGML tagging convention was chosen for use in the Darwin genetic database structure.

The `SearchTag` function searches an item of type *string* for an opening and closing SGML tag and returns the information contained between them. The general syntax is

`SearchTag(taglabel, target)`

where `taglabel` is an SGML type label and `target` is an arbitrary name.

```python
> SearchTag('AC',
> ' <ID>11S3_HELAN</ID><AC>P19084;</AC><DE>GLOBULIN</DE> ');
P19084;
```

We examine a number of related routines in Chapter 8—*Genetic Databases*.

### 2.3 Approximate string Search

At times, we may want to search a large body of textual information for all words close to a given pattern. Consider the help function ? offered in Darwin (§1.12). This function displays the description lines for all routines which have a word close to your query. If you type

```bash
?  square
```

you are given the description for the built-in Darwin function `sqrt` (amongst others). Darwin offers two routines to accomplish this type of query.
2.3.1 ApproxSearchString

To perform an approximate text search, we use the routine

\[
\text{ApproxSearchString} \left( \text{pattern} : \text{name}, \text{target} : \text{name}, \text{tol} : \text{posint} \right)
\]

where \text{pattern} is the pattern we are searching for, \text{target} is the body of text to be searched and \text{tol} is the positive integer representing the number of mismatches between the \text{pattern} and the \text{target} allowed. Note that spaces are counted as mismatches and the function is case insensitive (\text{hello} is equivalent to \text{HeLlo}). Like \text{SearchString}, if the \text{pattern} is found in the \text{target}, the function returns the offset of the character directly before where \text{target} was found in \text{pattern}. If it is not found (within the tolerance \text{tol}), the value \(-1\) is returned.

\[
> \text{ApproxSearchString}('\text{hallo}', 'AAAAAAAHeLLoBBBBB', 1);
8
\]

\[
> \text{ApproxSearchString}('\text{any}', 'jdkfajdsjfasdkjadsfj', 3);
0 \quad \# \text{any match is okay}
\]

\[
> \text{ApproxSearchString}('\text{nothing}', 'N.0.T.H.I.N.G.', 4);
-1
\]

2.3.2 BestSearchString

The \text{BestSearchString} function

\[
\text{BestSearchString} \left( \text{pattern} : \text{name}, \text{target} : \text{name} \right)
\]

finds the match of a pattern against a body of text which minimizes the total number of mismatches. It returns an offset to the character directly before where this best match occurs.

\[
> \text{BestSearchString}('\text{CYIQNCPBG}', 'PPATBCYTQCPLGFPTISPS');
5
\]

\[
> \text{BestSearchString}('\text{CYIQNCPBG}', 'XXXXXXXXXXXXXXXXXX');
\quad \# \text{just take the first}
\quad 0
\]

\[
> \text{BestSearchString}('\text{CYIQNCPBG}', 'XXXXXXXXXXXXXXXXXX');
\quad \# \text{just take the first}
\quad 0
\]
Chapter 3

Procedures

Suppose for a moment we have just finished performing several experiments and each of these experiments has produced a set of data points. Suppose we are required to find the average value for each individual experiment.

> experiment_1 := {33, 234, 5233, 122, 32043, 9}:
> experiment_2 := {6435, 2234, 22, 8902, 30183}:
> experiment_3 := {8893, 902, 10283, 9918020, 52, 2356}:
  .
  .
  .

We could accomplish this easily using our knowledge of the for loop and the length function.

> total := 0:
> for i from 1 to length(experiment_1) do
>       total := total + experiment_1[i]:
> od:
> lprint('The average is ', total/length(experiment_1));
The average is 6412.3333

Unfortunately, it becomes an exhausting typing exercise to repeat this code for each experiment, each time replacing experiment_i with experiment_i+1.
It would be nice to encapsulate this short sequence of statements into a single construct which would perform the same repetitive action but on different sets. The *procedure* construct in Darwin does exactly this.

Procedures, when properly used, enable programmers to build on the work of previous programmers, they make programs easier to read by hiding the low-level details of an operation, and they make modifying your programs in the future less painful.

### 3.1 The *proc* Command

A *procedure* is a programming construct which groups together a set of commands to form a single statement. Once grouped this procedure can be called with different *arguments*. In Darwin, procedures are data structures (we will come back to this topic later) and typically have a *name* in the same fashion as variables and other data structures have names. This *name* is used to *invoke* or *execute* the routine. The built-in commands *proc* and *end* are used to define procedures as follows:

```plaintext
> FirstTry := proc()
> print('We can put as many statements as we''d like in here');
> print('Now preparing for departure...');
> end;
```

The *body* of the procedure consists of two *print* statements. To invoke the procedure, we type:

```plaintext
> FirstTry();
```

We can put as many statements as we’d like in here
Now preparing for departure...

The parenthesis surround the *formal arguments* (or *formal parameters*) to a procedure. In the above case, there are no arguments. Arguments allow information to be passed to a procedure so that it may be used by the body. This data may be of any type. We can now write a procedure to find the average value in a set by passing an argument of type *set*. 
\[3.1. \ THE \ PROC \ COMMAND\]

\>$\text{SetAverage} := \text{proc}(s : \text{set})$
\>$\text{\quad description \ 'This procedure takes a set as an argument and}$
\>$\text{\quad \quad calculates the average value in the set'};;$
\>$\text{\quad total := 0;}
\>$\text{\quad for i from 1 to length(s) do}$
\>$\text{\quad \quad total := total + s[i];}$
\>$\text{\quad od;}
\>$\text{\quad lprint('The average is ', total/length(s));}$
\>$\text{\quad end;}$

In procedure \text{SetAverage}, there is one argument \emph{s} of type \text{set}. When we invoke \text{SetAverage} accompanied with the argument, Darwin checks to see if the data we have passed to it are of the correct type. The invocations

\>$\text{\text{SetAverage}([1, 5, 10, 15]);}$
\>
The average is 7.7500
\>$\text{\text{SetAverage}([66]);}$
\>
The average is 66

are perfectly okay although

\>$\text{\text{SetAverage}([1, 5, 10, 15]);}$
\>
\text{SetAverage expects a 1st argument, s:set, found: [1, 5, 10, 15]}
\>
\emph{Error, invalid arguments}
\>$\text{\text{SetAverage}('infiltrate and destroy');}$
\>
\text{SetAverage expects a 1st argument, s:set, found: infiltrate and destroy}
\>
\emph{Error, invalid arguments}

both result in errors. Specifying the type of the parameter in the \text{proc} declaration allows Darwin to perform \emph{type checking}. Computer science wisdom says that programs become easier to understand and debug when more \emph{type checking} is included in the program. However, specifying the type of arguments is optional in Darwin and if we had chosen to write the first line of \text{SetAverage} as

\>$\text{\text{SetAverage} := proc ( ) \quad \quad \# no type declaration}$
\>
...
no such type checking would be carried out. Any argument passed to the
routine would be accepted although this may cause errors within the body of
the procedure. We can extend \texttt{SetAverage} to perform stricter type checking.

> \texttt{SetAverage\_2 := proc(s : set(real))}
> \texttt{description 'This procedure takes a set of real values as an}
> \texttt{argument and calculates the average value in the set'};
> \texttt{total := 0;}
> \texttt{for i from 1 to length(s) do}
> \texttt{total := total + s[i];}
> \texttt{od:}
> \texttt{lprint('The average is ', total/length(s));}
> \texttt{end:}

Now, whenever \texttt{SetAverage\_2} is invoked, each element of the \texttt{s} is checked
whether it is of type \texttt{real}.

> \texttt{SetAverage\_2( {'I', 'am', 'sneaky'} );}  \# The first proc doesn't
> \texttt{# stop this. The error}
> \texttt{# occurs in the body.}

Error, (in SetAverage) invalid addition of text

\texttt{# the second does stop it}

> \texttt{SetAverage\_2( {'but', 'I', 'fail', 'anyway'} );}
\texttt{SetAverage\_2 expects a list argument, s: set(real),}
\texttt{found: {I, anyway, but, fail} Error, invalid arguments}

In general, each nested level of a parameter can be checked by nesting the
specified type inside of parentheses.

> \texttt{TypeCheckingProcedure := proc (weird : list(list(integer)))}
> \texttt{print('This data is A-OK', weird);}
> \texttt{end:}
> \texttt{TypeCheckingProcedure([[5, 6, 7], [8, 9, 10]]);}
\texttt{This data is A-OK}
\begin{verbatim}
  5  6  7
  8  9 10
\end{verbatim}
3.2 Commenting a Procedure: description

The description command in Darwin provides a means of documenting a procedure.

\[\text{SetAverage} := \text{proc}(s : \text{set})\]
\[\quad \text{local total;}\]
\[\quad \text{description 'This procedure takes a set as an argument and}\]
\[\quad \quad \text{calculates the average value in the set'};\]
\[\quad \text{total} := 0;\]
\[\quad \text{for i from 1 to length(s) do}\]
\[\quad \quad \text{total} := \text{total} + s[i];\]
\[\quad \text{od;}\]
\[\quad \text{print('The average is ', total/length(s));}\]
\[\text{end}:\]

If we type

\[\text{print(SetAverage);}\]

the text surrounded by the single quote symbols (’) following the description command is echoed to the terminal.

SetAverage: Usage: SetAverage( s:set )
This procedure takes a set as an argument and
calculates the average value in the set

The description command must follow any local or global declarations and any option commands.

\[\text{example} := \text{proc( ... )}\]
\[\quad \text{local x, y, z;}\]
\[\quad \text{global a, b, c,;}\]
\[\quad \text{option polymorphic;}\]
\[\quad \text{description 'The following text is displayed by the print command'};\]
\[\quad \ldots\]
\[\text{end}:
Computer science wisdom says that writing clear meaningful comments in your routines makes for easier to understand and debug code. However, the \texttt{description} command is optional.

### 3.3 Scoping Rules

The \textit{scope} of a name (variable names, data structure names, procedure names) is the part of the program within which the name can be used. Darwin employs \textit{dynamic scoping}, that is, Darwin does not require that you declare variables before you use them.\footnote{The other common scoping protocol is \textit{static scoping} where all variables must be declared before they are used.} Thus,

\begin{verbatim}
> x := 5:
> x := x * 99:
> y := z:
\end{verbatim}

are perfectly digestible by the Darwin system. This section explores the scoping rules employed by Darwin for variables, structures and routines. Instead of a formal description of the scoping rules Darwin uses, we use a series of examples which covers all cases one would normally encounter when writing programs.

So far all of the names we have defined have been \textit{global} meaning their scope is the entire Darwin environment. There are situations when we do not necessarily want some variables to be \textit{visible} to the entire world. The temporary variables created by procedures are a common example of this; the variables have no meaning outside of the scope of the procedure and should therefore only be accessible from inside that routine. We term such variables \textit{local} variables. Observe the role of \texttt{total} in the example below.

\begin{verbatim}
> new_SetAverage := proc( s : set(real) )
>   local total;
>   description 'This procedure takes a set of real values as an argument and calculates the average value in the set';
\end{verbatim}
3.3. SCOPING RULES

> total := 0;
> for i from 1 to length(s) do
>   total := total + s[i];
> od;
> print('The average is ', total/length(s));
> end;

Since variable total is used as a “scratch” variable during the computation of the average, no other procedure would require use of it. Therefore, it is advisable to make it local to the routine. To do this, we declare total to be of type local and now it is visible only within the body of procedure NewSetAverage. This explicit declaration is optional in Darwin. If you compare procedure NewSetAverage with the previous versions of SetAverage, you will see that we dynamically defined total. What does Darwin do when it encounters a name that has not yet been declared? This is an easy question with a complicated answer. The dogma we suggest you adhere to goes something like as follows:

the best way to learn the Darwin scoping rules is through experimentation; the best way to avoid learning the Darwin scoping rules is through always declaring your local variables.

When you begin to write large programs, subtle scoping problems may arise. These can be particularly difficult to pinpoint and remove. The explicit declaration of a local variable helps to document your code and provides some help for Darwin in determining the intent of your program.

We can use the built-in assigned function to test the scope of a variable. The function

assigned(t: name)

takes a parameter t of type name and returns true if t is currently being used as a name in this scope. Otherwise, it returns false.

> assigned(total); # check to see if total is a variable
> # from outside of the procedure
> SetAverage({5, 10, 15}); # call the procedure
> assigned(total);       # check to see if total is a
>                        # variable now from outside the
>                        # proc

Any local declarations must appear directly after the procedure declaration. They may follow or precede global declarations (see §3.3.2 below) but must precede any option commands (§11.3) or the description command (§3.2). If there is more than one local variable, they must be separated by commas.

> example := proc( )
>    local x, y, z;
>    global a, b, c;
>    option polymorphic;
>    description 'A description of the procedure';
> end:

3.3.1 Independent Scopes

Two variables with the same name but with different scopes are independent of each other. Changing the contents of one variable does not effect the contents of the other. The scope of the local variable u in DummyProcedure_1 below is independent of the scope of variable u in DummyProcedure_2.

> DummyProcedure_1 := proc()
>    lprint('Inside procedure 1');
>    lprint('The value of u before assignment: ', u);
>    u := -8000;
>    lprint('The value of u after assignment: ', u);
> end:

> DummyProcedure_2 := proc()
>    lprint('Inside procedure 2');
>    lprint('The value of u before assignment: ', u);
>    u := 999;
3.3. **SCOPING RULES**

> lprint('The value of u after assignment: ', u);
> end:

Note that each time we call either of the above procedures, the variable *u* has no value associated with it initially. Because it does not have a value, Darwin treats *u* as a name in the `lprint` command before the assignment of either -8000 or 999 to *u*. No matter which order or how many times we invoke the `dummy_procedures`, their local copies of the variable *u* remain unaffected by the assignments done to the other.

> DummyProcedure_1();
Inside procedure 1
The value of u before assignment:  u
The value of u after assignment:  -8000
> DummyProcedure_2();
Inside procedure 2
The value of u before assignment:  u
The value of u after assignment:  999
> DummyProcedure_1();
Inside procedure 1
The value of u before assignment:  u
The value of u after assignment:  -8000

### 3.3.2 Examining Global Variables from within Routines

Consider the procedure `TheGood` and the global variable `outside` defined below:

> outside := 1029384756:

> TheGood := proc( )
> lprint('The value of variable outside is: ', outside);
> end:

> TheGood();
The value of variable outside is:  1029384756
Invoking the procedure `TheGood` prints out the value assigned to the global variable `outside`. This shows that the scope of `outside` includes the body of the procedure `TheGood`. We may examine the contents of any global variable as long as neither of the following two conditions are violated in your procedure\(^2\):

1. The global variable does not appear in the left hand side of an assignment.

2. The global variable does not appear as the variable in a `for` loop.
   Example: `for glb from 1 to 10 do ...` where `glb` is a global variable.

If either of these conditions are violated, a local variable of the same name as the global variable is created and any values we assign to the local variable will not persist after the end of execution.

```idl
> outside := 1029384756;
> TheBad := proc( ) 
  # an extension to the_good
  outside := 2;       # We think we're assigning to the
  lprint('Outside: ', outside);  # global copy of outside, but we aren't
> end:

> TheBad();
Outside: 2
> outside;
1029384756
```

Since `outside` appears on the left hand side of an assignment, a local variable also named `outside` is created. If we were extremely careful programmers, we would re-write `TheBad` as follows:

```idl
> TheBetter := proc( )
  local outside;
```

\(^2\)These conditions are verified when you define your procedure.
3.3. *SCOPING RULES*  

> outside := 2;  
> lprint('Outside: ', outside);  
> end:

Now, there is no confusion whatsoever about which `outside` you are referring to when performing the assignment.

### 3.3.3 Modifying Global Variables

Suppose we would like our routine to modify a global variable. Consider the following variable `SmallestSoFar` and procedure `FindMinimum`.

> SmallestSoFar := 10^90;  
  # effectively infinity

> FindMinimum := proc(L : list)  
  > for i from 1 to length(L) do  
  >     if (L[i]<SmallestSoFar) then  
  >         SmallestSoFar := L[i];  
  >     fi;  
  > od;  
  > end:

Since `SmallestSoFar` is assigned $10^{90}$ outside of `FindMinimum`, it is a global variable to procedure `FindMinimum`. By default, we cannot change `SmallestSoFar` inside of `FindMinimum`. Since `SmallestSoFar` participates in the left hand side of an assignment, Darwin creates a new local variable `SmallestSoFar`. Executing this program in its current form results in a *values cannot be ordered* error since Darwin does not know how to compare the element `L[i]` with the uninitialized name `SmallestSoFar` when $i = 1$. To extend the scope of `SmallestSoFar` to include `FindMinimum` we use the `global` statement after the `proc` declaration.

> FindMinimum := proc(L : list)  
  >     global SmallestSoFar;  
  >     (as before)


> end;
Now any change made to SmallestSoFar persists after the end of the FindMinimum procedure.

> print(SmallestSoFar);
9.99999999999999e+89
> FindMinimum([89, 100, 2, 77, 33]);
> print(SmallestSoFar);
2

3.3.4 Modifying the Value of a Parameter

One common mistake many novice programmers make involves attempting to assign to a parameter name which has already been assigned a value. Darwin does not allow a new value to be assigned to the parameter. They may however be assigned a new undefined name. The following two versions of DummyProc exemplify this subtle difference of how Darwin treats a symbol and a value.

> DummyProc1 := proc( inside )
>     inside := 1;                  # assign to inside the value 1
> end:
> outside := 2;
> DummyProc1( outside );        # an error
Error, (in DummyProc1) invalid left hand side in assignment

> DummyProc2 := proc( inside )
>     inside := b;                # assign to inside a symbol b
> end:
> outside := a;
> DummyProc2( outside );        # now outside has symbol b assigned
> print(outside);                # to it.

In the first example DummyProc1, we are attempting to assign the value 1 to the parameter inside. Darwin does not like this and reports an appropriate
3.4. FUNCTIONS

error. In the second example, \texttt{DummyProc2}, we are attempting to assign the
name \texttt{b} to the parameter name. This is entirely legal.

If you are not confused yet, then try to explain the following behaviour:

\begin{verbatim}
> DummyProc3 := proc( inside )
>     bb := 7;
>     inside := bb;  # assign to inside a symbol b
> end:
> outside := aa:
> DummyProc3( outside );  # now outside has symbol b
>     # assigned to it.

7
> print(outside);

aa
\end{verbatim}

Here we have indirectly assigned the value 7 to the parameter \texttt{inside}. However, the change does not persist after the execution of the procedure.

3.4 Functions

A \textit{function} is a procedure which returns a value. We have already seen a
number of built-in Darwin functions in Chapter 1: \texttt{op}, \texttt{union}, \texttt{member},
\texttt{subset}, \texttt{length}, \texttt{mod} amongst others. The difference between \textit{procedures}
and \textit{functions} in Darwin is entirely conceptual. The exact same syntax is
used for defining both and, actually, \textit{procedures} do return a value. Recall
our procedure \texttt{SetAverage} from §3.1.

\begin{verbatim}
> SetAverage := proc( s : set )
>     local total;
>     description 'This procedure takes a set as an argument and
>                  calculates the average value in the set';
>     total := 0;
>     for i from 1 to length(s) do
>         total := total + s[i];
>     od;
\end{verbatim}
The procedure `SetAverage` returns the value `NULL`.

```plaintext
> x := SetAverage({1,2,3});
> if (x=NULL) then print('It is NULL'); fi;
It is NULL
```

Let us re-write `SetAverage` to be a function.

```plaintext
> NewSetAverage := proc( s : set(real) )
>     local total;
>     description 'This procedure takes a set of real values as an
>                     argument and calculates the average value in the set';
>     total := 0;
>     for i from 1 to length(s) do
>         total := total + s[i];
>     od;
>     ave := total/length(s);
>     ave;
> end:
```

Now the last statement executed when `NewSetAverage` is invoked is the bottom-most (trivial) statement `ave;`. The result of this expression is simply the value assigned to this variable. We say that the value of the entire function `NewSetAverage` is assigned the value of `ave` and it is this value which Darwin returns to the point where the function was called.

```plaintext
> x := NewSetAverage({3, 5});          # the value of function New_SetAverage
> print(x);                            # is assigned to variable x
4
```

### 3.4.1 The return Command

In some situations, it is much more convenient to exit a function when a certain condition arises rather than seeping through to the end of the function. To this end, the Darwin language contains the `return(exp)` command;
upon encountering this function within the body of a routine, Darwin immediately returns the result of evaluating expression \textit{exp} to the place where the function was invoked. Consider the following simple test for primality which returns either true or false.

\begin{verbatim}
> IsItPrime := proc (x : real)
>   if (mod(x,2)=0) then  # x is even
>     return(false);
>   else
>     for i from 3 to floor(sqrt(x)) by 2 do
>       if (mod(x, i)=0) then
>         return(false);
>       fi;
>     od;
>     fi;
>   true;
> end;
\end{verbatim}

Here there are three possible exit points from this function. If \textit{x} is even, then the value of the function becomes false. If \textit{x} is odd but it has a odd factor between 3 and the floor of its square root, then it also returns false. Only in the case when \textit{x} is prime does the function seep down to the last statement where its value is assigned true.

\begin{verbatim}
> IsItPrime(10039823478323782);  # it’s an even number
false
> IsItPrime(3478728737);  # odd but divisible by some number
false
> IsItPrime(101);  # it’s a prime number
true
\end{verbatim}

The value of a function is not restricted to being of type \texttt{real} or \texttt{boolean}. In general, functions may return any data type (including other functions). We could extend our (sophisticated) primality tester to return either true or false and one of the prime factors if it exists.
> IsItPrime := proc (x : posint)
> if (mod(x, 2)=0) then          # x is even
>     return([false, 2]);
> else
>     for i from 3 to floor(sqrt(x)) by 2 do
>         if (mod(x, i)=0) then
>             return([false, i]);
>         fi;
>     od;
>     fi;
>     [true,1];
> end;

At all three exit points, we return list structures. This need not be the case as Darwin allows at least two other ways of returning multiple values.

The first alternative is to return them as an expression sequence in which case we would re-write the return statements from IsItPrime as follows

> return(false, 2);      # instead of [false, 2]
> return(false, i);      # instead of [false, i]
> true, 1;               # instead of [true, 1]

We advocate against this method as it is sometimes difficult to handle these expression sequences when subsequently passing them to other functions.

The second alternative is to return the values as a data structure. In Darwin, a data structure is any expression which is syntactically a procedure call but does not evaluate (or it is simply not defined). In this way, we have potentially infinitely many different data structures (we deal with this subject extensively in Chapter 5).

> return(Primality( false, 2 ));  # instead of [false, 2]
> return(Primality( false, i ));  # instead of [false, i]
> return(Primality( true, 1 ));   # instead of [true, 1]
3.5 Nested Routines

A procedure is \textit{nested} if it is declared within the body of another procedure. The scoping rules for such procedures are the same as the scoping rules for local variables. The motivation for such constructs is similar to the motivation for local variables: if a routine is only used by one procedure, then it should not be visible to the remaining program. This results in easier to read code and less troublesome modifications. We present a simple example of a function \texttt{VectorMean} nested inside of a function \texttt{VectorVariance} but note that Darwin allows for an arbitrary depth of nesting.

\begin{verbatim}
VectorVariance := proc( M : array(real) )
    local variance, mean;
    variance := 0;

    VectorMean := proc( M1 : array(real) ) # a nested function
        local total;
        total := 0;
        for i from 1 to length(M1) do
            total := total + M1[i];
        od;
        return(total/length(M1));
    end;

    mean := VectorMean(M);
    for i from 1 to length(M) do
        variance := variance + (M[i] - mean)^2;
    od;
    variance;
end;
\end{verbatim}

The function \texttt{VectorVariance} calculates the variance for a linear \texttt{array} of \texttt{real} elements. In order to carry out this computation, it requires that the mean of the array is calculated. This is performed in the nested function \texttt{VectorMean}. This function accepts an \texttt{array} of \texttt{real} elements and returns
a single real number to the function \texttt{VectorVariance}. Only the body of the function \texttt{VectorVariance} has access to this function.

\begin{verbatim}
> VectorVariance([12, 6, 18, 13, 21, 7, 11, 12, 14]);
180.0000
\end{verbatim}

### 3.6 Variable Number of Arguments

All of the examples of procedures we have seen thus far have taken some fixed number of arguments: \texttt{IsItPrime} accepts a single positive integer; \texttt{VectorVariance} accepts an array of real values. However, it is sometimes convenient to create routines which accept a variable number of arguments of any type. This is especially useful if we do not know \textit{a priori} how many items of data the user may pass to the routine or of which type these items will be.

There are two Darwin system variables made available to the user to enable them to determine the number of parameters passed to the routine and to read the list of parameters passed to the procedure. The first is \texttt{nargs} (number of arguments) of type \texttt{real} and the second is \texttt{args} (arguments) which is an expression sequence containing the entire argument list passed to the routine. A third name \texttt{procname} stores the name of the routine.

\begin{verbatim}
> PrintArguments := proc( )
>   lprint('The name of this procedure is: ', procname);
>   lprint('The number of arguments is: ', nargs);
>   lprint('The arguments are: ', args);
>   lprint();
>   for i from 1 to nargs do
>     lprint('Argument #', i, ': ', args[i]);
>   od;
>   end;

> PrintArguments(1, 'hello', [3,4,5]);
\end{verbatim}
The name of this procedure is: print_arguments
The number of arguments is: 3
The arguments are: 1, hello, [3, 4, 5]

Argument # 1 : 1
Argument # 2 : hello
Argument # 3 : [3, 4, 5]

Notice that the formal argument (or formal parameter) list is empty. All
parameters to this procedure are assigned to the name args.

As an example application, we create a function minimum which accepts
any number of real values and returns the smallest one.

```plaintext
> minimum := proc( )
> local smallest;
> description 'Find the minimum of an arbitrary long list of real values';
> for i from 1 to nargs do
> if not (type(args[i], real)) then
> lprint('An error in function ', procname);
> error('A non real value passed as a parameter');
> fi;
> od;
> smallest := args[1]; # initialize it to the first element of the list
> for i from 2 to nargs do
> if (args[i]<smallest) then
> smallest := args[i];
> fi;
> od;
> return(smallest);
> end:

> minimum(1, 2, 3, 4);
1
```
> minimum(a, 9, 10, 11);  # an error

An error in function minimum
Error, (in minimum) A non real value passed as a parameter

Note that we must perform our type checking inside of the body of the function.
Chapter 4

Lists and Arrays Revisited

As the list data type (or, equivalently, the array data type) tends to be used so often in Darwin, we revisit the subject briefly in this section (see §1.5.2 of Chapter 1 – Exploring the Basics for the initial discussion).

4.1 Zippable Commands

(wait until G. finishes this routine) first example of option

4.2 Sorting

For arrays with elements of an homogeneous type, it is a simple matter to place them in ascending order with Darwin’s sort function.

> numbers := [1971, -1554, 1449, -45, 9000, 2001];
> sort(numbers);
[-1554, -45, 1449, 1971, 2001, 9000]

The list may contain elements of any type; the only requirement is that the elements are comparable, that is, the $<$, $>$, $<=$, $<>$ and $=>$ operators are applicable. For real values, these operators act in the typical manner. For string objects, the ordering is lexicographic. If, instead, we desire the
elements placed in descending order, we must pass a second argument to sort.

\[\text{sort(numbers, } x \rightarrow -x);\]

\[\{9000, 2001, 1971, 1449, -45, -1554\}\]

The choice of the name \(x\) in the above example is irrelevant, any legal Darwin variable name will suffice. The name \(x\) represents one item of the list \texttt{numbers}. In this case, Darwin first negates each element of \texttt{numbers} and then performs the sort operation on the resulting vector.

The \(x \rightarrow -x\) is simply syntactic sugar for a procedure definition. We could re-write this call as follows:

\[\text{neg := proc(x)}\]
\[\text{return(-x);}\]
\[\text{end;}\]
\[\text{sort(numbers, neg);}\]

\[\{9000, 2001, 1971, 1449, -45, -1554\}\]

If we issue

\[\text{sort(numbers, } x \rightarrow -\text{abs(x)});\]

\[\{9000, 2001, 1971, -1554, 1449, -45\}\]

we receive back the vector consisting of the absolute values of \texttt{numbers} in descending order. Now suppose we are given a multi-dimensional array of heterogeneous data such as the following\(^1\)

\[\text{ReadProgram('Sample/arrays')}\];
\[\text{print(classics);}\]

\[\text{[Joseph Felsenstein,}\]

\[\text{Phylogenies from molecular sequences: inference and reliability,}\]

\(^1\text{This list is contained in file Sample/arrays and can be loaded into Darwin by issuing the command ReadProgram('Sample/arrays'). It can also be downloaded from the Computational Biochemistry Research Group web cite [5].}\]
where each entry is composed of an ordered list of six items. The field order is authors’ names, title of paper, journal, year, volume and page numbers.
To sort the list into alphabetical order by authors’ names, we need only specify the correct field of `classics`.

```r
> print(sort( classics, x -> x[1] ));  # (ascending) sort by author’s name
[  
 [Dayhoff, Margaret O. and Schwartz, R. M. and Orcutt, B. C.,  
 A model for evolutionary change in proteins,  
 Atlas of Protein Sequence and Structure, 1978, 5, 345--352]  
 [Joseph Felsenstein,  
 Phylogenies from molecular sequences: inference and reliability,  
 Annual Revue of Genetics, 1988, 22, 521--565]  
]
> print(sort( classics, x -> -x[1] ));  # (descending) sort by
```
(a system bug - must be fixed)
> print(sort( classics, x -> x[4] ));  # sort by year
[
[Needleman, S. B. and Wunsch, C. D., A general method applicable to the search 
[Dayhoff, Margaret O. and Schwartz, R. M. and Orcutt, B. C.,
A model for evolutionary change in proteins,
Atlas of Protein Sequence and Structure, 1978, 5, 345--352]
[Smith, Temple F. and Waterman, Michael S.,
Identification of common molecular subsequences, J. Mol. Biol., 1981, 147, 
195--197]
[Joseph Felsenstein,
Phylogenies from molecular sequences: inference and reliability,
Annual Revue of Genetics, 1988, 22, 521--565]
]

Note, however, that the original copy of the array classics is unchanged. The sort statement returns a new copy of the array.

4.3 Searching

The function SearchArray returns the index of an element in an array if it exists in the array. Otherwise it returns 0.

SearchArray\(t\) :\{string, real\}, \(L :\text{array}\);

> SearchArray\(5\), [1, 2, 7, 5, 8]);
4
> SearchArray\('hi'\), ['hello', 'hallo', 'hey', 'hoi']);
0

The SearchOrderedArray function in Darwin searches an array \(A\) for a target value \(t\) but returns the index \(i\) such that \(A[i] \leq t < A[i+1]\).

SearchOrderedArray\(t\) : \{ string, real \}, \(A : \text{array}\)
4.3. SEARCHING

If the list $A$ is not ordered, the function returns the first such $i$ satisfying the above inequality.

> SearchOrderedArray(5, [2, 4, 6, 8, 10]);
2
> SearchOrderedArray('xianghong', ['chantal', 'gaston', 'mike',
  'ulrike', 'xianghong']);
5
> SearchOrderedArray(5, [10, 8, 6, 4, 2]);  # the list is unordered.
0
CHAPTER 4. LISTS AND ARRAYS REVISITED
Chapter 5

Structured Types

Section 1.5 of Chapter 1 introduced a number of Darwin built-in data types. A complete list of built-in types can be found in Table 5.1. We have also seen how elements from different types can be included in array and set data structures. The array classics from the previous section was an example of such a heterogeneous data structure.

> ReadProgram('Sample/arrays');
> print(classics);

[ Joseph Felsenstein,
  Phylogenies from molecular sequences: inference and reliability,
  Annual Revue of Genetics, 1988, 22, 521--565]

[Smith, Temple F. and Waterman, Michael S.,

[Dayhoff, Margaret O. and Schwartz, R. M. and Orcutt, B. C.,
  A model for evolutionary change in proteins,
  Atlas of Protein Sequence and Structure, 1978, 5, 345--352]

However, as the diversity of data in our programs become large, types such as list and set become increasingly insufficient for maintaining clean, readable programs. Even for the relatively small example of classics, the onus lies on the programmer’s shoulders to remember which field corresponds to which item of information. Most programming languages, including Darwin, offer some manner for creating structured data types to elevate these problems.

A structured data type in Darwin is built using functions. The best way to learn how to define a structured type in Darwin is through examples. Let us begin by defining a new type nonnegint (non-negative integer) which consists of posint (positive integers) and the value 0.

> nonnegint := proc(x : {0, posint})
> return(noeval(nonnegint(x)));
> end:

This function accepts any argument which is either of type posint or has value 0. The type checking is done within the formal parameter list. The body of the function simply returns an unevaluated copy of the the function nonnegint with the parameter x. The command noeval (no evaluation) delays the evaluation of its argument. Darwin simply returns the parameter of noeval as an object of type name. We can define variables of type nonnegint by calling the function with the appropriate argument. Observe what happens when the argument is not of the correct type.

> a := nonnegint(0);
a := nonnegint(0)
> b := nonnegint(5);
b := nonnegint(5)
> c := nonnegint(1388293823);
c := nonnegint(1388293823)
> d := nonnegint(-1);
nonnegint expects a 1st argument, x:{0,posint}, found: -1
Error, invalid arguments

A variable (an instance) of a structured type is simply an unevaluated procedure call in Darwin. There are two ways a procedure call can remain
unevaluated. The first is through the use of the `noeval` command described above. When we wrap the procedure call in a `noeval` command, Darwin just returns the procedure call and contents as is.

```plaintext
> delayed := noeval(factorial(5));
delayed := factorial(5)
> frustrated := noeval(print('I so desperately want to simplify'));
frustrated := print(I so desperately want to simplify)
```

The second way a procedure call will remain unevaluated is when the procedure name is undefined in the scope. ¹

```plaintext
> any := thing_goes(1, 2, 3);
any := thing_goes(1, 2, 3)
> I := have_unlimited('freedom');
I := have_unlimited(freedom)
> to_do := as_I_like(abc, 123, ['a', 'list']);
to_do := as_I_like(abc, 123, [a, list])
```

There are no procedures named `thing_goes`, `have_unlimited` and `as_I_like` defined so Darwin simply assigns the unevaluated name and parameters to the variables `any`, `I` and `to_do`. What type of data do these structured types allow? The answer is, literally, anything.

The variables which we create in this manner are assigned the type `uneval` (the values assigned to them look like unevaluated procedure calls).

```plaintext
> type(any, uneval);
true
> type(I, uneval);
true
> type(to_do, uneval);
true
```

¹It is advisable that you exit your current Darwin session and restart a fresh one. The names used below must not be assigned in your current session.
The manner in which one goes about defining structured types may seem a little abstract and, if you are an experienced programmer, much different than what you are used to with languages such as C or Pascal. But bear with us: this method of structured type creation turns out to be extremely versatile. As in our example \texttt{nonnegint}, we can place an unlimited amount of code in the procedure which defines a type. This allows us to perform an arbitrary amount of type checking and manipulation of data.

Darwin comes equipped with several built-in structured types. Table 5.2 contains a sample of the most used structures. Chapter 6 -- \textit{Iteration and Recursion} shows an application using the \texttt{Tree} structure. The remaining entries are explored in greater depth in Part II -- \textit{Darwin and Problems from Biochemistry}.

5.1 Defining New Structured Types: An Example

As a concrete example of how to go about creating a new structured data type, we will build a structure to hold a protein sequence and some information about it. In particular, our data type should have fields to contain the name of the sequence, which database the sequence is located in, the accession number of the sequence, the organism from where it was derived, the length of the protein sequence and, of course, the protein sequence itself. We begin by creating a data type \texttt{ProEntry} as follows.

\begin{verbatim}
> ProEntry := proc( )
>  description
>  ' Data type to contain information about a protein sequence
>  Parameters:
>  Name, DB, AC, Organism, Sequence, Length
>  ';

> if ( nargs=0 ) then
>  return(noeval(ProEntry('', '', '', '', '', -1)));
>
> elif ( nargs=6 ) then
\end{verbatim}
> return(noeval(ProEntry(args)));
> else
> print( ProEntry );
> error('Invalid ProEntry format');
> fi;
> end:

Let us follow through this definition line by line.

Recall that \texttt{nargs} is a system name within the scope of a routine which contains the number of arguments passed to it on that invocation. The \texttt{args} name is assigned the expression sequence consisting of all arguments passed to the routine.

The body of \texttt{ProEntry} consists of a single \texttt{if-then-fi} clause. If \texttt{ProEntry} is called with no arguments, Darwin creates a data structure \texttt{ProEntry} and initializes the six fields appropriately. The \texttt{noeval} statement delays Darwin from attempting to evaluate \texttt{ProEntry} by invoking it as it would normally do when it encounters a procedure or function name. Instead upon encountering a \texttt{noeval} command, Darwin treats the whole segment in the same manner it would treat a expression sequence and simply returns it.

If \texttt{ProEntry} is called with exactly six arguments, Darwin creates a new data structure \texttt{ProEntry}, it assigns the $i$th entry of \texttt{args} to the $i$th element of \texttt{ProEntry} for $i$ from 1...6, and returns it unevaluated.

If \texttt{ProEntry} is called with any other number of arguments, the \texttt{description} header for the structured type is issued and an error is reported (see Chapter 12 for an explanation of the \texttt{error} function).

We are now in a position to create a structure of type \texttt{ProEntry}. There are at least two ways to accomplish this: we can either pass the items of data for a protein sequence as arguments to \texttt{ProEntry} or we may create an empty \texttt{ProEntry} structure and assign to the elements of \texttt{ProEntry} ourselves.\footnote{The sequence for the Swiss-Prot entry given below has been abbreviated to the first 42 amino acids for space reasons.}
> # The first method

> protentry1 := ProEntry('ABL1_CAEL', 'SwissProt', 'P03949',
>                     'C. ELEGANS',
>                     'NNEWCEARLYSTRKNDASNQRLGEIGWVPNFIAPYNSLDK', 42);

protentry1 := ProEntry(ABL1_CAEL, SwissProt, P03949, C. ELEGANS,
NNEWCEARLYSTRKNDASNQRLGEIGWVPNFIAPYNSLDK, 42)

> # The second method

> protentry2 := ProEntry(); # first initialize an empty
# structure

protentry2 := ProEntry(, , , , -1)

> protentry2[1] := 'ABL1_CAEL';
> protentry2[5] := 'NNEWCEARLYSTRKNDASNQRLGEIGWVPNFIAPYNSLDK';
> protentry2[6] := 42;
> print(protentry2);
ProEntry(ABL1_CAEL, SwissProt, P03949, C. ELEGANS,
NNEWCEARLYSTRKNDASNQRLGEIGWVPNFIAPYNSLDK, 42)

Now we may examine each item of data contained in protentry1 by
indexing into it in the same manner we would index into the elements of an
array.

> protentry1[1]; # the name of the entry
ABL1_CAEL
> protentry1[2]; # the accession number for the entry
SwissProt
> protentry1[3..5]; # organism, sequence, length
P03949, C. ELEGANS, NNEWCEARLYSTRKNDASNQRLGEIGWVPNFIAPYNSLDK
5.1. DEFINING NEW STRUCTURED TYPES: AN EXAMPLE

Of course, we can define as many structures of type \texttt{ProEntry} as we would like and each structure will have the same format. This is already a marginal improvement over using, for example, the \texttt{list} type to manage our data. However, the real power manifested in Darwin’s structured types still lies ahead. We can improve our type definition to include three powerful tools which reduce the amount of error, simplify our programs and help keep our data stored in a uniform fashion. We modify type \texttt{ProEntry} throughout the next three subsections to include these features.

5.1.1 Error Checking

We can code error checking into our type definitions in two ways: (1) \textit{type checking} and (2) \textit{data verification}.

\textit{Type Checking:} Since our function declaration for \texttt{ProEntry} allows for a variable number of arguments we can not implement \textit{type checking} in the manner described in Chapter 3 \textit{Procedures} where we included the type of the parameter after the name of the parameter in the \texttt{proc} declaration line. Instead, we must delay checking until the body of the \texttt{proc}. We modify our definition of \texttt{ProEntry} so that it verifies that the information passed to it is of the correct type.

\textit{Data Verification:} Certainly, the calculation of the \texttt{Length} field would be better carried out by a computer than by an assistant with better things to do with his or her time. The computer would also be less prone to error (and the monitor screen free of fingerprints!). We can make \texttt{ProEntry} more robust by exploiting the flexibility the \texttt{args}, \texttt{nargs} built-in variables offer by allowing \texttt{ProEntry} to also accept only five arguments without \texttt{Length}. This last field will be calculated within the \texttt{ProEntry} function. If the user insists on supplying the length of the protein sequence, our procedure will merely check their work and report a warning if it does not match that supplied by the user.

\begin{verbatim}
> ProEntry := proc( )
>     description
\end{verbatim}
' Data type to contain information about a protein sequence
Parameters:
    Name : string ('' indicates field is uninitialized)
    DB   : string ('' indicates field is uninitialized)
    AC   : string ('' indicates field is uninitialized)
    Organism : string ('' indicates field is uninitialized)
    Sequence : string ('' indicates field is uninitialized)
    Length : integer (-1 indicates field is uninitialized)
'
if (nargs=0) then
    return(noeval(ProEntry('', '', '', '', '', -1)));
fi;

if not (type( args[1], string) and type(args[2], string)
    and type( args[3], string) and type(args[4], string)
    and type( args[5], string)) then
    print( ProEntry );
    error('Invalid types in ProEntry format');
    fi;

if (nargs=6) then
    if not (type(Length, integer)) then
        print(ProEntry);
        error('Invalid types in ProEntry format');
    else
        if (args[6]<length(args[5])) then
            print('Warning: you counted the length incorrectly', args);
        fi;
        return(noeval(ProEntry(args)));
    fi;
elif (nargs=5) then
    return(noeval( ProEntry(args, length(args[5]))));
else
5.1. DEFINING NEW STRUCTURED TYPES: AN EXAMPLE

> print( ProEntry );
> error('Invalid number of parameters to ProEntry');
> fi;
> end:
>
> protentry1 := ProEntry('ABL1_CAEEL', 'SwissProt', 'P03949',
> 'C. ELEGANS', 'NNEWCEARLYSTRKNDASNQRLGEIGWVP NSFIA PYNSLDK');

protentry1 := ProEntry('ABL1_CAEEL', 'SwissProt', 'P03949', 'C. ELEGANS',
'NNEWCEARLYSTRKNDASNQRLGEIGWVP NSFIA PYNSLDK', 42)

We could extend our error checking in several ways. For example, it would
be prudent to verify that each character of the Sequence field is one of the
twenty amino acids. We omit these straightforward modifications here due
to space considerations.

5.1.2 Simplification

Entries in data structures will at times contain incompletely resolved ex-
pressions. Simplification is the (partial or complete) transformation of an
expression into its simplest form. For instance, a structure for a data type
designed to hold complex numbers might contain Complex(1, 1) where the
first field represents the real part of the number and the second field repre-
sents the imaginary part. Here the data is represented in an overly verbose
manner and we could reduce it to the real number 1.

As a slightly artificial example in our protein sequence setting, suppose
you are creating an amino acid ProEntry data structure and the sequence
has been split into segments corresponding to, say, domains. Suppose fur-
thermore that you do not want to keep this extra information and would
rather simply represent the sequence as a simple string item. For example,
instead of the list seq

> seq := ['NNEWCEAR', 'LYSTRKNDASNQRL', 'GEIGWVP NSFIA PYNSLDK'];

a string sequence would suffice.
CHAPTER 5. STRUCTURED TYPES

\[\text{string_seq := 'NNEWCEARLYSTRKNDASNQRRRLGEIGWPSNFIAPYNSLDK'};\]

We could re-write ProEntry to accept a list or a string as the fifth parameter. We modify the if-then-fi that checks the type, add statements to make the simplification and return the appropriate modified structure.

\[\text{ProEntry := proc( )}\]

\[\quad\text{(as before)}\]
\[\quad\text{if not (type(args[1], string) and type(args[2], string)}\]
\[\quad\quad\text{and type(args[3], string) and type(args[4], string)}\]
\[\quad\quad\text{and type(args[5], list(string)) then}\]
\[\quad\quad\text{simple_seq := '';}\]
\[\quad\quad\text{for i from 1 to length(args[5]) do}\]
\[\quad\quad\quad\text{simple_seq := simple_seq.args[5][i];}\]
\[\quad\quad\text{od;}\]
\[\quad\text{fi;}\]

\[\quad\text{(as before)}\]
\[\quad\text{if (type(args[5], list(string))) then}\]
\[\quad\quad\text{return(noeval(ProEntry(args[1..4], simple_seq, length(simple_seq))));}\]
\[\quad\text{else}\]
\[\quad\quad\text{return(noeval(ProEntry(args[1..5], length(args[5]))));}\]
\[\quad\text{fi;}\]

\[\quad\text{(as before)}\]
\[\quad\text{end:}\]
5.2. NESTED TYPES

5.2.1 Normalization

In order to determine equality/inequality between two structures, there must be a normalized form for the data. Is the number 01 equal to 1? Treated as strings, they certainly are not. However, if we consider them to be integers, then they certainly are. Enforcing strict rules about how data should be stored in structures results in higher quality data which minimizes redundancy and chance of errors.

Returning to our protein sequence example, suppose we were collecting protein entries from a large number of sources. Some of these sources refer to the protein database SwissProt [3] as “SP” while others write “sprot” or possibly “Swiss”. If we know a priori the alternative values for a field, we can transform all entries into one normalized form.

```plaintext
> if ((args[2]='SP') or (args[2]='sprot') or (args[2]='Swiss')) then
>   DB_name := 'SwissProt';
> fi;
```

This should be done in the analogous manner for all fields Name, Accession, Organism and so forth of our structured type. The return statements should now return the modified forms of each element of args. For example,

```plaintext
> return(noeval(ProEntry(Name, DB_name, Accession, Organism_name, Seq, Length)));
```

5.2 Nested Types

The ability to nest structured types within structured types provides the flexibility to keep your data organized in an intuitive, easily understood format. A good structured type will reflect the natural groupings suggested by the data and such an organization will reduce the number of errors, increase readability and make future changes much easier.

In our protein sequence example, the fields Name, DB and the accession number AC are in some sense related items of data since they specify an entry in a genomic database. As such, we might be motivated to grant them their own structured type GenomicDB nested within ProEntry.
ProEntry := proc( )
    description
        'Structured type to contain information about a protein sequence'
    Parameters:
        Name, DB, AC, Organism, Sequence, Length
    ;
    if ( nargs=0 ) then
        temp := GenomicDB();
        return(noeval(ProEntry(GenomicDB('' , '' , '' , '' , '' , -1))));
    elif ( nargs=6 ) then
        return(noeval(ProEntry(GenomicDB(args[1..3]), args[4..6])));
    else
        print( ProEntry );
        error('Invalid ProEntry format');
    fi;
end:

We define a structure of type ProEntry as before. Notice the difference in how the information is stored internally.

> p := ProEntry('ABL1_CAEEL', 'SwissProt', 'P03949', 'C. ELEGANS',
                'NNEWCEARLYSTRKNDASNQRRLGEIGWPSNFIAPYNSLDK', 42);
> protentry1 := ProEntry(GenomicDB(ABL1_CAEEL,SwissProt,P03949),
                  C. ELEGANS, NNEWCEARLYSTRKNDASNQRRLGEIGWPSNFIAPYNSLDK,42)

5.3 Selection on Data Structures

In §5.1, we discussed some of the advantages structured types offer over lists and sets for storing data. Because instances of a structured type are created by calling a Darwin function which returns an unevaluated function as the structure, we are able to do extensive error checking, simplification and normalization. However, both simple types and the structured type ProEntry defined in §5.1 still have the undesirable property that we must remember which index of the structure corresponds to which field of the
5.3. SELECTION ON DATA STRUCTURES

In our basic example ProEntry, the first index is the name of the entry, the second is database where the entry was found, the third is the accession number into this database, and so forth.

> proent := ProEntry('ABL1_CAEEL', 'SwissProt', 'P03949', 'C. ELEGANS',
> 'NNEWCEARELYSTRKNDASNQRLGEIGWPSNFIAAPYNSLX', 42);
> proent[1];                   # retrieve the name of the sequence
ABL1_CAEEL
> proent[4];                   # retrieve the organism name
C. ELEGANS
> proent[6];                   # retrieve the length of the sequence
42

Darwin offers an alternative method called selection for accessing the contents of a structure. Essentially, selection allows us to assign a name to a field of a type in much the same way we assign names to variables and routines. To accomplish this, we create a function named ProEntry.select (in general, the naming convention is the name of the structure type, followed by an underscore symbol (_), followed by the Darwin keyword select). The keyword select is a built-in name with special polymorphic status (for more details on polymorphism see Chapter 10 – Overloading, Polymorphism and Object Orientation).

> ProEntry.select := proc(u, sel, val)
>     upper_sel := uppercase(sel);
>     if ( upper_sel = 'NAME' ) then
>         if ( nargs = 3 ) then u[1] := val else u[1] fi;
>     elif ( upper_sel = 'DB' ) then
>     elif ( upper_sel = 'AC' ) then
>     elif ( upper_sel = 'ORGANISM' ) then
>     elif ( upper_sel = 'SEQUENCE' ) then
> end proc;
>  elif ( upper_sel = 'LENGTH' ) then
>  fi;
>  end:

Here u is a structure of type ProEntry, sel is the name of the field we wish to access, and val is the value we would like to assign to field u[sel]. A function call such as

>  proent['DB'] := 'GenBank';

is equivalent to the call

>  ProEntry_select( proent, 'DB', 'GenBank' );

This statement modifies the contents of field 'DB' (the second index) of proent. To examine the contents of a field from proent we omit the third argument val. In this case val is empty and the system name nargs is assigned the value two. ProEntry_select only returns the contents of the field labeled 'DB'.

>  proent['Names'];
>  proent['DB'];
>  proent['DB'] := 'GenBank';
>  proent['DB'];

>  proent['organism'];
>  proent['organism'] := 'H. influenzae';
>  proent['organism'];
Be careful! The following two statements are not strictly semantically equivalent.

\[\text{> protent}[6] := 42;\]
\[\text{> protent}['\text{Length}'] := 42;\]

although they both assign values to the sixth element of the structure \text{protent}. The latter statement invokes the \text{ProEntry.select} function while the former statement does not. This means that whatever functions \text{ProEntry.select} performs above and beyond the assignment of 42 to the field \text{Length} (the sixth index) to \text{protent} will not be carried out by the assignment \text{protent}[6] := 42.

5.4 The Types of Structured Types

You can test whether a \text{name} has been assigned to a \text{procedure} as follows:

\[\text{> example := proc( ) } \quad \# \text{an empty procedure}\]
\[\text{> end:}\]
\[\text{> type(example, procedure)};\]

A valid Darwin name has type \text{name}.

\[\text{> type(example, name)};\]
\[\text{> type(a_valid_Darwin_name, name)};\]

An unevaluated procedure call has type \text{uneval}. The spirit of \text{uneval} comes from the fact that objects of this type “look like” procedure calls. The difference between objects of type \text{procedure} and type \text{uneval} is that the former evaluates while the latter does not.

Recall that there are two ways a procedure call will remain unevaluated in Darwin: either the procedure name is wrapped in a \text{noeval} command or the name is undefined in the environment.
> z := noeval(example());
> type(z, uneval);
> type(any_undefined_name(), uneval);

An object has type **structure** if it is either a built-in Darwin structured type or it is of type **uneval**.

> type(Gene, structure);  # the built-in structured type to hold a gene
> type(Tree, structure);   # the built-in structured type to hold a tree
> type(MultiAlign, structure); # the built-in structured type to hold a multiple align

> x := new_type();       # x is a variable of a new user defined type
> type( x, uneval );     # therefore it has type uneval
> type( x, structure );  # and type structure

If we create a new structured type such as **NewStruct** and a variable of this type

> x := NewStruct(a, 5, {1, 2, 3});

the type of x is only the general class noeval. We can explicitly make **NewStruct** a structured type recognized by Darwin by assigning the type name **NewStruct** the specification as an unevaluated function of type **anything**.

> NewStruct_type := specuneval(anything, NewStruct);

Now we may test whether the type of x is **NewStruct**.

> type(x, NewStruct);

We could encode stricter type checking into **NewStruct** by replacing the **anything** declaration in the **specuneval** command with a sequence of types.

> NewStruct_type := specuneval(string, integer, set, NewStruct);
<table>
<thead>
<tr>
<th>Type Name</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>constant</td>
<td>Any number or variable assigned a number.</td>
<td>1, 33, 203, 39293</td>
</tr>
<tr>
<td>boolean</td>
<td>boolean</td>
<td>true, false</td>
</tr>
<tr>
<td>real</td>
<td>Real numbers.</td>
<td>1, 1.1, 1.01, 1.001,...</td>
</tr>
<tr>
<td>integer</td>
<td>Integers.</td>
<td>... – 2, –1, 0, 1, 2...</td>
</tr>
<tr>
<td>posint</td>
<td>Integers greater than zero</td>
<td>1, 2, 3,...</td>
</tr>
<tr>
<td>string</td>
<td>Any sequence of symbols surrounded by single quotes</td>
<td>'hello'</td>
</tr>
<tr>
<td>anything</td>
<td>Any built-in type</td>
<td>posint, string</td>
</tr>
<tr>
<td>unequal</td>
<td>An unevaluated procedure call.</td>
<td>undefined()</td>
</tr>
<tr>
<td>procedure</td>
<td>A Darwin routine</td>
<td>mod, sin, exp</td>
</tr>
<tr>
<td>equation</td>
<td>A sequence of the form ( expression = expression )</td>
<td>( y = 5, x = z )</td>
</tr>
<tr>
<td>range</td>
<td>( x \ldots y ) where ( x, y ) are of type real</td>
<td>1..100</td>
</tr>
<tr>
<td>list</td>
<td>an ordered multiset (surrounded by [ , ] symbols)</td>
<td>[a, b, c]</td>
</tr>
<tr>
<td>array</td>
<td>an unordered set (surrounded by {, } or ( * ) symbols)</td>
<td>{1, 2, 3}</td>
</tr>
<tr>
<td>set</td>
<td>a two-dimensional square array</td>
<td>[[1, 2], [3, 4]]</td>
</tr>
<tr>
<td>database</td>
<td>A Darwin sequence database</td>
<td>DB</td>
</tr>
<tr>
<td>grid</td>
<td>A Darwin grid file</td>
<td>g where ( g = \text{CreateGrid}() );</td>
</tr>
<tr>
<td>structure</td>
<td>A built-in Darwin structured typed</td>
<td>Tree, Match, Gene</td>
</tr>
<tr>
<td>symbol</td>
<td>A sequence of symbols.</td>
<td>x, xyz, hello_there</td>
</tr>
<tr>
<td>name</td>
<td>A legal Darwin name.</td>
<td>abc, a1, x</td>
</tr>
</tbody>
</table>

Table 5.1: A complete list of built-in Darwin types.
<table>
<thead>
<tr>
<th>Type Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Match</td>
<td>Used for pairwise alignments.</td>
</tr>
<tr>
<td>NucPepMatch</td>
<td>Used for matching DNA with AA sequences</td>
</tr>
<tr>
<td>Gene</td>
<td>Used to hold a gene and annotation information</td>
</tr>
<tr>
<td>Tree</td>
<td>Used to hold a tree structure</td>
</tr>
<tr>
<td>Stat</td>
<td>Used in the collection of statistical information.</td>
</tr>
<tr>
<td>PatEntries</td>
<td>Used in the construction of patricia trees.</td>
</tr>
<tr>
<td>Graph</td>
<td>Used to hold combinatorial graphs.</td>
</tr>
<tr>
<td>Domain</td>
<td>Used to hold a domain structure</td>
</tr>
<tr>
<td>MultiAlign</td>
<td>Used to hold sequence for a multiple alignment.</td>
</tr>
</tbody>
</table>

Table 5.2: A list of built-in Darwin structured types.
Chapter 6

Iteration and Recursion

This chapter explores the different methods available to users for looping over a segment of code. In §1.9 of Chapter 1, we introduced the first such method: the for loop. We briefly revisit this construct here and show some alternative forms for this command. We introduce the second method of iterating in Darwin, the while loop, and give some short examples of its application.

We also discuss another powerful method of repeatedly executing code called recursion. When a procedure calls itself, either directly or indirectly, it is said to be making a recursive call. We show two examples of how recursion can be exploited to solve problems. The first of these computes a factorial via the divide and conquer method. The second example manipulates binary trees, a data structure used many times over in later chapters.

6.1 Iteration

Recall that a statement in Darwin is a command followed by either a semicolon or a colon. A sequence of statements is called a command sequence and consists of a sequence of commands separated by semicolons or colons. The separator after the last command is optional. The body of a loop consists
of a code segment surrounded by the Darwin built-in commands do and od. We can create a simple loop using these commands.

```plaintext
do
  print('I need a terminating symbol after me');
  print('I do too but the next command doesn\'t');
  print('Hit the <ctrl - c> sequence')
od;
```

### 6.1.1 The for Command

The syntax of the for command in Darwin is as follows

```plaintext
for var {from bgn} { to finish } {by incr} do
  { break }
  { next }
od
```

The brace symbols { and } denote optional code. If the from bgn is not included in the for command definition, the value 1 is assumed as the starting point. If to finish is omitted from the declaration, the for loop does not terminate unless a break is encountered. If the by incr is not included in the command, the value 1 is assumed as the increment incr. Initially, the variable var is assigned the value bgn. The body of a for loop is executed if and only if the value assigned to variable var is less than or equal to finish. After the last statement of the body is executed, the variable var is assigned the value var + incr each time through the loop. If var remains less than or equal to finish, the body is executed once again.

```plaintext
> for i from 1 to 10 by 1 do
  >   printf('%d ', i);
  > od;
1 11 21 31 41 51 61 71 81 91
```

The built-in Darwin name break immediately ends execution of the for loop regardless of the stage of execution. Encountering a next command in the
6.1.  ITERATION  

body causes the remaining statements (those following the next command to the end of the body) not to be executed. Control is sent back to the first line of the for loop where Darwin decides to execute the body if the incremented value in var is still less than or equal to higher.

The following loop never unravels in its entirety since a break command is encounter when i is assigned the value 3. Note that implicitly the for loop begins counting at 1.

```plaintext
> for i to 5 do          # equivalent to: for i from 1 to 5 do
  > print(i);
  > if (i=3) then break; fi;
> od;
1
2
3
```

The body of the following for loop is executed exactly four times in the following example.

```plaintext
> for i from 100.1 to 94 by -2 do
  > next;                        # no output is generated.
  > lprint(i);
> od;
```

It is possible to assign to the variable var within the body of the loop. However, this is considered poor programming style and can result in difficult to understand and debug programs.

```plaintext
> for i to 2 do
  > i := 1;                      # an assignment to the loop variable
  > od;
<infinite loop>
```

There is an alternative second syntax for the for loop. It allows one to traverse through a list, array or set easily.
for var in obj do
    { break }
    { next }
od

Here obj must be either a set, list, or array.

> for i in {1, 2, 3, 17, 19, 20} do
>     printf('Amino Acid: %s\n', IntToA(i));
> od;
Amino Acid: A
Amino Acid: R
Amino Acid: N
Amino Acid: T
Amino Acid: Y
Amino Acid: V

> for j in ['A', 'C', 'G', 'T', 'U'] do
>     printf('Nucleic Acid: %s\n', (IntToNucleic(NToInt(j))));
> od;
Nucleic Acid: Adenine
Nucleic Acid: Cytosine
Nucleic Acid: Guanine
Nucleic Acid: Thymine
Nucleic Acid: Uracil

6.1.2 The while Statement

The syntax of the while command in Darwin is as follows

\[
\text{while } \text{conditional boolean expression} \text{ do}
    \{ \text{break} \}
    \{ \text{next} \}
\od
\]
6.1. ITERATION

The while loop executes until the conditional boolean expression evaluates to false or it encounters a break command within the body of the loop. If it encounters a next command within the body of the loop, then the conditional boolean expression is evaluated immediately and if it still evaluates to true, the body is executed again starting at the beginning.

> i:=0;
> while (i<5) do
>     i:=i+1:  # increment i
>     if (i=3) then next; fi;
>     lprint(i);
> od:
>     1
>     2
>     4
>     5

> while (true<>false) do
>     if true then break; fi;
>     # or is it?
> od;

> i:=1:
> while (true) do
>     lprint(i);
>     if (i=10) then break; fi;
>     i:=i+1:  # a way to execute from a while loop
>     # at any point in the body
> od:
>     1
>     2
> ...
>     10

We could exit a while loop via a return command (§3.4 of Chapter 3) if the loop appears in a procedure. Remember, however, that the return exits
the current procedure. The `break` command only exits the loop.

```plaintext
> HarshBreakup := proc()
>   i := 1;
>   while (i < 4) do
>     if (i=3) then
>       return();
>     fi;
>     i := i + 1;
>   od;
>   print('The very last line');
> end:

> HarshBreakup();
<no output is generated>
```

### 6.2 Recursive Calls

When a routine calls itself either directly or indirectly, it is said to be making a **recursive call**. Recursion is the third technique for repeatedly executing a command sequence. It, however, does not require any additional built-in Darwin commands. The basic idea behind solving problems via recursion is to break the instance of the problem into smaller and smaller pieces until the pieces are so small they can be solved trivially. We can view a recursive routine as consisting of two parts.

- The recursive case. When the instance of the problem is still too large to solve directly, we split the problem into two or more smaller parts. It must be the case that the entire problem can be solved if each of the subproblems are solved.

- The basis (or ground) case. When the instance of the problem is small enough to be solved directly, we solve it. If the routine is a function, return the result of this subproblem to the function which called it.
6.2. RECURSIVE CALLS

A necessary condition for recursion to work correctly is that progress must always be made at each invocation of the routine. If no progress is made, then the recursion will continue to dive infinitely deeper (and you will eventually run out of memory).

It is required by law that every programming language manual use the factorial function in an example. We give ours here. Recall the definition of the factorial of a positive integer:

\[
\text{factorial}(n) = \begin{cases} 
  n \cdot n - 1 \cdot n - 2 \cdots 2 \cdot 1 & \text{if } n > 0 \\
  1 & \text{if } n = 0
\end{cases}
\]

We could re-express this function in a more recursive style. This will help with the transition from the mathematical definition to the Darwin routine.

\[
\text{factorial}(n) = \begin{cases} 
  n \cdot \text{factorial}(n - 1) & \text{if } n > 0 \\
  1 & \text{if } n = 0
\end{cases}
\]

Here the basis case of the function is when \( n = 0 \). We simply return 1. The recursive case splits the problem into two cases \( n \) and \( \text{factorial}(n-1) \).

It is not hard to see that multiplying \( n \) with the result of the subproblem \( \text{factorial}(n-1) \) will solve our original problem. The basis and recursive cases in our Darwin function correspond directly to the basis and recursive cases in the above equation.

```plaintext
> factorial := proc ( n )
>   if (n=0) then
>     return(1);
>   else
>     return(n * factorial(n-1));
>   fi;
> end:
>
> factorial(0);
1
> factorial(1);
```


1
> factorial(5);
120
> factorial(100);
9.3326215443944102e+157

Attempting to pass too large an integer to \texttt{factorial} may produce a rather extreme reaction from Darwin and the operating system you are working with. This is because the \textit{depth} of recursion (the number of recursive calls) becomes so large, the system runs out of memory.\footnote{There exists a built-in Darwin command \texttt{factorial(n)} (or alternatively \texttt{n!}). When the integer parameter is sufficiently large, Darwin approximates the result using the function a variant of the \textit{gamma} function [1].}

Recursion is especially convenient for working with \textit{phylogenetic trees}. Although there are many different variants of phylogenetic trees, the basic idea behind all of them is to represent the ancestral relationships between a set of taxa. We take this opportunity to introduce the Darwin data types \texttt{Tree} and \texttt{Leaf} as we will be working with them extensively in Chapter 24 – \textit{Phylogenetic Trees} of Part II. We can represent a phylogenetic tree for a set of species as a complete binary tree\footnote{Biologists sometimes use the term \textit{bifurcating trees} instead of the mathematical \textit{binary tree}.} with the \textit{vertices} of the tree (the points or nodes) representing species and the \textit{edges} (lines that connect points) representing ancestral relationships.

There are two types of \textit{vertices}:

- The leaves. These vertices have only one edge incident with them. When used to model phylogenetic trees, leaves typically represent extant species.

- The internal vertices. Complete binary trees come in two flavors: rooted and unrooted. In an unrooted complete binary tree, every internal vertex has two children and one parent. In a rooted complete
Figure 6.1: A rooted phylogenetic tree formed from the SH2 sequence for six species.
binary tree, every internal vertex has two children and one parent except for one distinguished vertex which does not have a parent. This sole vertex is referred to as the root. When used to model phylogenetic trees, the internal vertices typically correspond to extinct species.

These trees capture the biologic notion of speciation events. In Figure 6.1, a speciation event occurred in the ancestor of all fish (represented by the Swordfish X. Helleri) and the ancestor of amphibians and mammals. Next, a speciation occurred in the ancestor of the amphibians (represented by the frog here) and the ancestor of mammals and the chicken. Then comes the speciation between the ancestor of mammals and chicken, followed by a speciation of humans and the ancestor of mice and rats and, lastly, the divergence of mice and rats.

In Darwin we can define tree structures using the types Tree and Leaf. A Tree structure consists of a left child, a label and a right child. The left and right children consist of either another Tree structure or a Leaf structure. A Leaf structure contains only a label. The following code creates a Darwin tree for the tree shown in Figure 6.1.

```plaintext
> 1_1_1_1 := Tree( Leaf( 'SH30' ), 'A', Leaf( 'SH29' ) );
> 1_1_1 := Tree( 1_1_1_1, 'B', Leaf( 'SH36' ) );
> 1_1 := Tree( 1_1_1, 'C', Leaf( 'SH20' ) );
> 1 := Tree( 1_1, 'D', Leaf( 'SH23' ) );
> root := Tree( 1, 'root', Leaf( 'SH14' ) );
```

This code can be found in file Samples/tree or from the Computational Biochemistry Research Group web site [5]. This tree is a phylogenetic tree formed in Darwin for a set of sequences containing the SH2 domain. We will be using the SH1, SH2 and SH3 domains repeatedly throughout the latter chapters of this book. The following list (also contained in Samples/tree) maps labels for the sequences to the name of the organism from which they came.

```plaintext
> SH2toOrganism := [['SH20', 'CHICKEN'], ['SH30', 'MOUSE'],
```
6.2. RECURSIVE CALLS

> ['SH36', 'HUMAN'], ['SH29', 'RAT'], ['SH14', 'SWORDFISH'],
> ['SH23', 'FROG']:

Let us first design a recursive procedure to print out the tree in a more readable format.

```
FindOrganism := proc(label : string)
    global SH2toOrganism;
    for i from 1 to length(SH2toOrganism) do
        current := SH2toOrganism[i];
        if (current[1]=label) then
            return(current[2]);
        fi;
    od;
    return('Species not know');
end;
```

```
PrintTree := proc (T : {Leaf, Tree} )
    if (type(T, Leaf)) then  # The basis case: T is a leaf
        printf('%s', FindOrganism(T[1]));  # print out the organism name
    else
        printf(' [ ');  # The recursive case: T is a
        PrintTree(T['Left']);  # tree with a left and right
        printf(' , ');  # child. Recurse on both.
        PrintTree(T['Right']);
        printf(' ] ');  
    fi;
end;
```

```
```

The recursive case in `PrintTree` occurs when T is a Tree structure. Since phylogenetic trees are complete binary trees, T must have both a left and
right subtree. We call \texttt{PrintTree} with the left subtree as a parameter and then we call \texttt{PrintTree} with the right subtree as a parameter. The basis case in our recursive procedure \texttt{PrintTree} occurs when the parameter \texttt{T} consists of only a \texttt{Leaf} structure. In this case it passes the label of the \texttt{Leaf} to the \texttt{FindOrganism} function. This function matches the \texttt{Leaf} label to the corresponding organism name found in \texttt{SH2toOrganism} and returns it to \texttt{PrintTree}.

We will see many more examples of recursion with Darwin trees in later chapters. It can be difficult to follow (never mind write) a recursive routine at first but the effort is worthwhile. What may take many lines of code to do iteratively, may take only a few lines recursively. In this way, recursion can offer a crisp elegant perspective of many routines.
Chapter 7

Input/Output

In this chapter, we discuss the file handling routines available in Darwin. There are three different types of files Darwin recognizes.

The first of these is the standard file which contains Darwin programs, data structures or variable assignments. They can be accessed via the ReadProgram and ReadLibrary commands.

The second file type is the raw data file. It is sometimes convenient to simply load the entire contents of a file into an array so that it may be processed and formatted later (as we do in Chapter 8 – Genetic Databases). These files are read via the ReadRawFile command or the readlines command. When dealing with extremely large files (as is the case with, say, DNA flat files like EMBL), the raw file must be read line by line. Darwin offers several commands for establishing and reading from UNIX pipes.

The third type of file is the genetic database file. A genetic database is a collection of sequence entries annotated with information about these sequences. The genetic database is a cornerstone of the Darwin sequence comparison operations and, as such, it is stored in a special data structure to allow for fast access. We delay discussion about this third type of file until Chapter 8 – Genetic Databases.
7.1 Standard Files

As programs become large, it becomes convenient to break up them up into modular pieces. These modules or collections of related routines, structures and variables can be stored in their own separate file so that accessing them down the road is less troublesome than wading through a huge eclectic file jammed with the all of the routines you have written for every problem you have ever worked on.

Furthermore, it becomes very inconvenient to work directly in Darwin when writing non-trivial segments of code. It is advisable to prototype your Darwin code in a text editor of some sort. After saving the file, you can inject the new code into the environment.

Towards this end, Darwin offers two commands. The command ReadProgram which loads Darwin files from the underlying operating system and the command ReadLibrary which loads files from the Darwin library.

7.1.1 The ReadProgram Command

To load a standard file named filename into Darwin, type

```markdown
> ReadProgram('filename');
```

Note that we can include operating system path information.

```markdown
> ReadProgram('hallett/Darwin/Tree.Construction.Algorithms');
```

Darwin treats the contents of filename exactly as though you had typed the contents in from the keyboard directly. This means that you can not only load routines and structures but instruct Darwin to execute them. If filename contains the following

```markdown
> x := 5;       # the following code is located in the
> y := proc( )  # external file called 'filename'.
>       print('hi');
> end;
```
then Darwin will assign the value 5 to \( x \), invoke procedure \( y \) and afterwards
exit the session.

### 7.1.2 The `ReadLibrary` command

When Darwin was installed on your computer, a special directory was
created which contains the Darwin libraries (see § 1.1 and Appendix A – *Installation Instructions*). The Darwin system can be perceived as consisting of
two parts: the kernel and the libraries. The kernel is responsible for executing
statements, performing simple mathematical operations, and managing
memory. Instructions for more complicated computations, those built from
the basic primitives of the language, are stored in the Darwin library.

The power of a general purpose programming language comes from the
user’s ability to expand these libraries. When tackling a new problem, pro-
grammers typically build a set of routines and structures and store them
in a local file which they can then load in to Darwin. When the code is
perfected and they would like to make it publicly available to other users on
the system, they place their file in the library where all can now access it.

To read a file from the library, Darwin offers the `ReadLibrary` command.

```
> ReadLibrary(DrawPackage);  # load the library DrawPackage
```

This statement loads the entire contents of the file `DrawPackage` in the
Darwin library. Note that we do not specify the path to where the library
resides. The location of the library is set by a flag (the `-b` option) when
the Darwin session is initiated. The system variable `libname` contains the
library path. Appendix A – *Installation Instructions* contains further informa-
tion about these and related Darwin flags.

If we include a procedure, structured type or variable name as a second
argument to `ReadLibrary` command, only that object is read from the file.
> ReadLibrary(DrawPackage, DrawTree);
> # only load function 'DrawTree'
> # located in library file 'DrawPackage'
proc (t:Tree, title:Text, ColorNames:list, Species:list, SpeciesColor:list)
  local t2, rgb, xpts, maxmin, ypts, dt, lab, i; description
  ColorNames: optional list of colors, e.g. [re ..(800).. uitures etc; if
  length(t) < 5 and 2 < nargs then print(Use CT_Color to color the tree first!);
  print(CT_Color); RETURN(0) else t2 := t fi; if length(t2) < 8 and 4 < nargs
  then t2
...

7.1.3 The save Command

The save command allows users to send an object stored in memory to an
external file in such a format that it can be read back into the Darwin envi-
ronment with a ReadProgram command. In essence, users can send to disk
important parts of a work session that might be particularly long and com-
plicated to repeated thus allowing their sessions to be interrupted with the
minimal amount of inconvenience.

> w := 'a text name saved to a file';
> x := [1, 2, 3, 4];
> y := proc()
  return(noeval(copy(y(args))));
> end;
> z := y('abc', 5.5);
> save('filename', w, x, y, z);

Like the ReadProgram command, filename may be preceded by a path
name. The save command can send variables of any built-in type, instances
of built-in structured types, and procedures to an external file.
7.2 Raw Data Files

You have collected a large body of data through experimentation and you would now like to analyze this data with some particular software tool. Unfortunately, you must first place the data into a format that the tool can understand. Your choices boil down to either writing a script in a programming language to make the repetitive transformation or you must spend the next month of your life developing carpal tunnel syndrome accompanied by lower back problems painstakingly creating small mistakes while doing it all by hand. Incompatible formats are certainly not a new problem in computer science yet it can still be a laborious enterprise to normalize data.

This situation commonly occurs when dealing with genetic databases; the different databases (SwissProt, EMBL, GenBank, etc.) have their own unique format and tagging conventions. Before they are usable in Darwin, they must be transformed to correspond to Darwin’s protocols. This section introduces several commands which are designed to read raw data into the Darwin system where it can later be reformatted into the appropriate structure.

The ReadRawFile command loads the entire contents of a file into a Darwin name at once. Once in the array, users no longer have to worry about performing file operations but instead exploit the flexibility the name type has to offer.

The readlines command allows users to load a file line by line. This command is particularly useful when transforming extremely large databases (most notably nucleic databases) which can not fit into your systems memory all at once.

In the opposite direction, one needs to have the capability of sending data produced during a Darwin session to an external file. This is needed so users can produce reports about their findings and to redirect long computations to more permanent files for later examination.
7.2.1 Reading Raw Data Files

The ReadRawFile command

The ReadRawFile command is used as follows:

```plaintext
> contents := ReadRawFile('Sample/rawdata');
%Content: Darwin: Sequence Searching Facility
% ...(7845)

arwin plot
```

Now the contents of 'Sample/rawdata' is placed in an object of type string. The length of contents is equal to the number of characters contained in 'Sample/rawdata'.

Reading from Pipes

The readlines, OpenPipe, and ReadLine commands allow one to read a file in line by line. The are extremely useful when attempting to transform extremely large data files that can not fit into the memory offered by your system. This is typically the case when transforming genetic databases (especially DNA databases such as EMBL) into the Darwin format.

The readlines and OpenPipe both prepare a file to be read via a pipe line by line.

```plaintext
readlines(fn : string)
```

The readlines function accepts a filename (or path and file name) $fn$ as a parameter and establishes a pipe\(^1\) between Darwin and this file.

```plaintext
OpenPipe(cmd : string)
```

The OpenPipe command accepts a string item $cmd$ as a parameter which it passes to the operating system to execute. If the operating system is

---

\(^1\)Readers unfamiliar with pipes should not worry. The basic idea is that a special relationship is established between a Darwin program and the operating system. This allows for files to be transferred in a piecemeal fashion.
successful, **OpenPipe** acts in the same manner as **readlines** and establishes a **pipe** between Darwin and the result of **cmd**. If the operating system reports an error while attempt to execute **cmd**, **OpenPipe** echos this error back to the terminal.

Note, we are only allowed to establish one pipe at a time. However, the pipe does not effect any redirection of input/output caused by **WriteFile/AppendFile** commands.

```plaintext
> readlines('Sample/P00519');  # establish a pipe between Darwin and this file.
>                          # it can now be read line by line
>
> OpenPipe('zcat Sample/P00524.Z');
>                                # via the Unix command 'zcat', decompress this file and establish a pipe.

Once a file has been opened and a pipe established, we can read it line by line via the **ReadLine** command.

**ReadLine**;
```

The **ReadLine** command reads from the open pipe up until the next (new line) character (\n). Typically, the open pipe is the standard input stream (the keyboard). The **readlines** and **OpenPipe** redirect the pipe to a file. We can read the first ten lines of the compressed file **Sample/P00524.Z** as follows:

```plaintext
> ShortRead := proc( filename )
>       OpenPipe('zcat ',filename);
>       for i from 1 to 10 do
>          t := ReadLine();
>          lprint(i, ':', t);
>       od;
>       end:
>
> ShortRead('Sample/P00524.Z');
1 : SRC_RSvr STANDARD; PRT; 526 AA.
```
2 : AC  P00524;
3 : DT  21-JUL-1986 (REL. 01, CREATED)
4 : DT  01-JUL-1989 (REL. 11, LAST SEQUENCE UPDATE)
5 : DT  01-NOV-1995 (REL. 32, LAST ANNOTATION UPDATE)
6 : DE  TYROSINE-PROTEIN KINASE TRANSFORMING PROTEIN SRC (EC 2.7.1.112) (P60-
7 : DE  SRC).
8 : GN  V-SRC.
9 : OS  ROUS SARCOMA VIRUS (STRAIN SCHMIDT-RUPPIN).
10 : OC  VIRIDAE; SS-RNA ENVELOPED VIRUSES; POSITIVE-STRAND; RETROVIRIDAE;

The readlines, OpenPipe and ReadLine problems have displayed irregular
behaviour at times. We have found most (if not all) of these problems dis-
appear when you execute these commands from within a procedure. There
are some very subtle differences between executing a statement directly into
the global Darwin environment and executing a statement from within a
procedure. At the time you define a procedure, Darwin does some rather
extensive analysis of your routine. Beyond issues such as the scoping of local
and global variables, it “looks beyond” ReadLine commands. When execut-
ing a statement outside of a procedure, no such “look ahead” is possible and
it can become confused.

7.2.2 Creating Raw Data Files

The WriteFile and AppendFile commands are used to redirect output to
a file. When the output is directed to a file, it is not echoed to the screen.
The WriteFile command takes one parameter which is either a filename or
the built-in Darwin name terminal. If the parameter is a filename (or a
filename preceded by path information), then all subsequent output is di-
rected into this file. If the parameter is the built-in name terminal, then all
subsequent output is directed towards the standard output. This is typically
the terminal.

> WriteFile('book.tex');       # subsequent output is sent
7.3. **INITIALIZING YOUR DARWIN SESSION**

> lprint(‘A quick way to make a lot’); # to file book.tex
> lprint(‘of work for myself.’);
> WriteFile(terminal); # redirect output to terminal

The WriteFile command creates a new file `book.tex` if no file `book.tex` exists. If `book.tex` already exists, it is removed and a new file with the same name is created.

The AppendFile command is identical to the WriteFile command except that it appends all subsequent output from Darwin to the end of the specified file. If no file exists, a new file is created.

### 7.3 Initializing Your Darwin Session

If you find that you are ritually executing the same set of statements each time you begin your Darwin session, you may want to create a `.darwininit` file in your home directory. Each time you start the Darwin environment, the system loads this file automatically (it essentially performs a `ReadProgram(‘.darwininit’);` statement). Any set of legal statements can be placed in here including `ReadRawFile` and `ReadLibrary` commands. A typical `.darwininit` file might look like this

> CreateDayMatrices(); # calculate the Dayhoff matrices
> PepDB := ReadDb(‘cbrg/DB/SwissProt’); # load Swiss-Prot
> NucDB := ReadDb(‘cbrg/DB/EMBL’); # load the EMBL database
> ReadLibrary(‘MultiAlign’); # library file for performing MSAs
Chapter 8

Genetic Databases

Darwin comes equipped with a number of routines, data structures and types which make it easy to manipulate complete or partial sequence databases. These sequence databases may contain nucleotide, ribonucleotide or peptide sequences. In some sense, these tools form one of the cornerstones to the system and a fluency in their manipulation will greatly ease the difficulty of programming in the language.

This chapter offers a step-by-step guide to

- converting a sequence database to the Darwin format,
- loading and indexing a sequence database, and
- accessing the information contained in a database.

8.1 Sequence Databases

There is a growing number of sequence databases being made available to the general public. Each of these databases has their own raison d'être: some contain only protein sequences, or only nucleotide sequences, or the sequences specific to an organism. Unfortunately, there are as many formats
for databases as there are databases. However, most of the formats are well-defined and most sites offer short manuals detailing the tagging conventions and overall layout of their database.

Historically, the Computational Biochemistry Research Group has focused mainly on two databases: the annotated protein database Swiss-Prot [3, 4] and the nucleotide sequence database EMBL [10].

Figures 8.1 and 8.2 contain an example of a Swiss-Prot entry and a EMBL entry respectively. Each line (excluding the sequence entry SQ) begins with a two letter code which indicates the type of the line. Both in Swiss-Prot and EMBL, ID indicates the identification of the sequence, AC is the accession number, DE is the description, SQ is the actual sequence, and so forth. Unfortunately, this is not always the case. There are several types of lines which have no analog in the other database. For this reason, we must reformat all of the sequence information into a normalized form Darwin understands. We show how to perform this normalization using the Swiss-Prot database but note that the normalization of other databases follows, more or less, the same lines.

8.2 Darwin Sequence Databases

To represent sequence databases internally, Darwin has chosen to use the ISO-SGML (International Standards Organization - Standard Generalized Markup Language) tagging convention [21]. SGML has begun to be used extensively, namely with HTML and the World Wide Web (WWW). SGML offers many advantages:

- The format for an SGML tag consists an opening tag `<tag>` and a closing `\` where tag is a name consisting of any sequence of letters and numbers. The opening and closing tag surrounds a field of information. This provides an essentially infinite name space for tags.
Figure 8.1: A sample Swiss-Prot entry.

[A sample Swiss-Prot entry]
**CHAPTER 8. GENETIC DATABASES**

| XX | M14333; |
| XX | X1 | g18117 |
| XX | DT | 16-JUL-1988 (Rel. 16, Created) |
| DT | 06-JUL-1989 (Rel. 20, Last updated, Version 1) |
| XX | DE | Homo c-sym protooncogene, complete cds. |
| XX | KW | oncogene. |
| XX | OS | Homo sapiens (human) |
| XX | GC | Eukaryote; mitochondrial eukaryotes; Metazoan; Chordata; Vertebrata; Eutheria; Primates; Cetartiodactyla; Homo. |
| XX | HE | [1] |
| RP | 1-2647 |
| RX | HEBTLINE; 06287276. |
| NA | Sembh K., Bishirzawa M., Miyazima M., Yoshida M.C., Sukeyaw J., |
| NA | Yamamoto K., Toyoshima K. |
| RT | “yes-related protooncogene, syn, belongs to the protein-tyrosine kinase family”; |
| XX | DR | CPG1SLB; HSCSYMA; Release 3.0. |
| DR | SWISS-PROT; P00241; P51_HUMAN. |
| XX | CC | belongs to the protein-tyrosine kinase family of retroviral oncogenes. |

**Sequence** 2647 BP; 683 A; 695 C; 716 G; 553 T; 0 other;

**Key**

<table>
<thead>
<tr>
<th>Location/Qualifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-2647</td>
</tr>
<tr>
<td>/organism=&quot;Homo sapiens&quot;</td>
</tr>
<tr>
<td>/c-sym=&quot;c-sym&quot;</td>
</tr>
<tr>
<td>/db_xref=&quot;PDB:18117&quot;</td>
</tr>
<tr>
<td>/db_xref=&quot;SWISS-PROT:P00241&quot;</td>
</tr>
<tr>
<td>/translation=&quot;MCGVCQKEDKEAKLTDERDGSLEQSGTVDFTQP5PHGVT&quot;</td>
</tr>
<tr>
<td>/locus_tag=&quot;HSCSYMA&quot;</td>
</tr>
<tr>
<td>/protein=&quot;c-sym&quot;</td>
</tr>
</tbody>
</table>

**Figure 8.2:** A sample EMBL entry.

[A sample EMBL entry. We have abbreviated the entry somewhat for space reasons.]
8.2. **DARWIN SEQUENCE DATABASES**

- SGML tags can be nested to conform to any structure; this allows for a very rich and flexible structuring of data.

- Compared with other formats which use blank lines and spacing patterns, SGML tags are quite economical in terms of storage.

A Darwin sequence database has three simple conventions:

1. Each entry must begin with the tag `<DB>` and end with the tag `</DB>`.

2. The sequence (whether it consists of nucleotides, ribonucleotides or peptides) must begin with the tag `<SEQ>` and end with the tag `</SEQ>`.

3. No character except those representing a nucleotide (or ribonucleotide, peptide as the case may be) is allowed between the opening and closing `<SEQ>`, `</SEQ>` tags.

Beyond these, users may include as many fields as they would like with each entry and the choice for tag names for these fields are entirely the users choice.

<table>
<thead>
<tr>
<th>Type</th>
<th>Legal Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNA</td>
<td>A,C,G,T</td>
</tr>
<tr>
<td>RNA</td>
<td>A,C,G,U</td>
</tr>
<tr>
<td>Mixed</td>
<td>A,C,G,T,U</td>
</tr>
</tbody>
</table>

Table 8.1: A table showing the different types of Darwin sequence databases and the legal bases for each.

Table 8.1 lists the types of sequence databases allowed in Darwin with the legal characters associated with each type.
The following variable ProtoOncogene contains what a Darwin entry might look like for the Swiss-Prot entry from Figure 8.1. The backslash symbol (\) allows us to split entries across lines.

> ProtoOncogene := <E>
> <ID>FYH_HUMAN</ID>
> <AC>P06241</AC>
> <DE>PROTO-ONCOGENE TYROSINE-PROTEIN KINASE FYN (EC 2.7.1.112) (P59-FYN)
> (SYN) (SLK)</DE>
> <OS>HOMO SAPIENS (HUMAN)</OS>
> <OC>EUROPTOTA; META20A; CHORDATA; VERTEBRATA; TETRAPODA; MAMMALIA</OC>
> <EUTHERIA; PRIMATES</E>
> <DR>PDB; 1SHF</DR>
> <KW>PROTO-ONCOGENE; TRANSFERASE; TYROSINE-PROTEIN KINASE; PHOSPHORYLATION;
> ATP-BINDING; MYRISTYLACTION; SH3 DOMAIN; SH2 DOMAIN; 3D-STRUCTURE</KW>
> <FT>ACT_SITE 389 389</FT>
> <SEQ>GCVQCKDKEATKLEEDAGSLNQSSG7RYYGTDTPQHYPSFGVTSIPNYNNFHAAGQGLTVFGVMS
> SSHTGLTRGRTGGTVLAVYAYEARTEDLSFHKGKEQILNVEGDEWAEARLTTGETGYIPSNY
> VAPVDSIQAEWWYFGKLGRKDAEQLSFGNPRTFLIRSETTKGAYLSLISRDWDMDKGDHVRHYK1
> RKLDPQGYYITITRAQETFLLQ VLQHYSSERAAGLCRLVPCHKQPNRLTLSV KTDVWEIPRESLQL
> IKRLGNQQFGEVWGNTWNGNTRA ICTLKPCTMSPESFLFEEAQIMKKHRKDLKVQLYAVVSEPPYIV
> TEYMNKSSLDDFLDKDEGRALKLPLNMDAAAQVAAAGMAYIERNMYIHRDLRSANILVNGNIKCIADF
> GLARLEDNEYTARQGAKFPIKWTAPAALYGRFTIKSDVWSFGILLTELVTGKVREIPQMNBBVLE
> QVFRGMXPCPQDCPISLHELMIHCKKDFEEPRTFEYLQSFLEDYFATPEYQYPGENL</SEQ>
> </E>

Notice that not all type lines from the Swiss-Prot entry are present in the Darwin version. Depending on the user's objectives, some information kept in the original database may be discarded in an effort to minimize the amount of memory required for the database and save storage space. In particular, we have chosen to only keep the ID (identification), AC (accession number), DE (description), OS (organism species), KW (keyword), a part of the FT (feature table), and SE (sequence entry) fields.
8.3. **BUILDING A DARWIN SEQUENCE DATABASE**

Now, via the `SearchTag` command introduced in §2.2.3, we can extract information in any of the fields.

> `SearchTag('DE', ProtoOncogene);
> SearchTag('FT', ProtoOncogene);

The general procedure for building a Darwin sequence database is as follows:

1. For each entry of the sequence database, convert the entry to the Darwin SGML format.

2. Concatenate each converted entry into one external file.

3. Load the file into Darwin via the `ReadDb` command.

The next subsection presents a simple Darwin program to perform steps 1 and 2 above for the Swiss-Prot database.

### 8.3 Building a Darwin Sequence Database

The previous section showed what a typical Darwin entry might look like. In this section, we build a rudimentary parsing engine to convert the entire contents of Swiss-Prot into a Darwin friendly format. The Darwin library offers the function `SpToDarwin` which performs a slightly extended version
of what we perform here.

**Calling Sequence:**

```
SpToDarwin(flatfile, darwinfile, descr : name, compressed : boolean)
SpToDarwin(flatfile, darwinfile, descr : name)
```

**Parameters:**
- `flatfile` : name
- `darwinfile` : name
- `descr` : name
- `compressed` : boolean

**Returns:** Converts a Swiss-Prot flat file (`flatfile`) into a Darwin loadable file (`darwinfile`). The parameter `descr` should contain a name item with the `DESCR`, `DBNAME` and `DBRELEASE` tags. If `compressed` is specified and true, `flatfile` is read using the Unix command `zcat`.

Our procedure, `ConvertDB` accepts either three or four parameters. The first argument `flatfile` is the flat file or raw data file containing all the Swiss-Prot entries we would like to convert. The second argument `darwinfile` is the destination file where we will store the converted form. The third argument allows us to specify the name of our database. The value of `descr` is wrapped in `<DESCR>, </DESCR>` tags. If the boolean variable `compressed` is specified and true, then `flatfile` is first decompressed via the Unix `zcat` command.

After establishing a pipe between Darwin and the flat file `flatfile` (see §7.2.1) and writing the database descriptor to file `darwinfile`, we can begin parsing the Swiss-Prot flat file. The parsing follows a simple schematic shown in Figure 8.3.

```lisp
ConvertDB := proc( flatfile : name, darwinfile : name, 
                   descr : name, compressed : boolean )
local state;
  secstruct := CreateString(1..5000);    # temporary holder for FT tag
```
Figure 8.3: The parser for a Swiss-Prot file. We begin in state Outside Entry. When any line tag is found which is a member of the set TagsToKeep, the state is changed to Inside Entry. This state is responsible for creating the SGML code for all labels except SQ and FT as both of these states require special attention. State SQ is responsible for parsing a sequence tag and the sequence stretches across multiple lines of the flat file without further SQ tags. With respect to the FT (feature table), we are only interested in keeping FT information corresponding to secondary structure assignments. We switch to state FT Tag Initial upon encountering the first such FT tag. State FT Tag Latter is reached only if we encounter an FT tag which has information we keep. The symbols // mark the end of an entry. In this case, we switch to state Outside Entry. State Finish is reached only after every entry from the flat file is processed.
state := 0;
# state -> 0 : Outside Entry; 1 : Inside Entry; 2 : FT Tag Initial
# 3 : FT Tag Latter 4 : SQ Tag 5 : Finish
TagsToKeep := {'ID', 'AC', 'DE', 'OS', 'OC', 'FT', 'SQ'};

if compressed then
    # establish a pipe between Darwin and
    OpenPipe('zcat ./flatfile');
else
    readlines(flatfile);
fi;

WriteFile(darwinfile);
printf('<DBDESCR>%s<CONVDATE>%s</CONVDATE></DBDESCR>
', descript, date());

if (state=0) then
    if (t=EOF) then
        state := 5; next;
    # go to state Finish and skip to top of loop
    endif
    if (member(tag, TagsToKeep)) then
        state := 1; printf('
\nE');
    # go to Inside Entry open an entry and skip to top
    else
        # do nothing
        fi;
else
    if (tag='//') then
        state := 0; printf('
\nE');
    # close entry
    if (tag='FT') then
        state := 3; next;
    endif
    if (tag='SQ') then
8.3. BUILDING A DARWIN SEQUENCE DATABASE

    state := 4; next;
    elif (member(tag, TagsToKeep)) then
        ParseTag(t); next;
    fi;
    elif (state=2) then
        if (tag='FT') then
            temp := ParseFTLine(t);
            if (temp <> NULL) then  # information from the line was kept
                state := 3; printf(\"\n<FT>\%s', temp);  # good FT tag
            fi;
        elif (tag='//') then
            state := 0; printf(\"\n</E>\n\');
            # go back Outside Entry and close entry
        else
            state := 1; next;  # go back to Inside Entry
        fi;
    elif (state=3) then
        if (agt='FT') then
            for i to length(sec_structure) do sec_structure(i):=' '; od;
            length_structure := 0;
            ParseFTLine(t, sec_structure, length_structure);
        elif (tag='//') then
            state := 0; printf(\"\n%<FT></E>\n\', sec_structure[1..length_structure]);
        else
            state :=1; next;
        fi;
    elif (state=4) then
        if (tag='SQ') then
            printf(\"\n<SEQ>\%s</SEQ>', ParseSequence());
            next;
        else
            state := 1; next;
        fi;
    elif (state=5) then
break;
fi;

t := ReadLine();  # get the next line from the pipe.
tag := t[1..2];      # get the next tag.
od;

WriteFile(terminal);
end:

All that remains from completing our parser is to write the procedures
ParseTag, ParseSequence, and ParseFTLine.

ParseTag := proc( t : name )
tag := t[1..2];
printf(’\n<%s>’, tag);
while (true) do
  p:=3;
  while (t[p] = ’ ’) do p := p+1 od;
  printf(’%s ‘, t[p].length(t));
t := ReadLine();
if (t[1..2] <> tag) then
  printf(’</%s>\n’);
  break;
fi;
end;

ParseSequence := proc( )
seq := ’ ’;    # seq holds the partial sequence
while (true) do
  t := ReadLine();  # get the next line from the pipe
tag := t[1..2];
  if (tag = ’ ’) then
for p from 3 to length(t) do
    if (t[p] > ' ') then
        seq := seq . If(AToInt(t[p]) = 0, 'X', t[p]);
    fi;
    od;
else
    return(seq);
    fi;
end;

ParseFTLine := proc( t : name, sec_struct : name, length_struct : posint )
global sec_struct, length_struct;
    temp := sscanf(2+t, '%s %d %d');
    if (length(temp) = 3) and (((temp[1] = 'TURN') or (temp[1] = 'HELI X')
                         or (temp[1] = 'STRAND')) then
        if (temp[3] > length_struct) then length_struct := temp[3]; fi;
            if (temp[i] = 'TURN') then sec_struct[i] := 't';
            elif (temp[i] = 'HELI X') then sec_struct[i] := 'h';
            elif (temp[i] = 'STRAND') then sec_struct[i] := 's';
            fi;
        od;
        fi;
    end;

8.3.1 Map Files and Patricia Trees

If a sequence database (in the correct SGML/Darwin format) has never been
loaded before, Darwin loads the entire contents of the file into memory where
it is analyzed and re-organized into a more convenient format for the system.
Using the name you choose for darwinfile in the previous section or the
file Sample/SH2, load this file via the ReadDb command.
> ReadDb('Sample/SH2');

Reading 76825 characters from file Sample/SH2
Pre-processing input (peptides)
78 sequences within 78 entries considered
Building new Pat index in file Sample/SH2.tree with 43582 entries
Pat index with 43582 entries
  sorted, from "A</SEQ></E>

Peptide file(Sample/SH2(76825), 78 entries, 43582 aminoacids)

If the size of the file is at least the value assigned to the system variable mapsizes (see §11.2 – Internal Variables), a special file is created with the extension .map added to the name of your flat file. This file contains information regarding the contents of the database include sequence type, sequence size, location of the sequences, and the number of entries. If the size of your file not does reach the lower bound defined by mapsizes, Darwin recalculates this information each time the database is loaded. If you would like to force Darwin to build such a map, one need only lower mapsizes to a value less than the number of characters contained in the database.

> ReadDb('Sample/SH2');
Reading 76825 characters from file Sample/SH2
Pre-processing input (peptides)
78 sequences within 78 entries considered
Building new Pat index in file Sample/SH2.tree with 43582 entries
Pat index with 43582 entries
  sorted, from "A</SEQ></E>

Peptide file(Sample/SH2(76825), 78 entries, 43582 aminoacids)

> Set(mapsize=76825);
131072

> ReadDb('Sample/SH2');

Reading 76825 characters from file Sample/SH2
Pre-processing input (peptides)
78 sequences within 78 entries considered
Creating file Sample/SH2.map for mapping
Peptide file(Sample/SH2(76825), 78 entries, 43582 aminoacids)

Darwin also places the contents of the SEQ field for each entry into a special data structure called a *patricia tree*. This is reported by the `ReadDb` command in the lines

Building new Pat index in file SH2.tree with 43582 entries
Pat index with 43582 entries

This structure facilitates fast searching and matching operations on the database. Darwin stores the *patricia tree* structure in a file named with a `.tree` extension added to the flat file name. Now, everytime this database is loaded, the `.tree` file is also loaded.

With large databases, the Patricia tree may take a long time to build and the resulting `.tree` file may be much larger than the database itself. Although the Patricia tree speeds up searching and matching operations performed on the database, it is optional. If you decide that you do not want Darwin to build such a structure, create an empty file with name `database.tree` in the same directory as your sequence database `database`. The easiest way to do this in Unix is via the `touch` command as follows:

```
touch database.tree
```

Now, when you re-load your database, Darwin acknowledges the existence of the empty `.tree` file and does not attempt to rebuild the tree.

The *patricia tree* data structure itself is a type of binary tree which has some nice properties that allow fast and efficient searches with very long or unbounded length strings. Most standard computer science algorithm and data structure textbooks give an introduction to the data structure. Amongst others see [14] for further information.
The **ReadDb** command returns an object of the built-in structured type database. By default, this structure is assigned to the system variable **DB** but we may assign it to any Darwin name.

```r
> ReadDb('Sample/SH2');
> type(DB, database);
> SH2 := ReadDb('Sample/SH2');
> type(SH2, database);
```

### 8.4 Accessing a Darwin Sequence Database

This section contains a complete description of the Darwin built-in type database and details how to access the individual components of a genetic database. We begin by loading a genetic database and assigning it to the (default) system variable **DB**.

```r
> DB := ReadDb('Sample/SH2');
```

The **DB** name has a special status in Darwin. When performing matches on entries of a database, the matching algorithms require that the database be assigned to **DB** in most cases.

Table 8.2 lists the various selectors available for the structured type database. These selectors allow us access to the information kept in the .map file.

```r
> DB[FileName];
Sample/SH2
> DB[type];
Peptide
> DB[TotAA];
43825
> DB[TotChars];
76825
> DB[TotEntries];
78
> DB[Entry, 10];
```
8.4. ACCESSING A DARWIN SEQUENCE DATABASE

<table>
<thead>
<tr>
<th>Selector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry, i</td>
<td>Returns the offset of the ( i )th entry</td>
</tr>
<tr>
<td>FileName</td>
<td>Returns the external filename. (^a)</td>
</tr>
<tr>
<td>Pat, i</td>
<td>Returns the offset into the patricia array of the ( i )th sequence character.</td>
</tr>
<tr>
<td>string</td>
<td>Returns the database as a string.</td>
</tr>
<tr>
<td>TotAA</td>
<td>Returns the total number of bases.</td>
</tr>
<tr>
<td>TotChars</td>
<td>Returns the total number of characters from all fields of the database.</td>
</tr>
<tr>
<td>TotEntries</td>
<td>Returns the total number of entries.</td>
</tr>
<tr>
<td>type</td>
<td>Returns the type of the database. (^b)</td>
</tr>
</tbody>
</table>

\(^a\)This does not include any path information.
\(^b\)This is either type Peptide, DNA, RNA, or Mixed.

<table>
<thead>
<tr>
<th>12701</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; DB[Pat, 1];</td>
</tr>
<tr>
<td>75820</td>
</tr>
<tr>
<td>&gt; DB[string];</td>
</tr>
<tr>
<td>&lt;DBNAME&gt;SH2 Database&lt;/DBNAME&gt;&lt;DBRELEAS$1.0&lt;/DBREL ..(76825)..</td>
</tr>
<tr>
<td>G&lt;/SEQ&gt;&lt;/E&gt;</td>
</tr>
</tbody>
</table>

Darwin allows multiple sequence databases to be loaded simultaneously. These can be assigned to any valid Darwin name. For instance,

\[
\text{> SP := ReadDb('"cbrg/DB/SwissProt'); \quad \# load SwissProt &}
\]
\[
\text{> \quad \# assign it to the variable SP}
\]

\[
\text{SP := Peptide file(/homes/cbrg/DB/SwissProt(34675450), no index, 52205 entries, 18531385 aminoacids)}
\]

\[
\text{> EMBL := ReadDb('"cbrg/DB/EMBL39/EMBL39.fun'); \quad \# load EMBL &}
\]
\[
\text{> \quad \# assign it to variable EMBL}
\]

\[
\text{> DB := SP;}
\]
DB := Peptide file(/homes/cbfrg/DB/SwissProt(34675450), no index, 52205 entries, 18531385 aminoacids)

With the assignment of SP to DB, the Swiss-Prot database becomes the current database. The type of both variables is database.

> type(SP, database);
type(SP, database);
true
> type(EMBL, database);
type(EMBL, database);
true

The system variable DB holds special status in Darwin. The database assigned to DB is the current database, that is, when any searches and matches are performed, they are performed here.

8.4.1 The Entry Structure

The entire database Sample/SH2 is stored in Darwin as a string of length DB[0], DB[1] stores the name, Figure 8.4 gives a graphical view of how information is organized internally. DB[string] points to the beginning of this name. Recall that each entry from a sequence database in Darwin is wrapped in the SGML tags <E>, </E>. To extract the entire contents of an entry, we use the Entry structured type.

> ReadDb('Sample/SH2');
> first := Entry(1);
first := Entry(1)
> second := Entry(2);
second := Entry(2)
> last_three := Entry(76, 77, 78);
last_three := Entry(76, 77, 78)
> print(first);

<E><ID>ABL1_CAEL</ID><AC>P03949;</AC><DE>TYROSINE PROTEIN KINASE ABL-1 (EC 2.7.1.112) (FRAGMENT).</DE><OS>CAENORHABDITIS ELEGANS</OS><OCAKYOTYA; METAZOA; ACHELOMATES; NEMATODA; SECERNENTEA; Rhabditida.</OC><KW>TRANSFERASE; TYROS
8.4. ACCESSING A DARWIN SEQUENCE DATABASE

INE-PROTEIN KINASE; SH2 DOMAIN; SH3 DOMAIN.</KW><FT>ACT_SITE 283 283</FT><SEQ>NNEWCEARLYSTRKNDASNQRRLGEIGWPSMFIAFYN
SLDKYTWHGKISRDSDEAILGSGITSGFLVRESRESISQYTISVRHDFVRVHPYRINV
DNTKWMFITQEVKFRTLGVEHHHSVADGLICLMLMPASKKDDKGRGLFLSPLNPAP
WELRSEIHMNKLGQGQYDVYGEGYKRHDCTIAVKALEDAMPIHLFELAEEAMIKD
LHHKNLVRLLGVCTHEAPFYIITEFMGCNGLLELYLRTDKSLLPPIIILVQMAQUIAG
MSYLEARHFIHRDLAARNLCSVHENIKIADFGLARFMKEDTAYTAHAKKFPIKWATAP
EGLAFNTFSKSDVWAFGVVLEIAAYMAPYPGVGLSNVYGLLENFRMDGPQCP
SVYRLMLCWNWSPSDRPFRFDHFIUNELENLISSNLNDVQKIKLDKedesKRR
NHVRESDSKRSHSHDDDRDSLHESRNSNEIPNFSFRDTSVSVHFFWTSKV
TSFDAQQGPPPPPQVNTKPKLKSVLNSNARHASEEFERNEQDDVVLAEKNVR</SEQ></E>
> print(</E>
<E>3ID>ABL2_HUMAN</ID><AC>P42684</AC><DE>TYROSINE-PROTEIN
KINASE ABL2 (EC 2.7.1.112) (TYROSINE KINASE ARG)</DE><OS>
HOMO SAPTIS (HUMAN)</OS><CE>EUARKOTA; METAZO; CHORDAT
A; VERTEBRATA; TETRAPODA; MAMMALIA; EUTERIA; PRIMATES</CE>
</KW><KW>ACT_SITE 409 409</KW><SEQ>MGQG
VQVGEAFLQQPQPREGRGSAARPSGRRDPAGRTTGFHFITQHDFASCVEDG
FEGDKTGSSPEALHPYGDVPEQALNEAIRwSSKENEILGATESDPNLTLVALYDF
AEGNTLSTKGEKLRVLQYNQGWSKRESNQGQWVPSNYITPWNKLEKHSSYHPGV
SRRASYEYLSSSLINGFLVRESSESPQGSLISLGYEGRVHYRINTTADGKVYVTAES
RFSFLAELVHHHSVADGLVTLTHVAPKCNKPTVYGPSIHDKWEMERTDITMKHL
GGQYGEYVGYVWKKYLSTLTAVKTLKEDTMEVEEFLKEAAVMKEIKHPNLVQLLGV
CTLEPPFYIETYMPYGNLHLLRECDREETAVVLLATMQISSAMEYELKKNFIHRDL
AARNCLVGENHVVKADFGLSLRMTGDYTAAHAGKFPIKWTAPESLAYNTFSIKSDV
WAFGVVLEIAAYMAPYPGPDSLQVYDLEKGYRMEQPEGCPPKVEELMRAECKWSYP
ADRSFSAEHQQAFETMFDSSISEEVAELGRAASSSSVVPYPLRPILPSKTRTLKK
QVFNKCHEGAQDAQDEASMPLAPFIRGAAQASSGSPALPRKQNKSPLLIZEDAKET
CFTDRAKKGFGSFMKRMNAPPKRSFSSRFMENQPHKKYETLGNFSSVSLQHADG
FSFTPAQQEANLVPPKCYGGSFQARNLCMDDGGGGGSGTAGGGWSGTGFFTPRLIK
KTLGLRAKPTASDDTSPFPBSNSTSSSMSSGLLLPEQDRMAMTLPRNCQRSKLQLERTV

STSSQPEEENVRANDMLPKKSEESAAPSRERPKAKLLPRGATAPLRTLPSGDLAITEKDPPG\nGVAGVAAAPKGEKNGGARLGMAGVPEQDEQPSPAKAPVLPPTTHNKV\nVLSPTLKHPTADVQLGTDSQGNKFLLSEHVTVSSGKDRPRVKPCAPPPPV\nRLLQHPSICSDFPEPTALTAGGQSTSEGGKGAALGVQISGKAGRPVMPPQVPL\nPTSSISPAKMANGTAGKVALRKTQAAEKISADKISKEAALCDDLSSLATEFVPN\nSQLVDTGHQ2LLDYCSGYVDQCIPTRKFAFREAVSKLESLQELQVSSAAAAGVPGTNPV\nVLNNLSCVQESIDVVR</SEQ></E>

We can isolate the contents of a specific SGML tag by including the tag in single quotes and square brackets.

> first['ID'];  # get the identification tag of the 1st entry
ABL1_CAEL
> first['SEQ'];  # get the sequence for the 1st entry
NNE\NCEARLYSTRKRDASNQRRLGEIGWVPSNF\LYAPYNLDKTYTW\N657\N.. DVVP\NKVR

> second['FT'];
ACT_SITE 409 409

> last_three['DE'];  # get the description tag
8.4. ACCESSING A DARWIN SEQUENCE DATABASE

# for the last three entries

[PROTO-ONCOGENE TYROSINE-PROTEIN KINASE YRK (EC 2.7.1.112) (P60-YRK) (YES RELATED KINASE).]

TYROSINE-PROTEIN KINASE ZAP-70 (EC 2.7.1.112) (70 KD ZETA-ASSOCIATED PROTEIN).

TYROSINE-PROTEIN KINASE ZAP-70 (EC 2.7.1.112) (70 KD ZETA-ASSOCIATED PROTEIN).

Notice that when an Entry structure has only a single posint parameter, as is the case with first and second above, and we select for a specific tag, then it returns the contents contained in this field as a name object. When more than one entry is specified, as is the case with last_three, it returns a list of string objects. The i-th element of this list corresponds to the i-th parameter of Entry.¹

8.4.2 The Offset Structure

Because a Darwin sequence database is stored as a string object assigned to DB[string], we can access elements of it as we would any array or string.

> ReadDb('Sample/SH2');    # load the small Swiss-Prot database
> DB[string][1];            # the first character of the database
<
> DB[string][1..55];        # the first 55 characters of the database
<DBNAME>SH2 Database</DBNAME><DBRELEASE>1.0</DBRELEASE>
> DB[string][368..923];     # the sequence of the first entry.
NNEWCEARLYSTRKDASINQRLGEIGWPSNFAPYNSLDKTYTWYHGI ..(556).. DDVVPFAEKNV

If we cross-reference the above statements with Figure 8.4, we see that DB[string][1] is the first character of the database which is an opening angle bracket. The first through fifty-first characters offset from DB[string]

¹Recall the SearchTag function introduced in Chapter 2, Section 2.2.3. Function SearchTag('/AC', 'ID:<ID>ABL1:CAEEL</ID><AC>P03949;</AC>') selects with the Entry(x) structure automatically searches the x-th entry of the DB database for the appropriate SGML tag.
Figure 8.4: A diagram showing the database structure DB. Here DB is a protein database consisting of 78 sequences from Swiss-Prot. The newline characters have been changed to space symbols.
8.4. ACCESSING A DARWIN SEQUENCE DATABASE

consist of the database header and characters 368 through 923 represent the sequence of the first entry. The statement

```plaintext
> print(Offset(2536));
ID>ABL1_HUMAN</ID><AC>P00519;</AC><DE>PROTO-ONCOGE ..(74289) .. G</SEQ></E>
```

is equivalent to

```plaintext
> print(DB[string][2536+1 .. DB[TotChars]]);
ID>ABL1_HUMAN</ID><AC>P00519;</AC><DE>PROTO-ONCOGE ..(74289) .. G</SEQ></E>
```

The Offset structure allows us to find in which entry a particular offset from DB[string] lies. In Figure 8.4, offset 2536 lies in entry number 3.

```plaintext
> entry_num := Entry(Offset(2536));
entry_num := Entry(3)
> print(entry_num);
<E><ID>ABL1_HUMAN</ID><AC>P00519;</AC><DE>PROTO-ONCOGE TYROSINE-PROTEIN KINASE ABL (EC 2.7.1.112) (P150) (C-ABL) .</DE><OS>HOMO SAPIENS (HUMAN).</OS><OC>EUKARYOTA; METAZO ...
```

Darwin converts Offset(2536) into the entry number located at position DB[string][2536].

```plaintext
> entry_numbers := Entry(Offset(1, 55, 60, 881, 937, 2475, 2533, 76800));
entry_numbers := Entries(0, 0, 1, 1, 2, 3, 78)
```

Alternatively, we may select on an Entry structure using the option 'Offset' or '0'. This returns a Offset structure containing the offset from DB[string].

```plaintext
> Entry(1)['Offset'];
Offset(56)
> Entry(1)['0'];
Offset(56)
> Entry(2, 3, 77, 78)['Offset'];
Offset(936, 2532, 74814, 75833)
```
8.4.3 The Sequence Structure

We have already seen how sequences can be extracted using selection and the Entry structured type.

> Entry(4)['SEQ'];
MGAQQGKDGAHSGGGSGAPVSCIGLSSSPVASVPHCISSSSGVSSAP ..(1520)..
SLRQISNALNR

This statement returns a copy of what is found in Entry(4) of the Sample/SH2 database as a string.

However, it is sometimes convenient to instead reference a sequence in the database rather than making a copying of it. The Match structures which we explore in Chapter 21 - The Pairwise Comparison of Sequences require such references. To reference a sequence in Darwin, we provide the offset of the sequence from DB[string]. The Sequence structure allows us do this easily. Given a Entry structure, it returns the offset to the end of the opening <SEQ> tag for that entry in the form of an unevaluated Sequence function call.

> Sequence(Entry(1));
Sequence(367)
> Sequence(Entry(2));
Sequence(1338)
> Sequence(Entry(76, 77, 78));
Sequence(74267, 75202, 76197)

We can combine the Entry, Offset and Sequence structured types to return the offset of the sequence contained in an entry given only an offset from DB[string].

> offset_from_DF := 45000;
> entry_number := Entry(Offset(offset_from_DF));
entry_number := Entry(45)
> seq := Sequence(entry_number);
seq := Sequence(44545)
> seq;
8.4. **ACCESSING A DARWIN SEQUENCE DATABASE**

Sequence(44545)
> print(seq);
MKERVKEMK VFGCRLNFWHIGHEDQFGNQRQRRLQPQIRAASPNSTTNSQ
FSLQHNSGGSSLGGGGVGGGLGGSGGLGGGGSCTPTSQPQSTTFKQSPTL
LNGNMLDDANMPGGITPGTPNSKAKDNSHFVKLVVALYLGKAIEGGDSLVEKNA
EYEVIDDSQEHWKKVDALGNVGYIPSNVQAEAILGLERHEYWGVYSRQRAELSLL
KQGDKECFVVRXSTKGTYTSLHTKVFQSHKHYHKIQMRCYEYLSKEHCETIPDILNYHRHNSGGACRLKSSPCDRIYPPTAGLHDKWEIHPIQLMELGSGQFG
VVRRGKWRSIDTAVKMKETMSEDDFIEEAKVMKLQHPNLVQLGVCTKHPITY
IVTEYMKHGSLLYNLRRHEKTLGNMGLLDMCQIVSKGMYLERHNYIHRDLAARN
CLVGSENVVKVADFGLARYVLDDQYTSSTGKTPIKwAPEVLYNYTRFSSKSDVWAY
GVLMEIFTGKMPVRGKNTVEVVERQRGIILEKPKSCAKIEYDVKLCWSHGPEE
RPABRVLMDQALVAQTLTD

Darwin offers a simpler way to find the offset of a sequence for an entry. Selecting on an Entry structure with option 'SequenceOffset' or, simply, 'S0' returns a Sequence structure containing the offset.

> Entry(1) ['S0'];
Sequence(367);
8.4.4 The String Function

The String function allows one to extract sequences of an entry to be extracted easily.

Calling Sequence:

\[
\text{String}(x_1, x_2, \ldots x_k) \\
\text{String(Sequence}(y_1, y_2, \ldots, y_{k'}) \\
\text{String}(x_1, x_2, \ldots x_k, \text{Sequence}(y_1, y_2, \ldots, y_{k'}))
\]

Parameters:
\[x_i : \{\text{posint}, 0\}\]
\[y_i : \{\text{posint}, 0\}\]

Returns: This function returns a list of sequences. For each \(x_i\), \(\text{DB}[\text{string}][x_i..\text{DB}[\text{TotChars}]]\) is returned. For each element \(y_i\) of a Sequence structure, the contents of the \(<\text{SEQ}>\), \(<\!/\text{SEQ}>\) tags for the entry found at Sequence\((y_i)\) are returned as a string.

\[> \text{String}(1, 100, 1000, 20000, 70000); \quad \# \text{first form of String} \]

\[\text{[DBNAME]}\text{SH2 Database}\langle\text{DBNAME}\rangle<\text{DDBRELEASE}>1.0</\text{DBCRELE} \ldots(76824)\]
\[.. \text{G}/\text{SEQ}?></\text{E}>, \text{ROSINE-PROTEIN KINASE ABL-1 (EC 2.7.1.112) (F} \]
\[\text{RAGME} \ldots(76725)\ldots \text{G}/\text{SEQ}?></\text{E}>, \text{E ABL2 (EC 2.7.1.112) (TYROSIN} \]
\[\text{E KINASE ARG)}/\text{DE}>< \ldots(75825)\ldots \text{G}/\text{SEQ}?></\text{E}>, \text{VSTSQQLQPALHVAE} \]
\[\text{GMEYLESKKLVHRLAARNILVSEDLVAKVSDKFGL} \ldots(56825)\ldots \text{G}/\text{SEQ}?></\text{E}>, \text{MAN}/\text{ID}<\text{AC}\rangle\text{P07947;}/\text{AC}<\text{DE}>	ext{PROTO-ONCOGENE TYROSIN} \ldots(6825)\ldots \]
\[\text{G}/\text{SEQ}?></\text{E}>>] \]

\[> \text{seq\_offsets} := \text{Sequence(Entry}(1, 2, 78)); \quad \# \text{get the sequence offsets} \]
\[\text{seq\_offsets} := \text{Sequences}(367, 1338, 76197) \]

\[> \text{the\_seqs} := \text{String(seq\_offsets)}; \quad \# \text{get the sequences} \]
\[\text{the\_seqs} := \text{[NNNEWCEARLYSTRKDASNOQLRGEIGWPSNFIALPINSLDKSYTWYHG} \]
8.4. ACCESSING A DARWIN SEQUENCE DATABASE

KI ..(557) .. DVVPLAEKNVR, MGQQVGRVGEAPGLQQPFRGIRGSSAARPSSGRRR
DPAGRTETGFI N ..(1182) .. CVQESDVVVR, MPDPAAHLPFFYGSISRAEAE
EHLKLAGMDGLFLLRQCLRSLGGYVSL ..(618) .. QCEQVAEACG]

> mixed := String(60000, 70000, Sequence(2533));

mixed := [YVAEYKSLDAE EWFGQVVRVDAEKQLMFPNNLGSFLIRDSDTTPGDFS.
 ..(16825) .. G</SEQ></E>, MAN</ID><AC>P07947;</AC><DE>PROTO-ONE
OGENETYROSIN ..(6825) .. G</SEQ></E>2, MLEICLKLGCKS KGLSSSSSSC
YLEEALQ RPVDSPGQLSEAARWNS ..(1130) .. SVKIEISDIVQR]

8.4.5 The SearchDb Function

It is easy to search for all entries in a sequence database that contain a
specific pattern. The function

SearchDb(pat1, pat2, ..., patk : { string, set(string) })

accepts a sequence of patterns where each pattern is either a string or a set
of string objects. When multiple parameters are contained in a set, the
SearchDb returns the logical OR of the results, that is, it returns all Entry
that contain any one of the patterns contained in the set. The comma
symbol represents the logical AND of the arguments. In this case, SearchDb
returns only those Entry that contain all such patterns. SearchDb searches
the database currently assigned to the system variable DB.

> DB := ReadDb(‘Sample/SH2’): # load the SH2 database
    # & assign it to DB
> SearchDb(‘mouse’); # returns all Entry() which contain
    # the pattern ’mouse’.

Entry(7,8,10,13,17,22,26,32,34,37,39,47,64,67,69,73,78)

> SearchDb(‘SH3’);

Entry(1,2,3,4,5,7,8,9,10,11,12,13,14,16,17,18,19,21,22,
23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,
> SearchDb('mouse', 'SH3'); # return all Entry() which contain
  # both 'mouse' and 'SH3'.
Entry(7,8,10,13,17,22,26,32,34,37,39,47,67,69,73)

> SearchDb({'mouse', 'human'}); # return all Entry() containing
> # human or mouse
Entry(2,3,7,8,9,10,12,13,16,17,21,22,23,25,26,30,31,32,
    33,34,36,37,38,39,41,47,53,63,64,66,67,68,69,72,73,77,78)

> SearchDb('SH3', {'mouse', 'human'}); # return all Entry() containing
> # (SH3 and (mouse or human))
Entry(2,3,7,8,9,10,12,13,16,17,21,22,23,25,26,30,31,32,
    33,34,36,37,38,39,41,47,53,66,67,68,69,72,73)

The SearchDb function is case insensitive, thus searching for pattern 'mouse'
is equivalent to searching for 'MOUSE'. Any pattern is allowed as a parameter
to the SearchDb function. We can even search for peptide sequences.

> SearchDb('NGGYI'); # a segment of the SH2 domain
Entry(8,24,25,26,27,35,38,39,40,43,70,71,72,73,74,75)

Note, however, that this is an exact match – in computer science terms – a
common substring search.

8.5 Indexing a Darwin Sequence Database

The SearchDb function can search the entire DB sequence database pro-
vided that the entire database is loaded into memory. With extremely
large databases this can be a problem; this is particularly true with DNA
databases such as EMBL that are on the order of hundreds of megabytes. Even with moderate-sized databases, searches via SearchDb can be rather slow. The following total CPU times were recorded for searching the SwissProt database for patterns via SearchDb.

```plaintext
> rtime(SearchDb('hello'));                      # a pattern which is not a DNA, RNA or AA sequence.
> rtime(SearchDb('aaa'));                        # a common pattern takes a long time
> rtime(SearchDb('SLVHRLIKDRIPANNDIYVLKGDLY'));  # a AA sequence
> rtime(SearchDb('P30376'));                     # searching for an accession number
> rtime(SearchDb('143Z_SHEEP'));                 # searching for an identification name
```

For commonly searched for strings, such as accession numbers and entry IDs, this sluggishness can be an annoyance.

To circumvent these problems, Darwin offers routines to create grid files.

A grid file... DESCRIPTION GOES HERE

We present a short Darwin program to create a grid file indexed by the AC (accession number) and ID (identification) field of the Swiss-Prot database. An extended version of what follows can be found in the Darwin library function CreateSpGrid.

Calling Sequence:

```
CreateSpGrid(filename)
```

Parameters:

```
filename : name
```

Returns: Builds a grid file indexed by the ID (identification) field and AC (accession number) field of the database located at system variable DB. It stores this grid file in external filename.

```
IndexDB := proc( filename : name )
    format := SetGridFile(ID=string, AC=string, Start=integer, End=integer);
    # create a structure of type GridfileFormat
```

gf := CreateGrid(filename, format);

for i from 1 to DB[TotEntries] do
    entry := GFstructure();
    holder := op(String(Entry(i)));
    entry['ID'] := SearchTag('ID', holder);
    AC := SearchTag('AC', holder);
    entry['AC'] := ac[1..CaseSearchString(';', AC)];
    entry['StartOffset'] := DB[Entry, i];
    entry['EndOffset'] := If( i < DB[TotEntries], DB[Entry, i+1]-1, DB[TotChars] );
    AddGrid( gf, entry );
    od;
CompressGrid(gf);
CloseGrid(gf);
end:

8.6 References

[3, 4, 10, ?, 14]
Chapter 9

Randomization, Statistics and Visualization

In Chapter ?? – *Generating Random Sequences* we combine randomization with statistical analyses to test the accuracy of our algorithms and hypotheses. By creating random amino acid sequences and feeding these as input to our programs for phylogenetic trees, we can obtain expected values and variances for the random cases. Comparing these results to the values for actual sequences, we can conclude that the results from our algorithms are either marginal, significant or insignificant.

There are many built-in Darwin tools for creating graphical representations of standard statistical objects such as dot plots and histograms. We introduce these functions here and discuss the system variables associated with these routines.

We also present the basic functions for creating pictures of binary trees. The more specific options associated with these functions are omitted from this discussion and introduced in Chapter 24 – *Phylogenetic Trees* where we use them extensively.
9.1 Randomization

This section explores some of the basic Darwin functions for randomization.

A number generator starts with an initial number $s_0$ and produces a sequence of numbers $s_1, s_2, \ldots$. This initial number $s_0$ is called the seed for the sequence. If the number generator was producing a truly random sequence of numbers, then having seen the partial sequence $s_0, s_1, s_2, \ldots, s_i$, for any $i$, it is impossible to predict the number $s_{i+1}$.

Producing truly random sequences is impossible or at least extremely difficult (and hints at some deep philosophic issues). Instead, most languages provide functions to generate pseudo random numbers. Pseudo-random sequences also begin with a seed $s_0$ and produce a sequence $s_1, s_2, \ldots$. However, pseudo-random sequences have a period, that is, there exists an $i$ such that $s_i = s_0, s_{i+1} = s_1, \ldots$ and it is theoretically possible to invert the generation function which takes $s_0, s_1, \ldots, s_{i-1}$ and produces $s_i$. In other words, we could deduce the $s_i$ having seen $s_0, s_1, \ldots, s_{i-1}$. In practice, for most applications, these pseudo random generators are more than sufficient.

The Random Function  In Darwin, the `Rand` function begins with a pre-defined seed and produces a pseudo-random sequence of numbers between $0 \ldots 1$.

```r
> Rand();
0.03219977
> Rand();
0.3408
> Rand();
0.7549
```

If we are to quit, and then restart a Darwin session, we would see that entering

```r
> Rand();
0.03219977
```
> Rand();
0.3408
> Rand();
0.7549

produces the exact same sequence. This is a nice property which can be exploited when debugging programs that make use of randomization as it allows us to compare successive executions of our program on the same input values.

**Planting a Seed**  The `SetRand` function allows users to define their own seed value.

> # restart Darwin
> SetRand(5);
> Rand();
0.3377
> Rand();
0.5751

**Random Seeding**  If instead we would like Darwin to choose a different seed value (besides the predefined seed value), the command `SetRandSeed` can be used. This command generates a seed value as a function of the current date and time of the system clock.

> # restart Darwin
> SetRandSeed();
> Rand();
0.9637

**Random Permutations**  A permutation of a set $P$ is an ordered sequence of all the elements of $P$, with each element appearing exactly once. For example, there are six permutations of the set $P = \{1, 2, 3\}$

$< 1, 2, 3 >, < 1, 3, 2 >, < 2, 1, 3 >, < 2, 3, 1 >, < 3, 1, 2 >, < 3, 2, 1 >$
The CreateRandPermutation function creates a random permutation of the set \( P_n = \{1, 2, \ldots, n\} \).

\[
\begin{align*}
&\text{CreateRandPermutation(5)}; \\
&\{1, 2, 4, 3, 5\} \\
&\text{CreateRandPermutation(5)}; \\
&\{5, 2, 3, 4, 1\}
\end{align*}
\]

Note that CreateRandPermutation uses the same predefined seed as the function Rand. Issuing either SetRand or SetRandSeed changes the sequence.

**Scrambling Strings** The Scramble function produces a permutation of a string.

\[
\begin{align*}
&\text{Scramble('I am all mixed up');} \\
&\text{I unp mmlIdi aax} \\
&\text{Scramble('I am all mixed up');} \\
&\text{u pm la liIda xem}
\end{align*}
\]

Like the CreateRandPermutation function, Scramble uses the same seed as the function Rand. Issuing either SetRand or SetRandSeed effects the generated sequence.

## 9.2 Basic Visualization Functions

Darwin is capable of producing various types of plots and graphs which can later be viewed outside of the Darwin environment. It can produce either a file in the PostScript format [20] or in the Unix tek protocol [26]. Most printers and plotters support at least one of these popular formats.

### 9.2.1 \texttt{set(plotoutput)}

By default, any plots and graphs generated by Darwin are sent to the file \texttt{temp.ps} in the directory where your Darwin session was started. We can
9.2. BASIC VISUALIZATION FUNCTIONS

re-direct such output to any file by setting the plotoutput system variable via the Set command (§11).

> Set(plotoutput='~/hallett/genetic.ps');

9.2.2 Set(plotdevice)

To specify which protocol we wish Darwin to use when generating graphs and plots, we set the system variable plotdevice via the Set command (§11.2).

The option tek causes Darwin to produce code compatible with the Unix functions graph, tplot and DrawPlot. Only simple line drawings can be produced by Darwin for this format.

> Set(plotdevice=tek);

Darwin will produce PostScript code in portrait format (7.5” wide and 10.5” high without scaling) when placed in psportrait mode.

> Set(plotdevice=psportrait);

PostScript code in landscape format (10.5” wide by 7.5” high without scaling) is specified by either pslandscape or ps.

> Set(plotdevice=pslandscape);
> Set(plotdevice=ps);

By default, the plotdevice is set to pslandscape.

9.2.3 Creating Plots

The DrawPlot command in Darwin can be used to generate everything from simple dot plots to arbitrarily complex shapes. The following section explores the different forms for this command.
CHAPTER 9. RANDOMIZATION, STATISTICS AND VISUALIZATION

Plotting a Set of Points

There are two ways in which we can plot a set of points in Darwin. We can plot a set of magnitudes by using \texttt{DrawPlot} as follows:

\begin{verbatim}
DrawPlot(t : list(real))
\end{verbatim}

Darwin connects consecutive points of \( t \) with a line. The \( x \)-axis is labeled with \texttt{1..length(t)} and the \( y \)-axis is labeled with the elements of \( t \).

\begin{verbatim}
> DrawPlot([0.01, 0.08, 0.21, 0.41, 0.19, 0.09, 0.02]);
\end{verbatim}

We can now view and print the results by executing the commands

\begin{verbatim}
% ghostview temp.ps
% lpr temp.ps
\end{verbatim}

directly from within the Unix environment. Other operating systems will require different commands for viewing and printing your graphs. From within Darwin, we can view the contents of \texttt{temp.ps} by executing

\begin{verbatim}
> ViewPlot();
\end{verbatim}

This command only displays the file named \texttt{temp.ps} located in the current directory.

If, instead, we would rather plot a set of points in the plane, we use \texttt{DrawPlot} as follows:

\begin{verbatim}
DrawPlot(t : list(real, real))
\end{verbatim}

Each element \( t = [u, v] \) specifies a point in the Cartesian (\( xy \)-plane). Consecutive points are connected with a line.

\begin{verbatim}
> sierpinsky := [[0,0], [1,-1], [3,-1], [4,0], [5,-1], [4,-2],

> [4,-4], [5,-5], [4,-6], [3, -5], [1, -5], [0,-6], [-1,-5],

> [0,-4], [0,-2], [-1,-1], [0,0]];

> DrawPlot(sierpinsky);
\end{verbatim}

The plot of \texttt{sierpinsky} is shown in Figure 9.1.
Figure 9.1: A plot of points in the Cartesian plane. This basic figure is used to generate the Sierpinsky series.
Plotting a Function and Range

\texttt{DrawPlot}(fnc : \texttt{procedure}, r : \texttt{range})

The range \( r \) labels the \( x \)-axis and the function determines the values for the \( y \)-axis. We show a simple example which plots the \texttt{sin} function for the range \(-5..5\).

\begin{verbatim}
> my_sine := proc(x)
> return(sin(x));
> end;
> DrawPlot( my_sine, -5..5);
\end{verbatim}

In the above example, we defined a function \texttt{my_sine} which tells \texttt{Darwin} how to treat each point in the range \(-5..5\). \texttt{Darwin} allows us the following short hand notation.

\begin{verbatim}
> DrawPlot(x -> sin(x), -5..5);
\end{verbatim}

The plot is shown in Figure 9.2.

\textbf{Superimposing Plots}

It is easy to superimpose different plots by passing multiple functions as a set to \texttt{DrawPlot}.

\begin{verbatim}
DrawPlot(\{\(x_1, x_2, \ldots, x_k\}\}, r)
\end{verbatim}

Each \( x_i \) must either be a \texttt{procedure} or it must consist of a list of points. The following example compares a set of points given by the list \texttt{distr} with its normal distribution.

\begin{verbatim}
> # the first distribution
> distr := [0.01, 0.08, 0.21, 0.41, 0.19, 0.09, 0.02];

> # its normal distribution
> mean := 3.03: variance := 1.35:
> DrawPlot( \{distr,
> x -> exp(-(x-mean)^2/variance) / sqrt(2*Pi*variance)},
> 0..6);
\end{verbatim}

The result is shown in Figure 9.3.
9.2. BASIC VISUALIZATION FUNCTIONS

Figure 9.2: A graph of the function $\sin(x)$ for range $-5..5$.

Figure 9.3: A comparison of two distributions.
More Complicated Plots

The final version of the DrawPlot command allows for text, lines or circles to be drawn. The parameters passed to this form of the DrawPlot command are somewhat more complicated than the examples above.

\textbf{DrawPlot}(\textit{obj} : \textit{set})

Each element of the set \textit{obj} must consist of a \textit{LTEXT}, \textit{CTEXT}, \textit{RTEXT}, \textit{LINE}, or \textit{CIRCLE} structure. Table 9.1 gives a description and argument list for each.

\begin{verbatim}
<table>
<thead>
<tr>
<th>Function</th>
<th>Parameters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LTEXT</td>
<td>(x : real, y : real, txt : string)</td>
<td>Aligns leftmost point of txt at point (x, y).</td>
</tr>
<tr>
<td>CTEXT</td>
<td>(x : real, y : real, txt : string)</td>
<td>Centers txt at point (x, y).</td>
</tr>
<tr>
<td>RTEXT</td>
<td>(x : real, y : real, txt : string)</td>
<td>Aligns rightmost point of txt at point (x, y).</td>
</tr>
<tr>
<td>LINE</td>
<td>(u : real, v : real, x : real, y : real)</td>
<td>Places a line between point (u, v) and (x, y).</td>
</tr>
<tr>
<td>CIRCLE</td>
<td>(x : real, y : real, r : real)</td>
<td>Places a circle at point (x, y) with radius \textit{r}.</td>
</tr>
</tbody>
</table>
\end{verbatim}

Table 9.1: The shape parameters for the \textbf{DrawPlot} function.

\textit{Darwin} can not produce \texttt{tek} code for these types of plots. Only \texttt{PostScript} code can be generated. Therefore, the system variable \texttt{plotdevice} must be assigned either \texttt{psportrait}, \texttt{pslandscape}, or \texttt{ps}.

As an illustration, we draw a chain consisting of 801 overlapping circles placed at the points \((x, \sin(x))\), for \(-10 \leq x \leq 10\).

```plaintext
> pts := CreateArray(1..801):  \# allocate an array to store the points
> for i from 0 to 800 do
>    \# calculate next centroid.
>    scratch := -10 + i * 20 / 800:
>    \# create a circle of radius 10.
>    pts[i+1] := CIRCLE(scratch, sin(scratch), 10);
```
Figure 9.4: A figure showing 801 overlapping circles drawn along the sine curve.

\begin{verbatim}
> od:
> # add the label
> pts := [op(pts), CTEXT(-1.3, 1, 'Circles and Sine')];
> DrawPlot(pts):
\end{verbatim}

The results of this short program are shown in Figure 9.4.

### 9.2.4 Creating Dot Plots

### 9.2.5 Creating Histograms

The function

\begin{verbatim}
DrawHistogram(y : list(real) \{,lbl : list(string)\})
\end{verbatim}

Given a list of real numbers, Darwin creates a histogram scaled appropriately. The list of labels \textit{lbl} is optional. As an example, we calculate the frequency of each of the twenty amino acids in our small version of Swiss-Prot Sample/SH2.
Figure 9.5: Amino acid frequencies in the Sample/SH2 database.

DB := ReadDb('Sample/SH2');  # load the SH2 database
labels := CreateArray(1..20);  # create an array of labels
for i from 1 to 20 do labels[i] := IntToAAA(i) od:  # and assign each a name
counts := GetAaCount(DB);  # count the bases
DrawHistogram(counts, labels);  # create the histogram

Figure 9.5 contains the results of our experiment.

9.2.6 Drawing Trees

The mathematical objects complete binary trees are often used for modelling phylogenetic trees. Darwin contains a number of routines to generate plots of these objects. Phylogenetic trees come in many flavours and these will be explored in Chapter 24 - Phylogenetic Trees in much greater depth. This section shows the basic functions available for generating graphical representations of rooted and unrooted trees.
### 9.2. Basic Visualization Functions

#### Rooted Binary Trees

The following function

\[
\text{DrawTree}(t : \text{Tree}, \text{title} : \text{string})
\]

draws the rooted binary tree specified by \(t\). In Darwin, we can define tree structures using the structured types \text{Tree} and \text{Leaf}. Here each vertex (node or point) in our \text{Tree} structure consists of a left child, a distance from the root of the tree to this vertex and a right child, in that order. The left and right children consist of either another \text{Tree} structure or a \text{Leaf} structure. A \text{Leaf} structure contains a label followed by the distance from it to the root of the tree. The following code creates a Darwin tree for the tree shown in Figure 9.6.

```latex
> f := \text{Tree}(\text{Leaf}(\text{'A'}, 33), 18, \text{Leaf}(\text{'B'}, 25)); \\
> g := \text{Tree}(\text{Leaf}(\text{'C'}, 33), 25, \text{Leaf}(\text{'D'}, 32)); \\
> h := \text{Tree}(g, 6, \text{Leaf}(\text{'E'}, 31)); \\
> \text{root} := \text{Tree}(f, 0, h);

> \text{DrawTree(root, ' A small tree ')};
```

Often it is more convenient to draw phylogenetic trees "upside down". To accomplish this, we negate the distances of our tree. Figure ??.

```latex
> f := \text{Tree}(\text{Leaf}(\text{'A'}, -33), -18, \text{Leaf}(\text{'B'}, -25)); \\
> g := \text{Tree}(\text{Leaf}(\text{'C'}, -33), -25, \text{Leaf}(\text{'D'}, -32)); \\
> h := \text{Tree}(g, -6, \text{Leaf}(\text{'E'}, -31)); \\
> \text{root} := \text{Tree}(f, 0, h);

> \text{DrawTree(root, 'An upside down tree')};
```

#### Unrooted Binary Tree

The following function

\[
\text{DarwinUnrootedTree}(t : \text{Tree}, \text{title} : \text{string})
\]

generates the unrooted binary tree specified by the \text{Tree} structure \(t\). Fig-
9.3. STATISTICS

ure ?? contains the unrooted version of the tree from Figure ???. There trees
are also referred to as circular divergence trees.

9.2.7 Annotating Plots and Graphs

9.3 Statistics

9.3.1 Calculating an Exponential Fit

9.3.2 Linear Regressions

9.3.3 Single Value Decompositions

9.3.4 The stats command

9.3.5 The updateStat command

9.3.6 Generating Distributions

9.3.7 Smoothing Data Points
An unrooted tree

Figure 9.7: An example of a unrooted tree in Darwin. The small circle represents the hypothesized location of the root.
Chapter 10

Polymorphism

The preceding chapters have already introduced a number of procedures which act differently depending on the number and type of the parameters passed to it. As another concrete example, consider the Darwin procedure print.

\[ > \text{print}(5.78); \quad \text{# argument is of type real} \]

5.7800

\[ > \text{print}('a \text{ string}'); \quad \text{# argument is of type string.} \]

a string

\[ > \text{print}([1, 2, 3, 4]); \quad \text{# argument is a list} \]

[1, 2, 3, 4]

\[ > \text{print}([[[1, 2], [3, 4]]]); \quad \text{# argument is a square matrix} \]

\[
\begin{array}{cc}
1 & 2 \\
3 & 4
\end{array}
\]

Notice that Darwin differentiates between the list [1, 2, 3, 4] and the square matrix [[[1, 2], [3, 4]]] and chooses a nicer format for the latter. This behavior indicates that \text{print} must examine the type of the argument and we could imagine a procedure deep inside the machinery of Darwin that looks something like this

\[
\text{print} := \text{proc}() 
\]

153
for i from 1 to length(args) do
  if (type( args[i], real )) then
    call the routine for printing out real values
  elif (type( args[i], string )) then
    call the routine for printing out string strings
  elif (type( args[i], list )) then
    if (type( args[i], matrix )) then
      call routine to print out a square matrix
    else
      call routine to print out a list
    fi;
  .
  . (do this for every different type in Darwin)
  .
end:

This is all well and fine for the built-in data types but how should Darwin determine how to treat new user defined data structures? The answer is to allow the user to write their own specially tailored version of print for their data type. We shall write a simple print function for our example ProEntry from Chapter 5 on page 5.1.

> ProEntry_print := proc( )
>  protent:=args;
>  lprint('\nName: ', protent[1], ' Organism: ', protent[4]);
>  lprint('DB: ', protent[2], ' Accession Number: ', protent[3]);
>  lprint('Sequence length: ', protent[8], '\n
');
>  for i from 1 to protent[6] do
>    if (mod(i,50)=0) then
>      printf(‘\n  ‘);
>    fi;
>    printf('%c',protent[5][i]);
>  od;
>  printf(‘\n’);
The procedure name `ProEntry_print` was not an arbitrary choice. The
general format `DataType.print` indicates to Darwin that this is a new form
of the general procedure `print` which is to be called when the type of the
argument to `print` is `DataType`.

```plaintext
> celegans := ProEntry('ABL1_CAEL', 'SwissProt', 'P03949', 'C. ELEGANS',
       'NEWCEARLYSTRKNDASNQRRLGEIGWVPSNFIAPYNSLDK', 42);
> print(celegans);  # structure celegans has type ProEntry

Name:  ABL1_CAEL  Organism:  C. ELEGANS
DB:  SwissProt  Accession Number:  P03949
Sequence length:  42

NEWCEARLYSTRKNDASNQRRLGEIGWVPSNFIAPYNSLDK
```

Whereas most names refer to exactly one procedure or function, the name
`print` has now been assigned to a set of procedures. We say that `print` is
an overloaded name. Overloading is a form of what is called polymorphism.\(^1\)

In Darwin there are a handful of built-in overloaded names. These include
`print`, `HTML` (§ 14) and `select` (§ 5.3). In general, we can overload any name
by using the Darwin statement `option polymorphic`. The example below
sketches how to create a polymorphic function `frequency` which calculates
the frequency of each base in a sequence. We will have two versions: one for
dNA, and one for amino acid sequences. The first step towards creating a
polymorphic function `frequency` is to tell Darwin that the name `frequency`
is polymorphic. This is accomplished by creating a procedure which is empty
save for the command `option polymorphic`.

```plaintext
> frequency := proc( )  # an empty procedure which states
```

\(^1\)Polymorphism comes in two forms: (1) parameteric polymorphism and (2) overloading
on names or operators. Parameteric polymorphism has already been implicitly discussed
in §5.1. We note that all forms of polymorphism are possible in Darwin except overloading
on operators.
> option polymorphic;  # that frequency is polymorphic
> end;

There is no change in how we define our structured types nor in how we allocate structures for these types.

DNA := proc( )
    description 'A data structure to hold a DNA sequence';
    if ( nargs=0 ) then
        return(copy(noeval(DNA(''))));
    elif ( nargs=1 ) then
        return(copy(noeval(DNA(args))));
    else
        print( DNA );
        error(' Incorrect format in structure DNA ', args);
    fi;
end:

AA := proc( )
    description 'A data structure to hold an AA sequence';
    if ( nargs=0 ) then
        return(copy(noeval(AA(''))));
    elif ( nargs=1 ) then
        return(copy(noeval(AA(args))));
    else
        print( AA );
        error(' Incorrect format in structure AA ', args);
    fi;
end:

> x := DNA('ACCGACGGACTACCGAGAGTCCCA');
> y := AA('QHFPSTHEQCDNRAAAATGWYV');

The final step involves creating the appropriate versions of frequency. In this case, we require two such functions: DNAfrequency for the DNA type
and AA\texttt{frequency} for the AA type. We use two built-in Darwin functions \texttt{NToInt} and \texttt{AToInt}. These functions convert a DNA base into an integer between one and four and convert an amino acid into an integer between one and twenty respectively.

\begin{verbatim}
> DNA\_frequency := proc( seq )
>     total := CreateArray(1..4, 0);
>     for i from 1 to length(seq) do
>         total[NToInt(seq[i])] := total[NToInt(seq[i])] + 1;
>     od;                     # NToInt(x) returns an integer between 1 and 4
>     for i from 1 to length(total) do
>         total[i] := total[i]/length(seq);
>     od;
>     total;
> end:
>
> AA\_frequency := proc( seq )
>     # initialize a 20 element vector to zeros
>     total := CreateArray(1..20, 0);
>     for i from 1 to length(seq) do
>         total[AToInt(seq[i])] := total[AToInt(seq[i])] + 1;
>     od;
>     # AToInt(x) returns an integer between 1 and 20
>     for i from 1 to length(total) do
>         total[i] := total[i]/length(seq);
>     od;
>     total;
> end:
\end{verbatim}

To calculate the frequency for a sequence, we need only call the function \texttt{frequency} with the sequence. Darwin determines the type of the sequence and calls the appropriate version of the function.

\begin{verbatim}
> frequency(x);
\end{verbatim}
0.2917, 0.3750, 0.2500, 0.08333333]
> frequency(y);
[0.1364, 0.04545455, 0.04545455, 0.04545455, 0.04545455, 0.09090909, 0.04545455, 0, 0.09090909, 0, 0, 0, 0, 0.04545455, 0.09090909, 0.04545455, 0.09090909, 0.04545455, 0.04545455]

Readers familiar with languages such as C++, Java, and SmallTalk may see traces of object oriented programming. While strictly speaking, Darwin is not an object oriented programming language, the ability to overload on names, to pass a variable number of arguments to a routine and to make these routines polymorphic allow users to build programs in a similar style as to what you would expect to see in a purely object oriented language. The biggest disparity between Darwin and these languages is caused by the scoping rules which make it inconvenient to stay strictly within these paradigms. Nevertheless, the authors feel that the inclusion of these structures in Darwin allow one to create robust, easy to understand and error free programs. The file Sample/Entry contains a complete version of our structured type ProEntry that we hope may act as a prototype for new users.²

10.1 Object-relational Databases

10.2 Using the Database Built into Darwin

10.3 Using the Opera Facilities in Darwin

²See also the Computational Biochemistry Research Group web site [5].
Chapter 11

System Commands

The are several Darwin commands which allow one to communicate with the underlying operating system and to set Darwin internal variables. We explore these in the following section.

11.1 Communicating with the Operating System

The following commands are system dependent, therefore, the precise syntax for your system may differ slightly from those described below.

11.1.1 Date and Time

The command \texttt{date} returns the current date and time as a string.

\begin{verbatim}
> date();
\end{verbatim}

If you prefer to know the number of seconds since 00:00:00 GMT, January 1, 1970 in seconds, then issue the \texttt{rtime} command (running time) with no arguments.

\begin{verbatim}
> rtime();
\end{verbatim}

The \texttt{rtime} function can also except an expression as a parameter. In this form, \texttt{rtime} returns the time in seconds needed to evaluate this expression.
for i from 1 to 100001 by 500 do
  lprint(i, rtime(factorial(i))); od;

As an aside, note that the running time does not grow linearly with the size of the factorial we are computing. This is because the built-in factorial function approximates the true value for large numbers using a variant of the gamma function (see §?? in Part ??—The Reference Guide and [11]). Chapter 13 Measuring Performance explores this and related functions further.

11.1.2 Operating System Commands

CallSystem(cmd : string)
Darwin allows users to execute operating system commands. The CallSystem command accepts a single parameter cmd of type string. The contents of cmd will be executed by the underlying operating system and the value it returns is subsequently returned by the CallSystem command. This ability is especially useful for examining files before reading/writing to/from them. For example, by using the Unix [26] command ls, we can check if a specific file exists in the current directory.

> CallSystem('ls Sample/arrays');       # if it exists, 0 is returned.
> CallSystem('ls Sample/nofile');      # Unix returns value 512.

Chapter ?? offers an in-depth look at how to execute programs written in another language from within Darwin.

TimedCallSystem(cmd : string, timeout : posint)
The TimedCallSystem command operates in the same manner as the CallSystem command except it takes an optional second parameter timeout of type posint. When a second argument is supplied, Darwin passes the contents of cmd to the operating system for execution. If the operating system does not complete execution of the command within this bound, the result [-1, (Timeout)] is returned. If the operating system does respond within this
time bound, a list with two items is returned. The first element of this list is the value returned by the system command and the second element is a string containing the output generated by the system command. Chapter 16 – *Calling External Functions* gives an example application of using these and related commands.

### 11.1.3 Garbage Collection

During a computation, Darwin generates many *scratch work* expressions which the user never sees directly (they only see the final result of the statement they submitted). Over the course of a session, the memory needed for these computations can become very large. This, in turn, can cause your system to act sluggish. Fortunately, the memory used by these *scratch work* expressions is recyclable. Entering the statement

```latex
> gc();
```

tells Darwin to immediately coalesce all unused memory space. This can be a costly operation as Darwin must essentially travel through the entire chunk of allocated memory searching for fragments. By default, Darwin performs a garbage collection operation each time a program uses 250,000 words of memory. This can be changed by using the Set command and the internal variable gc (see § 11.2).

```latex
> Set(gc=500000);  # garbage collection every 500,000 words allocated
```

The *increment* (in this case 500,000) specifies how many words must be allocated before Darwin performs a garbage collection. For most small applications, users need never worry about setting the gc variable but if you should notice that your program is using an amount of space close to the total offered by the system, it is prudent to set gc lower than the default value or to explicitly include gc statements in your programs.

To suppress the garbage collection information Darwin displays, one must set the internal variable printgc to false.
11.2 Internal Variables

There are numerous system variables strewn throughout Darwin. A system variable is assigned a value by using the \texttt{Set} function. This function accepts as a parameter any one of the options in Table 11.1. This is followed by equal sign (=) which is, in turn, followed by a value for that option. The command returns the previous value assigned to this variable.

\begin{verbatim}
> x := Set(screenwidth=5); # things get really narrow now
> print('Heidi comes from Heidiland');
> Set(screenwidth=x); # everything back to normal
\end{verbatim}

\begin{table}[h]
\begin{tabular}{|l|l|p{6cm}|}
\hline
Name & Type & Description \\
\hline
echo & 0... & Sets the level of input/output information displayed. \\
\hline
gc & integer & Sets the frequency (in words allocated) for garbage collection. \\
\hline
mapsize & integer & Sets the minimum size (in chars) required for Darwin to build a .map file. \\
\hline
plotdevice & \textit{printing style} \(^a\) & Sets the protocol for subsequent print files. \\
\hline
plotoutput & filename & Specifies where files contain plotter code is sent. \\
\hline
printgc & boolean & Toggles displaying garbage collection information. \\
\hline
printlevel' & integer & Sets the amount of information printed out which is during execution. \\
\hline
profile & boolean & Toggles printer/plotter profile mode. \\
\hline
prompt & string & Sets the Darwin prompt. \\
\hline
quiet & boolean & Toggles the suppression of output. \\
\hline
server & boolean & Places Darwin in server mode. \\
\hline
screenwidth & pointer & Sets the width of a line for all subsequent output. \\
\hline
\end{tabular}
\caption{Options for the \texttt{Set} function.}
\end{table}

\(^a\)Choices include \texttt{ps, pslandscape, psporatit, tek}.
11.3 Options

The option command in Darwin flags procedure as being special in some way. Any such declarations must be placed immediately following any local and global variable declarations and preceding a description command.

```plaintext
> example := proc( )
>   global a, b, c;
>   local x, y, z;
>   option polymorphic;
>   description 'This procedure serves only as an example.';

   (body of procedure)
>
> end:
```

Chapter 10 shows how the option polymorphic command allows one to overload on the name of the routine. Section 4.1 describes how routines can automatically handle vector operations!!. Section 13 describes how option numeric can speed up purely mathematical functions and Section 12.1.2 introduces the option trace command for aiding in the process of debugging. The remaining option we have yet not seen is the option builtin command. This command is used to indicate to Darwin that the routine is a built-in routine, that is, it is located in the Darwin kernel. If you examine the file named darwinit located in the Darwin library, you will see the declaration and description line for every routine in Darwin. Some listed routines also contain the command option builtin. When a user invokes a routine, Darwin looks for that name in this list. If it finds it and it is accompanied by the option builtin command, it knows that this function has not been coded in Darwin. Users of Darwin can not add new built-in routines as this would require a re-compilation of the Darwin kernel.
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>builtin</td>
<td>Indicates that the routine is located in the kernel.</td>
</tr>
<tr>
<td>numeric</td>
<td>Can be used to optimize routines which only perform numeric operations.</td>
</tr>
<tr>
<td>polymorphic</td>
<td>Allows the name of the routine to be overloaded (§ 10).</td>
</tr>
<tr>
<td>trace</td>
<td>Directs Darwin to display the value of parameters, the result of every executed statement, and the return value of the procedure</td>
</tr>
<tr>
<td>zippable</td>
<td>Allows the routine to operate on either numeric values or vectors.</td>
</tr>
</tbody>
</table>

Table 11.2: The option command.
Chapter 12

A Guide to Debugging

You will inevitably make mistakes when writing Darwin programs. For the
most part, the error will quickly reveal itself and inserting the forgotten
semicolon or od command will resolve the standstill. However, there are
those unforgettable moments where you will, with a religious fervor, believe
you are correct and the misunderstanding due to a deep Darwin bug. This
may be the case and, if so, you should immediately send electronic mail to
the Computational Biochemistry Research Group at ETH–Zürich so we may examine the situation.

While waiting for the new release, you may want to try some of the fol-
lowing commands to help find these “subtle system errors”.

12.1 Tracing and the Print Level

Both option trace and printlevel are designed to print out step-by-step
details of the execution of your Darwin program. printlevel is an internal
variable that controls the amount of information printed out during execu-
tion. It has default value 1 but can be given any integer value via the Set
command (see § 11.2).
12.1.1 printlevel

When the system variable \texttt{printlevel} is set to the default of 1, only the results of Darwin statements and statements nested at depth one inside for, while and if commands are displayed. If \texttt{printlevel} is set to value 0, only Darwin statements which were directly entered to system are displayed. Given a negative value, \texttt{printlevel} displays only the result of print statements. Enter the following short segment of code after setting the \texttt{printlevel} variable to negative one, zero and then one and observe the differences in output it generates.

\begin{verbatim}
> lprint('Calculating the sum from 1..5);
> total := 0;
> for i to 5 do
>   total := total + i;
> od;
\end{verbatim}

Assigning larger values to \texttt{printlevel} causes larger amounts of information to be displayed. If \texttt{printlevel} is assigned two, then all statements executed within routines, if, for, and while commands nested at a depth less than or equal to two are displayed. At value three, all routines, if, for, and while commands nested at a depth less than or equal to three are displayed and so forth.

12.1.2 trace

The \texttt{trace} can be used in conjunction with the \texttt{option} command in Darwin (see §11.3) or it can be used as a Darwin command directly by passing a name of a procedure as a parameter. When applied to procedures, the \texttt{trace} command directs Darwin to display the result of every executed statement accompanied with the value of the parameters at the entry point to the routine and the value being returned. The \texttt{option trace} combination must follow any local and global variable declarations after the routine declaration.

\begin{verbatim}
> Sum := proc(val : real)
\end{verbatim}
12.1. TRACING AND THE PRINT LEVEL

> local total;
> option trace;
>       # must follow local and global variable declarations
> 
> total:=0;
> for i from 1 to val do
> total :=total+i;
> od;
> total;
> end;
> Sum(5);
{--> enter Sum, args = 5
total := 0
total := 3
total := 6
total := 10
total := 15
15
<-- exit Sum = 15} 
15

Alternatively, we could omit the option trace command and invoke trace as follows.

> Sum := proc( val : real )
> local total;
> 
> total:=0;
> for i from 1 to val do
> total :=total+i;
> od;
> total;
> end;
> trace(Sum);
12.2 Trapping Errors

Darwin offers two built-in functions which help users write programs which handle errors in a graceful manner and return meaningful, easy to understand messages. By default, Darwin immediately halts execution of a procedure when it encounters an error. It provides a brief message of what has transpired and waits for the user to resume entering statements. The traperror function allows us to override this behaviour.

\texttt{traperror(exp : expression)}

When an expression is “wrapped” inside of a traperror function, errors which occur during the evaluation of \texttt{exp} do not cause Darwin to halt execution. Instead, the error message is returned by the traperror function in the form of a string item. If no error occurs, the evaluation of \texttt{exp} is returned.

\begin{verbatim}
> msg := traperror( 100 / 20 );    # all is okay.
5
> msg := traperror( undefined_symbol / 20 );  # this is an error
msg := invalid term in product
\end{verbatim}

After the second call to traperror above, we can write code to examine the contents of \texttt{msg} and act accordingly. One option would be to halt the execution of Darwin (as it does by default) but include information about the assigned values to elements of \texttt{exp}. We can stop execution and print out these values via the error function.

\texttt{error(errormsgs)}

This command returns to the top level of execution and issues the error messages \texttt{errormsgs}.

\begin{verbatim}
> weird_function := proc( a, b )
>     msg := traperror( a ^ log( b ) );
>     if ( type( msg, string ) ) then     # an error occurred
>         error( 'An error occurred in weird_function. Arguments: ', a,b );
> \end{verbatim}
12.2. TRAPPING ERRORS

> fi;
> end:
>
> weird_function(1, d);
Error, (in weird_function)

An error occurred in weird_function. Arguments: , 1, d

We can exploit this tandem of functions to help aid in the debugging process. If you encounter a “fragile” routine displaying seemingly inconsistent behaviour, inserting traperror and error function calls within the code may greatly reduce the amount of time spent searching for the problem.
Chapter 13

Measuring Performance

Darwin offers several tools for measuring the CPU usage. These are especially convenient when attempting to optimize your code and make projections about the necessary time required to complete a long computation. Programs written in Darwin code can sometime run much slower than written in another language such as C. Unfortunately, writing programs in C can be a non-trivial exercise. This trade-off leads to questions about when Darwin code will suffice for the task at hand and when another language must be used. Measuring the performance of your Darwin prototypes is a necessary step towards resolving this issue. Chapter ?? contains a working example of how to go about this. This section explores some of the primitive functions Darwin offers.

13.1 option numeric

Routines which perform only mathematical calculations can be optimized in Darwin by using the option numeric command. Like all option commands, this must be placed inside of a procedure directly after any global and local variable declarations and preceding a description command.

\[ \text{Sum := proc( val )} \]
local total;
option numeric;
total := 0;
for i from 1 to val do
    total:=total+i;
od;
return(total);
od;

13.2 A Small Example

To show the difference in running time between a routine which includes the option numeric versus one that does not include this option, we iterate over a non-trivial mathematical expression and check the elapsed CPU time.

SmallExample := proc ( )
    local total;
    option numeric;

    total := 0;
    for i from 1 to 10000 do
        total := total  * (mod (i, 50)) - log(i) + sqrt(i);
    od;
    return(total);
end:
Chapter 14

Producing HTML Code

Not yet implemented.
Chapter 15

Darwin’s Interprocessor Skills
Chapter 16

Calling External Functions

Not written yet.
Part II

Darwin and Problems from Biochemistry
The second part of this book consists of a series of chapters, each of which is concerned with one particular problem from bioinformatics. The order of the chapters mirrors Darwin's *tower of information*. The tower is founded upon raw sequence data in the form of DNA or RNA. In terms of the biochemist's workbench, this is our raw material and it is with this that the bioinformaticist's job begins. Each successive "step up" the tower represents a movement away from the raw unprocessed information and closer to an complete understanding of how the sequences act as a functional unit in the living organism.

At the top of our tower lies the secondary structure prediction. This, in some sense, is the ultimate goal for our system and the end of the bioinformaticist's job. Our secondary structure predictions are based on multiple sequence alignments. These alignments indicate conserved/non-conserved areas of a protein and highlight the different structural units such as alpha helices and beta sheets. Of course, this implies that the accuracy of our structure predictions are dependent on the accuracy of our multiple sequence alignments. In turn, the accuracy of our alignments is dependent on how accurate our phylogenetic trees represent the true ancestral relationships between the species from which the sequences are taken. Our phylogenetic trees are constructed from the pairwise distances and variances derived from the pairwise comparison of protein sequences. And, at the bottom of our tower, the protein sequences are extracted from the raw DNA or RNA supplied to us by the biochemist.

Of course, any mistake at any level of this tower percolates upwards. But, conversely, any improvement to an algorithm does too.

We do not claim that the solutions present herein are the only or the best way to go about solving any particular bioinformatics problem. The algorithms we have chosen to include in the Darwin libraries have strong arguments, both mathematical and biological, suggesting they will perform well in practice. However, there are other methods (requiring possibly un-
realistic resource demands) that may be more pertinent to your particular situation and data. The strength of Darwin lies in the fact that any method (assuming it is algorithmic) can be programmed in the language.

Each of the following chapters contains:

- a statement of the problem at hand,
- a discussion concerning any biologic assumptions we make about the data,
- an explanation of how we model the problem mathematically,
- a description of the algorithm,
- a discussion about the accuracy and efficiency of our algorithm, and
- a short guide to the literature.
Beyond the understanding of the Darwin libraries, we hope such a presentation gives users

- an understanding of some of the classic problems from bioinformatics,

- an understanding of the underlying biochemistry involved in these problems,

- an understanding of the mathematical model upon which these algorithms are predicated,

- an understanding of how the algorithms works, and

- a conceptual overview of how to structure programs in Darwin.
Chapter 17

Modelling Evolution
Chapter 18

Point Accepted Mutations and Dayhoff Matrices
CHAPTER 18. POINT ACCEPTED MUTATIONS AND DAYHOFF MATRICES

This chapter is mainly concerned with the definition and construction of matrices which are used to score the quality of an alignment of two amino acid sequences. Typically, these similarity matrices contain a value proportional to the probability that amino acid $i$ mutates into amino acid $j$ for all pairs of amino acids.

The construction of such matrices is straightforward and natural. By examining a large sample of verified pairwise alignments of amino acids, we can extract frequency information of the form

amino acid $i$ mutated into amino acid $j$

If the sample is large enough to be statistically significant and contains a diverse range of example alignments, then the resulting matrices should reflect the true probabilities of mutations occurring through a period of evolution.

An alignment between two amino acid sequences might look as follows

\begin{align*}
\text{sequence1} & : A_1 \ A_2 \ A_3 \ A_4 \ A_5 \ A_6 \\
\text{sequence2} & : B_1 \ B_2 \ B_3 \ B_4 \ B_5 \ B_6
\end{align*}

where $A_i$ is an amino acid from sequence 1 aligned against amino acid $B_i$ in sequence 2. Each such matching between two amino acids is assigned a score from a similarity matrix. The score for the entire match is the sum of the scores of the individually matched amino acids. Here, one typically uses the Dayhoff similarity matrix [9] since the entries have some nice algebraic properties which are exploited by the algorithms.

The best (or maximum likely) alignment between two sequences is that alignment which is most probable, i.e., that alignment with the highest score relative to the Dayhoff matrix. Via the classic Needleman and Wunsch [23] dynamic programming algorithm, we are able to find this maximum quickly and efficiently. The construction of such alignments is explored in depth in Chapter 21 - The Pairwise Alignment of Sequences.

The following sections provide part of the ground work necessary for performing pairwise alignments. We begin by presenting our mathematical
model of evolution and measures for the amount of evolution. We discuss
the routines available in Darwin for the construction of the first Dayhoff
matrix [9] and explain how their method can be improved upon.

In this chapter we are solely concerned with the mutation events which
are point accepted mutations or, as they are sometimes referred to in the
literature, substitutions. Chapter 19 Insertions and Deletions describes our
model for the two other forms of mutation: insertions and deletions.

18.1 Modeling Evolution

A mutation matrix, denoted by $M$, describes the probabilities of amino acid
mutations for a given period of evolution.

$$\Pr\{\text{amino acid } i \rightarrow \text{amino acid } j\} = M_{ij}$$

The value $1 - M_{ii}$ then represents the probability of mutating away from
$i$.

The matrix $M$ corresponds to a model of evolution in which amino acids
mutate randomly and independently from one another but according to some
predefined probabilities. This is a Markovian model and, while simple, it is
one of the best methods available for modeling evolution.

There are two main assumptions implicit in such a model: (1) amino acid
substitutions subsequent in time are independent of preceding substitutions
and (2) substitutions at specific positions in the protein sequence are in-
dependent of substitutions elsewhere in the sequence. Intrinsisc properities
of amino acids, like hydrophobicity, size, charge, etc. can be modelled by
appropriate mutation matrices. However, we can not model dependencies
which relate one amino acid characteristic to the characteristics of its neigh-
bours. Of course, these assumption are not strictly true and readers are
referred to [17] where the authors perform an analysis of the magnitude of
this disparity.
Amino acids appear in nature with different frequencies. These frequencies are denoted by \( f_i \) and correspond to the steady state of the Markov process defined by the matrix \( M \), that is, the vector \( f \) is any of the columns of \( M^\infty \) or the eigenvector of \( M \) whose corresponding eigenvalue is 1 (\( M \cdot f = f \)).

We can not distinguish between a mutation from \( i \) into \( j \) and a mutation from \( j \) into \( i \). This implies a simple relation for the entries of \( M \):

\[
f_i \cdot M_{ij} = f_j \cdot M_{ji}
\]

### 18.1.1 Point Accepted Mutations

The definition of matrix \( M \) describes mutation over a given period of evolution. In order to proceed, we must quantify this change in a mathematically meaningful way. Dayhoff et. al.[9] introduced the term PAM (point accepted mutation) unit. A 1-PAM unit is the amount of evolution which will change, on average, 1% of the amino acids. In mathematical terms, this is expressed as a matrix \( M \) such that

\[
\sum_{i=1}^{20} f_i (1 - M_{ii}) = 0.01
\]

where \( f_i \) is the frequency of the \( i^{th} \) amino acid. (Recall that \( M_{ii} \) represents the probability amino acid \( i \) does not change.)

If we have a probability or frequency vector \( p \), the product \( M \cdot p \) gives the probability vector (or the expected frequency of \( p \)) after an evolution equivalent to 1-PAM unit.

Alternatively, if we start with amino acid \( i \) (a probability vector which contains a 1 in position \( i \) and 0s in all others), \( M \cdot p = M_{si} \) (the \( i^{th} \) column of \( M \)) is the corresponding probability vector after one unit of random evolution.

After \( k \) units of evolution (a \( k \)-PAM evolution), a frequency vector \( p \) will be changed into the frequency vector \( M^k \cdot p \).
Note that PAM distance does not correlate in any immediate way to \textit{chronological time}. Evolutionary rates may be very different between species and proteins.

18.1.2 Dayhoff Matrices

For reasons both historical and algebraic, the mutation matrix is transformed into a new matrix termed a \textit{Dayhoff matrix} (in honour of the first author, Margaret O. Dayhoff). The Dayhoff matrix, $D$, is related to a 250-PAM mutation matrix by

$$D_{ij} = 10 \cdot \log_{10} \left( \frac{M^{250}_{ij}}{f_i} \right)$$

A 250-PAM distance corresponds to approximately 17\% identity between two sequences (see §?? for the proof). Many believe this distance to be at the limit of our ability to detect homology based on sequence data alone.

Aligning sequences by dynamic programming using Dayhoff matrices is equivalent to finding the alignment which maximizes the probability that the two sequences evolved from a common ancestor as opposed to being random sequences. We are comparing two events:

a) that the two sequences are independent of each other, and hence an arbitrary position with amino acid $i$ aligned to another arbitrary position with amino acid $j$ has the probability equal to the product of the individual frequencies

$$Pr\{\text{independent alignment of } i \text{ and } j\} = f_i f_j$$

b) that the two sequences have evolved from some common ancestral sequence after some amount, $t$, of evolution.

$$Pr\{i \text{ and } j \text{ from a common ancestor } x\} = \sum_x f_x Pr\{x \rightarrow i\} Pr\{x \rightarrow j\}$$

$$= \sum_x f_x (M')_{ix} (M')_{jx}$$
\[
\begin{align*}
\sum_{x} f_{j}(M_{ij}) & = \sum_{x} f_{j}(M^{t}_{ij}) \\
& = f_{j}(M^{t}_{ij}) = f_{i}(M^{t}_{ji})
\end{align*}
\]

We use \( \Sigma \) as a shorthand notation for \( \Sigma_{i \in A} \) where \( A \) is the alphabet of amino acids.

The entries of the Dayhoff matrix are ten times the logarithm of the quotient of these two probabilities.

\[
D_{ij} = 10 \cdot \log_{10} \left( \frac{Pr\{i \text{ and } j \text{ descended from } x\}}{Pr\{i \text{ and } j \text{ are independent}\}} \right)
\]

(The factor of 10 is included for purely historical reasons.)

Since dynamic programming maximizes the sum of the similarity measure, dynamic programming maximizes the sum of the logarithms or the product of these quotients. Therefore, dynamic programming finds the alignment which maximizes the probability of having evolved from a common ancestor (a maximum likelihood alignment) against the null hypothesis of being independent.

### 18.1.3 Interpreting Scores

The resulting measure of similarity after a matching is a sum of Dayhoff entries, and hence it is 10 times the logarithm of this probability. For example, a matching between two amino acid sequences with a similarity (or cost or score) of 238.8 means that the probability of both sequences coming from a common ancestor, as opposed to being a random alignment, is \( 10^{23.88} \) times more likely. Although crude, this gives a rule of thumb for estimating the quality of a matching.

This fact makes aligning with Dayhoff matrices a soundly based algorithm. This has been noted by many people including [2, 9, 12, 13] yet such methodologies have been largely ignored.
18.2 The Original Dayhoff Matrices

In the late 1960s and 1970s, Dayhoff et. al. published a series of papers containing the first similarity matrices. As more sequence data (a larger statistical sample) became available, they repeated their construction making more and more accurate estimations. In the 1978 paper, Dayhoff, Schwartz and Orcutt[9] examined 1572 accepted mutations between 34 superfamilies of closely related sequences.

We can recompute the original Dayhoff matrix using the function

Calling Sequences:

\begin{verbatim}
CreateOrigDayMatrix(mutations, counts, PAM)
CreateOrigDayMatrix(mutations, counts, 1..UpperPam)
\end{verbatim}

Parameters:

- \texttt{mutations} : array(real, real)
- \texttt{counts} : array(real)
- \texttt{PAM, UpperPam} : real

Returns: \texttt{DayMatrix}

\textbf{Synopsis}: This function returns the Dayhoff matrix (structured type \texttt{DayMatrix}) computed from a given observed mutation matrix \texttt{mutations}, a frequency vector \texttt{counts} and a PAM distance \texttt{PAM} (or range of PAM distances beginning at 1).

The Darwin built-in matrix \texttt{Mutations1978} contains the observed mutation count report by Dayhoff et. al. The vector \texttt{OrigFreq} contains entries proportional to the reported frequencies.

\begin{verbatim}
> print(Mutations1978);
> OrigTot := [87, 41, 40, 47, 33, 38, 50, 89, 34, 37,
>             85, 81, 15, 40, 51, 70, 58, 10, 30, 65];
\end{verbatim}
> OrigFreq := OrigTot/sum(OrigTot);

It is not clear whether the vector OrigFreq or the amino acid frequencies for the entire database should be used in the computation of the Dayhoff matrix. We compare the difference between OrigFreq and the frequencies for Swiss-Prot version 33. The function GetAaCount(DB) returns a list containing the number of appearances of each amino acid in the database.¹

Many frequencies are significantly different.

> DB := ReadDb('cbrg/DB/SwissProt');
> SP33Totals := GetAaCount(DB);
> SP33Freq := SP33Totals/sum(SP33Totals);
> printf('%6s %6.1f%% %6.1f%%
', 'Orig SP33');
> for i to 20 do
> printf('%6s %6.1f%% %6.1f%%
', IntToAmino(i), 100*OrigFreq[i], 100*SP33Freq[i]);
> od;

Alanine 8.69% 7.55%
Arginine 4.10% 5.16%
Asparagine 4.00% 4.55%
Aspartic acid 4.70% 5.30%
Cysteine 3.30% 1.70%
Glutamine 3.80% 4.03%
Glutamic acid 5.00% 6.32%
Glycine 8.89% 6.86%
Histidine 3.40% 2.23%
Isoleucine 3.70% 5.73%
Leucine 8.49% 9.32%
Lysine 8.09% 5.95%
Methionine 1.50% 2.36%
Phenylalanine 4.00% 4.07%
Proline 5.09% 4.92%

¹The GetAaCount function requires that a patricia tree has been created for the database assigned to DB (a non-empty .tree file). See §8.3.
18.2. THE ORIGINAL DAYHOFF MATRICES

<table>
<thead>
<tr>
<th>Amino Acid</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serine</td>
<td>6.99%</td>
</tr>
<tr>
<td>Threonine</td>
<td>5.79%</td>
</tr>
<tr>
<td>Tryptophan</td>
<td>1.00%</td>
</tr>
<tr>
<td>Tyrosine</td>
<td>3.00%</td>
</tr>
<tr>
<td>Valine</td>
<td>6.49%</td>
</tr>
</tbody>
</table>

We compute the Dayhoff matrices both with OrigFreq and SP33Freq at a PAM distance of 250 (a long distance).

> OrigDM := CreateOrigDayMatrix(Mutations1978, OrigFreq, 250);
> SPDM := CreateOrigDayMatrix(Mutations1978, SP33Freq, 250);

Table 18.1 contains a list of selectors for the DayMatrix structured type.

<table>
<thead>
<tr>
<th>Selector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FixedDel</td>
<td>Adjusted deletion penalty.</td>
</tr>
<tr>
<td>DelFixedLog</td>
<td>Logarithmic deletion penalty.</td>
</tr>
<tr>
<td>IncDel</td>
<td>Deletion penalty increment.</td>
</tr>
<tr>
<td>MaxOffDiag</td>
<td>Max. value not on the diagonal.</td>
</tr>
<tr>
<td>MinSim</td>
<td>Min. value in the Dayhoff matrix.</td>
</tr>
<tr>
<td>PmNumber</td>
<td>PAM distance for which this matrix was computed.</td>
</tr>
<tr>
<td>Sim, i, j</td>
<td>The similarity score for amino acid i and amino acid j.</td>
</tr>
<tr>
<td>type</td>
<td>The type of the Dayhoff matrix. ie. Peptide, DNA.</td>
</tr>
</tbody>
</table>

Table 18.1: The selectors for the DayMatrix structured type.

Comparing the two matrices shows some significant differences. These will somewhat change the results of our alignment algorithms.

> print(OrigDM);
DayMatrix(Peptide, pm=250, Sim: max=17.302, min=-7.510, max offdiag=6.951, del=-19.814-1.396*(k-1))
C 12.0
S  -0.0  1.6
T  -2.2  1.3  2.6
P  -2.7  0.9  0.3  5.9
A  -2.0  1.1  1.2  1.1  1.8
G  -3.3  1.1  -0.0  -0.5  1.3  4.8
N  -3.6  0.7  0.4  -0.5  0.2  0.4  2.0
D  -5.1  0.3  -0.1  -1.0  0.3  0.6  2.1  3.9
E  -5.3  -0.0  -0.4  -0.6  0.3  0.2  1.4  3.4  3.9
Q  -5.3  -0.5  -0.8  0.2  -0.4  -1.2  0.8  1.6  2.5  4.1
H  -3.4  -0.8  -1.3  -0.3  -1.4  -2.1  1.6  0.7  0.6  2.9  6.6
R  -3.6  -0.3  -0.9  -0.2  -1.6  -2.6  -0.0  -1.3  -1.1  1.2  1.5  6.1
K  -5.4  -0.2  -0.0  -1.2  -1.2  -1.7  1.0  0.1  -0.1  0.7  -0.1  3.4  4.7
M  -5.2  -1.6  -0.6  -2.1  -1.2  -2.8  -1.8  -2.6  -2.2  -1.0  -2.2  -0.5  0.4  6.6
I  -2.3  -1.4  0.1  -2.0  -0.5  -2.6  -1.8  -2.4  -2.0  -2.5  -2.0  -1.9  2.2  4.6
L  -6.0  -2.8  -1.7  -2.6  -1.9  -4.0  -2.9  -4.0  -3.3  -1.8  -2.1  -3.0  -2.9  3.7  6.0
V  -1.9  -1.0  0.3  -1.2  0.2  -1.4  -1.8  -2.2  -1.6  -1.9  -2.3  -2.5  -2.5  1.8  3.7  1.8  4.3
F  -4.3  -3.2  -3.1  -4.6  -3.5  -4.8  -3.5  -5.6  -5.4  -4.7  -1.8  -4.5  -5.3  0.2  1.0  1.8  -1.2  9.1
Y  0.4  -2.8  -2.8  -5.0  -3.5  -5.3  -2.1  -4.3  -4.3  -4.0  -0.1  -4.2  -4.5  -2.5  -1.0  -0.9  -2.5  7.0  10.0
W  -7.5  -2.3  -5.0  -5.5  -5.6  -6.8  -3.9  -6.6  -6.8  -4.6  -2.5  2.3  -3.3  -4.1  -5.0  -1.7  -6.1  0.5  0.5

> print(SFDM);
DayMatrix(Peptide, pam=250, Sim: max=16.847, min=-8.222, max offdiag=6.785,
del=-19.814-1.396*(k-1))
C  12.2
S  1.4  1.8
T  -0.4  1.5  2.6
P  -0.8  1.1  0.5  5.8
A  -0.2  1.2  1.3  1.3  1.7
G  -1.0  1.4  0.5  0.1  1.6  4.4
N  -1.8  0.9  0.5  -0.4  0.3  0.7  2.4
D  -3.3  0.2  -0.2  -1.0  0.3  0.8  2.2  4.1
E  -3.6  -0.2  -0.6  -0.7  0.2  0.3  1.3  3.5  4.3
Q  -3.4  -0.5  -0.8  0.3  -0.4  -0.8  0.7  1.5  2.5  4.2
H  -1.3  -0.3  -0.8  0.1  -0.7  -1.1  1.6  0.8  0.8  2.8  4.8
R  -2.1  -0.4  -1.0  -0.3  -1.6  -2.3  -0.2  -1.8  -1.7  1.2  1.6  6.7
K  -2.9  0.1  0.2  -0.7  -0.7  -0.8  1.1  0.2  -0.0  0.9  0.6  3.4  4.0
M  -4.2  -2.2  -1.2  -2.8  -1.7  -3.1  -2.5  -3.7  -3.2  -1.6  -2.4  -1.2  0.2  8.5
18.3. BETTER DAYHOFF MATRICES

\[
\begin{array}{cccccccccccc}
I & -1.2 & -2.0 & -0.3 & -2.7 & -1.0 & -2.8 & -2.5 & -3.3 & -3.0 & -2.9 & -2.7 & -2.9 & -2.3 & 1.3 & 5.5 \\
L & -4.5 & -3.1 & -2.1 & -2.8 & -2.1 & -3.9 & -3.3 & -4.7 & -4.1 & -2.1 & -2.0 & -3.7 & -2.9 & 3.2 & 1.7 & 5.8 \\
V & -0.4 & -1.0 & 0.1 & -1.2 & 0.1 & -1.0 & -1.9 & -2.5 & -2.3 & -2.1 & -2.0 & -3.0 & -2.3 & 1.1 & 3.5 & 1.4 & 4.1 \\
F & -2.6 & -3.3 & -3.4 & -4.6 & -3.6 & -4.5 & -3.7 & -6.1 & -6.1 & -4.9 & -1.6 & -5.0 & -5.1 & -0.7 & 0.4 & 1.4 & -1.7 & 9.0 \\
Y & 1.8 & -2.8 & -2.8 & -4.9 & -3.4 & -4.7 & -2.1 & -4.7 & -4.8 & -4.2 & 0.1 & -4.6 & -4.2 & -3.7 & -1.8 & -1.4 & -2.9 & 6.8 & 10.2 \\
W & -6.5 & -3.1 & -5.9 & -6.3 & -6.3 & -7.1 & -4.7 & -7.7 & -8.2 & -5.6 & -2.9 & 1.3 & -3.9 & -6.0 & -6.9 & -2.8 & -7.3 & -0.4 & -0.9 & 16.8 \\
\end{array}
\]

> OrigDM[MaxSim];    SPDM[MaxSim];
17.3021    16.8467
> OrigDM[MinSim];    SPDM[MinSim];
  -7.5098     -8.2217
> OrigDM[MaxOffDiag]; SPDM[MaxOffDiag];
 6.9511      6.7851
> OrigDM[FixedDel];  SPDM[FixedDel];
 -19.8137    -19.8137

Matchings performed via dynamic programming will apply penalties for deletions of length \( k \) according to \textbf{FixedDel} + (\( k - 1 \)) \textbf{IncDel}. This gap penalty implies that a gap of length \( k \) occurs with probability \( 0.00076 \cdot (0.7251)^k \). Chapter 19 – \textit{Insertions and Deletions} describes this scoring function in more depth.

18.3 Better Dayhoff Matrices

The Dayhoff matrix computed by Dayhoff et. al. [9] was based on an insufficient number of matched amino acid pairs to sustain an analysis of substitution rates any more sophisticated than that implied by the Markov model. Today, it is a relatively easy (and computationally feasible) task to gather on the order of millions of amino acid matchings. One would typically need to perform a “self-matching” of only one entire database (such as Swiss-
Proto) to gather a sufficient amount of data.\footnote{An exhaustive matching of the Swiss-Prot database is available at the Computational Biochemistry Research Group web site [5].}

The article \textit{Analysis of mutation during divergent evolution}, Gonnet, Cohen and Benner, (1992) [16] details the first exhaustive “self-matching” of the Swiss-Prot vers. 23 database. At that time, Swiss-Prot consisted of approximately 27,000 sequences. New Dayhoff matrices were formed from the frequency information contained in the alignments remaining after inspecting each by hand and removing suspect alignments (those thought not to be true or not due to point mutations, insertions or deletion events). Section 18.4 – \textit{Estimating Mutation Matrices} describes their methodology. This section explores the Darwin commands for building such matrices.

\subsection{The \texttt{CreateDayMatrices} Function}

For various purposes, including the estimation of PAM distances between two aligned sequences, it is necessary to use an array of Dayhoff matrices computed for a range of PAM distances. Chapter 21 – \textit{The Pairwise Comparison of Amino Acid Sequences} discusses this estimation in greater depth. The Darwin function \texttt{CreateDayMatrices} computes such a range of
18.3. BETTER DAYHOFF MATRICES

“enhanced” Dayhoff matrices.

Calling Sequences:

\texttt{CreateDayMatrices()}

Returns: \texttt{DayMatrix}

\textbf{Synopsis:} This function performs the following three calculations:

1. It assigns a Dayhoff matrix computed at PAM distance 250 to the system variable \texttt{DM}.

2. It computes 1266 Dayhoff matrices for various PAM distances between 0.049 and 1000 and assigns the \texttt{list} of such matrices to the system variable \texttt{DMS}.

The PAM distances for these matrices are not restricted to integers. The \texttt{CreateDayMatrices} function produces a large number of matrices at low PAM.

\begin{verbatim}
> CreateDayMatrices();
> for i from 1 to length(DMS) do
>   printf(' %5.5f', DMS[i][PamNumber]);
> od;
0.04945 0.05055 0.05167 0.05282 0.05399 0.05519 0.05642 0.05767 0.05896 0.06027 0.06161 0.06297 0.06437 0.06580 0.06727 0.06876 0.07029 0.07185 0.07345 0.07508 0.07675 0.07845 0.08020 0.08198 0.08380 0.08566 0.08757 0.08951 0.09150 0.09354 0.09561 0.09774 0.09991 0.10213 0.10440 0.10672 0.10909 0.11152 0.11400 0.11653 0.11912 0.12176 0.12447 0.12724 0.13006 0.13295 0.13591 0.13893 0.14202 0.14517 0.14840 0.15170 0.15507 0.15851 0.16204 0.16564 0.16932 0.17308 0.17693 0.18086 0.18488 0.18899 0.19319
\end{verbatim}
Comparing the 250-PAM original Dayhoff matrix with the 250-PAM “enhanced” Dayhoff matrix reveals that, although similar, the matrices have significant differences.

```plaintext
> OrigTot := [87, 41, 40, 47, 33, 38, 50, 89, 34, 37, 85, 81, 15, 40, 51, 70, 58, 10, 30, 65];
> OrigFreq := OrigTot/sum(OrigTot);
> OrigDM := CreateOrigDayMatrix(Mutations1978, OrigFreq, 250);

> print(OrigDM);
DayMatrix(Peptide, pam=250, Sim: max=17.302, min=-7.510, max offdiag=6.961, del=-19.814-1.396*(k-1))
C 12.0
S -0.0  1.6
T -2.2  1.3  2.6
P -2.7  0.9  0.3  5.9
A -2.0  1.1  1.2  1.1  1.8
G -3.3  1.1 -0.0 -0.5  1.3  4.8
N -3.6  0.7  0.4 -0.5  0.2  0.4  2.0
D -5.1  0.3 -0.1 -1.0  0.3  0.6  2.1  3.9
E -5.3 -0.0 -0.4 -0.6  0.3  0.2  1.4  3.4  3.9
Q -5.3 -0.5 -0.8  0.2 -0.4 -1.2  0.8  1.6  2.5  4.1
H -3.4 -0.8 -1.3 -0.3 -1.4 -2.1  1.6  0.7  0.6  2.9  6.6
R -3.6 -0.3 -0.9 -0.2 -1.6 -2.6 -0.0 -1.3 -1.1  1.2  1.5  6.1
K -5.4 -0.2 -0.0 -1.2 -1.2 -1.7  1.0  0.1 -0.1  0.7 -0.1  3.4  4.7
M -5.2 -1.6 -0.6 -2.1 -1.2 -2.8 -1.8 -2.6 -2.2 -1.0 -2.2 -0.5  0.4  6.6
I -2.3 -1.4  0.1 -2.0 -0.5 -2.6 -1.8 -2.4 -2.0 -2.0 -2.5 -2.0 -1.9  2.2  4.6
L -6.0 -2.8 -1.7 -2.6 -1.9 -4.0 -2.9 -4.0 -3.3 -1.8 -2.1 -3.0 -2.9  3.7  2.4  6.0
V -1.9 -1.0  0.3 -1.2  0.2 -1.4 -1.8 -2.2 -1.8 -1.9 -2.3 -2.5 -2.5  1.8  3.7  1.8  4.3
F -4.3 -5.2 -3.1 -4.6 -3.5 -4.8 -3.5 -5.6 -5.4 -4.7 -1.8 -4.5 -5.3  0.2  1.0  1.8 -1.2  9.1
Y  0.4 -2.8 -2.8 -5.0 -3.5 -5.3 -2.1 -4.3 -4.3 -4.0 -0.1 -4.2 -4.5 -2.5 -1.0 -0.9 -2.5  7.0 10.
W -7.5 -2.3 -5.0 -5.5 -5.6 -6.8 -3.9 -6.6 -6.8 -4.6 -2.5  2.3 -3.3
```
18.3. BETTER DAYHOFF MATRICES

\[-4.1 \quad -5.0 \quad -1.7 \quad -6.1 \quad 0.5 \quad 0.0 \quad 17.3\]

> print(DM);
DayMatrix(Peptide, pam=250, Sim: max=14.152, min=-5.161, max offdiag=5.080,
  del=-19.814-1.396*(k-1))

C 11.5
S 0.1 2.2
T -0.5 1.5 2.5
P -3.1 0.4 0.1 7.6
A 0.5 1.1 0.6 0.3 2.4
G -2.0 0.4 -1.1 -1.6 0.5 6.6
N -1.8 0.9 0.5 -0.9 -0.3 0.4 3.8
D -3.2 0.5 -0.0 -0.7 -0.3 0.1 2.2 4.7
E -3.0 0.2 -0.1 -0.5 -0.0 -0.8 0.9 2.7 3.6
Q -2.4 0.2 0.0 -0.2 -0.2 -1.0 0.7 0.9 1.7 2.7
H -1.3 -0.2 -0.3 -1.1 -0.8 -1.4 1.2 0.4 0.4 1.2 6.0
R -2.2 -0.2 -0.2 -0.9 -0.6 -1.0 0.3 -0.3 0.4 1.5 0.6 4.7
K -2.8 0.1 0.1 -0.6 -0.4 -1.1 0.8 0.5 1.2 1.5 0.6 2.7 3.2
M -0.9 -1.4 -0.6 -2.4 -0.7 -3.5 -2.2 -3.0 -2.0 -1.0 -1.3 -1.7 -1.4 4.3
I -1.1 -1.8 -0.6 -2.6 -0.8 -4.5 -2.8 -3.8 -2.7 -1.9 -2.2 -2.4 -2.1 2.5 4.0
L -1.5 -2.1 -1.3 -2.3 -1.2 -4.4 -3.0 -4.0 -2.8 -1.6 -1.9 -2.2 -2.1 2.8 2.8 4.0
V -0.0 -1.0 0.0 -1.8 0.1 -3.3 -2.2 -2.9 -1.9 -1.5 -2.0 -2.0 -1.7 1.6 3.1 1.8 3.4
F -0.8 -2.8 -2.2 -3.8 -2.3 -5.2 -3.1 -4.5 -3.9 -2.6 -0.1 -3.2 -3.3 1.6 1.0 2.0 0.1 7.0
Y -0.5 -1.9 -1.9 -3.1 -2.2 -4.0 -1.4 -2.8 -2.7 -1.7 2.2 -1.8 -2.1 -0.2 -0.7 -0.0 -1.1 5.1 7.8
W -1.0 -3.3 -3.5 -5.0 -3.6 -4.0 -3.6 -5.2 -4.3 -2.7 -0.8 -1.6 -3.5 -1.0 -1.8 -0.7 -2.6 3.6 4.1 14.2

To extract an entry of DMS with a particular PAM distance, we use the function.

`SearchDayMatrix(p : real, D : array(DayMatrix))`

where `p` is the target PAM distance and `D` is the array of Dayhoff matrices.

> print(SearchDayMatrix(250, DMS));
DayMatrix(Peptide, pam=250, Sim: max=14.152, min=-5.161,
  max offdiag=5.080, del=-19.814-1.396*(k-1))

...
202 CHAPTER 18.  POINT ACCEPTED MUTATIONS AND DAYHOFF MATRICES

It is equivalent to the matrix assigned to variable DM.

The CreateDayMatrices function must be called before you attempt to perform a match between two sequences. Unless the memory requirements of your system are extremely limited, it is best to get into the habit of always calling this function when you begin your new session.

It is interesting to examine how these matrices change from low PAM distance where the matrix is positive on the diagonal and very negative in the off-diagonal to high PAM distance where the entries smooth out.

> print(DMS[1]);

DayMatrix(Peptide, pam=0.0494497, Sim: max=18.839, min=-50.808,
max offdiag=-24.377, del=-47.348-1.396*(k-1))

C 17.2
S -31.5 12.2
T -34.6-25.7 12.1
F -46.5-31.6-32.5 13.4
A -31.1-27.3-30.6-31.8 11.1
G -38.3-31.8-38.4-37.9-31.3 11.3
N -37.2-28.6-30.6-37.1-35.4-32.2 13.4
D -45.3-31.8-33.1-35.8-34.3-33.5-27.0 12.7
E -48.7-32.4-33.8-34.7-31.7-36.8-32.6-25.8 12.3
Q -41.8-31.4-32.0-32.7-32.4-35.9-30.4-31.8-26.2 14.2
H -35.2-33.3-32.8-35.8-35.2-37.1-28.4-32.5-32.4-27.3 16.3
R -37.3-33.5-33.7-36.2-34.8-35.5-33.4-37.5-34.1-28.2-31.3 12.7
K -46.6-32.6-31.4-34.8-33.9-37.0-29.7-33.5-29.0-27.2-31.6-25.6 12.3
M -34.6-34.7-32.3-44.2-32.4-40.7-38.4-45.9-36.7-30.8-34.2-37.3 16.4
I -38.6-39.3-33.4-40.6-38.1-47.8-38.6-46.9-40.3-38.9-38.4-40.0-37.3 12.4
L -37.6-38.8-37.0-36.9-35.4-44.1-40.8-48.5-40.5-34.0-37.5-37.1-37.4-25.9-27.3 10.3
V -32.4-37.4-30.1-37.7-29.7-42.6-40.3-47.1-35.7-37.0-39.6-37.8-37.2-30.6-24.4-30.1 11.6
F -35.0-41.1-37.8-42.6-38.2-45.3-39.5-46.2-45.6-39.4-33.6-44.2-43.4-29.5-32.5-29.3-34.8 13.9
Y -34.1-35.0-37.8-39.7-38.8-42.7-34.6-38.9-40.2-37.2-27.2-36.1-37.7-35.7-36.8-35.6-36.0-24.7 14.1
W -35.5-38.4-41.7-45.2-41.6-39.6-41.4-50.8-42.6-37.1-35.9-34.5-43.7-35.9-38.4-35.9-41.7-28.9-28.2
18.3. BETTER DAYHOFF MATRICES

18.8

C S T P A G N D E Q H R K M
I L V F Y W

\> print(DMS[length(DMS)]);

DayMatrix(Peptide, pam=1000, Sim: max=3.184, min=-0.454, max offdiag=0.882, 
del=-15.338-1.396*(k-1))

C 0.9
S -0.0 0.0
T -0.0 0.0 0.0
P -0.1 0.1 0.0 0.3
A 0.0 0.0 0.0 0.0 0.0
G -0.1 0.1 0.0 0.1 0.1 0.5
N -0.1 0.1 0.0 0.1 0.0 0.1 0.1
D -0.1 0.1 0.0 0.1 0.0 0.2 0.1 0.2
E -0.1 0.1 0.0 0.1 0.0 0.1 0.1 0.1 0.1
Q -0.1 0.0 0.0 0.0 0.0 0.1 0.1 0.1 0.1 0.1
H -0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
R -0.1 0.0 0.0 0.0 0.0 0.1 0.1 0.1 0.1 0.1 0.1
K -0.1 0.0 0.0 0.1 0.0 0.1 0.1 0.1 0.1 0.0 0.1 0.1
M 0.0 -0.1 -0.0 -0.1 -0.0 -0.2 -0.1 -0.2 -0.1 -0.1 -0.0 -0.1 -0.1 0.2
I 0.0 -0.1 -0.0 -0.1 -0.0 -0.3 -0.1 -0.2 -0.1 -0.1 -0.1 -0.1 -0.1 0.2 0.2
L 0.0 -0.1 -0.0 -0.1 -0.1 -0.3 -0.1 -0.2 -0.2 -0.1 -0.1 -0.1 -0.1 0.2 0.3 0.3
V 0.1 -0.1 -0.0 -0.1 -0.0 -0.2 -0.1 -0.1 -0.1 -0.1 -0.0 -0.1 -0.1 0.2 0.2 0.1
F 0.1 -0.2 -0.1 -0.2 -0.1 -0.4 -0.2 -0.2 -0.2 -0.1 -0.2 -0.2 0.2 0.2 0.2 0.6
Y 0.1 -0.1 -0.1 -0.2 -0.1 -0.3 -0.1 -0.2 -0.2 -0.1 0.0 -0.1 -0.1 0.2 0.2 0.1 0.5 0.5
W 0.1 -0.2 -0.2 -0.4 -0.2 -0.5 -0.3 -0.4 -0.3 -0.2 0.0 -0.2 -0.3 0.2 0.2 0.2 0.1 0.9 0.9 3.2

18.3.2 The CreateDayMatrix Function

We can ask Darwin to produce a Dayhoff matrix for a specific PAM distance or specific range of PAM distances via the CreateDayMatrix function and
the global variable logPAM1.

**Calling Sequences:**

`CreateDayMatrix(logmat, pam)`

`CreateDayMatrix(logmat, r)`

**Parameters:**

- `logmat` : array(real, real)
- `pam` : real > 0
- `r` : range, \( 0 < r_1 \leq r_2 \)

**Returns:** DayMatrix or array(DayMatrix)

**Synopsis:** This function computes a similarity matrix (enhanced Dayhoff) from the logarithmic mutation matrix \( \logPAM1 \) and a specified PAM value or range of PAM values.

These matrices are equivalent to those produced by the `CreateDayMatrices` function.

```plaintext
> p55 := CreateDayMatrix(logPAM1, 55);
```

The variable \( \logPAM1 \) deserves some attention. The entries of this matrix are the logarithm of a 1-PAM mutation matrix. It is particularly economical to compute \( k \)-PAM mutation matrices from this transformed matrix as we need only compute

\[
M^k = e^{k \logPAM1}
\]

We use this to compute a 55-PAM similarity matrix.

```plaintext
> PrintMatrix( 10000*logPAM1, '%4d' );
```

```
-110  5  5   6   12   9   11   5   2   5   6   9   2   10   30   14   1
  4  -93  5   2   2   16   4   3   8   1   2   30   2   0   3   5   5   4
  3  4-112 18   2   8   5   6  13   1   1   10   1   1   2   13   8   1
  4   2   22  -95   0   7  28   5   6   0   0   5   0   0   3   7   5   0
  3   1   1   0  -55   0   0   1   1   1   1   0   1   1   0   3   1   1
  4  11   7   5   1-145  18   2  14   1   3   15   6   1   4   5   5   1
```
18.3. BETTER DAYHOF MATRICES

\[\begin{array}{cccccccccccccccc}
8 & 5 & 6 & 31 & 0 & 28 & -111 & 2 & 7 & 1 & 1 & 15 & 3 & 0 & 4 & 7 & 5 & 1 \\
11 & 4 & 9 & 7 & 2 & 4 & 3 & -48 & 3 & 0 & 1 & 3 & 1 & 0 & 2 & 10 & 2 & 2 \\
1 & 4 & 7 & 3 & 1 & 9 & 3 & 1 & -106 & 1 & 1 & 3 & 2 & 2 & 1 & 2 & 3 & 1 \\
2 & 1 & 2 & 0 & 2 & 2 & 1 & 0 & 2 & -122 & 22 & 2 & 27 & 7 & 1 & 1 & 5 & 2 \\
6 & 4 & 2 & 0 & 3 & 8 & 2 & 1 & 3 & 36 & -82 & 3 & 49 & 22 & 4 & 3 & 4 & 5 \\
5 & 33 & 13 & 5 & 0 & 23 & 15 & 2 & 8 & 2 & 2 & -118 & 5 & 1 & 4 & 6 & 9 & 1 \\
3 & 1 & 1 & 0 & 2 & 4 & 1 & 0 & 2 & 10 & 12 & 2 & -142 & 5 & 0 & 2 & 3 & 1 \\
1 & 0 & 1 & 0 & 3 & 1 & 0 & 0 & 4 & 5 & 10 & 0 & 9 & -78 & 0 & 1 & 1 & 10 \\
6 & 2 & 2 & 2 & 0 & 5 & 3 & 1 & 2 & 1 & 2 & 3 & 0 & 1 & -58 & 6 & 5 & 0 \\
23 & 5 & 17 & 8 & 9 & 9 & 7 & 8 & 6 & 1 & 2 & 7 & 4 & 1 & 8 & -139 & 33 & 2 \\
11 & 5 & 11 & 6 & 4 & 8 & 5 & 2 & 7 & 6 & 2 & 9 & 7 & 2 & 7 & 33 & -122 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 3 & 0 & 0 & 0 & -44 \\
1 & 2 & 2 & 1 & 3 & 1 & 1 & 0 & 13 & 1 & 2 & 1 & 2 & 2 & 1 & 1 & 2 & 2 \\
15 & 2 & 1 & 0 & 8 & 3 & 4 & 1 & 2 & 51 & 14 & 3 & 12 & 5 & 2 & 3 & 14 & 1 \\
\end{array}\]

\[\text{\texttt{> M55 := exp( 55*logPAM1 );}}\]

\[\text{\texttt{PrintMatrix( 1000*M55, '\%4d' );}}\]

\[
\begin{array}{cccccccccccccccc}
19 & 13 & 74 & 606 & 4 & 33 & 94 & 22 & 27 & 3 & 2 & 24 & 4 & 2 & 15 & 30 & 24 & 2 \\
12 & 3 & 4 & 1 & 742 & 2 & 1 & 3 & 5 & 4 & 4 & 1 & 6 & 6 & 1 & 11 & 6 & 5 \\
18 & 40 & 26 & 23 & 4 & 463 & 56 & 10 & 45 & 6 & 12 & 47 & 20 & 5 & 17 & 21 & 19 & 7 \\
45 & 20 & 38 & 30 & 13 & 19 & 17 & 772 & 15 & 3 & 4 & 16 & 7 & 3 & 14 & 40 & 15 & 8 \\
7 & 15 & 24 & 12 & 6 & 28 & 12 & 5 & 564 & 4 & 4 & 14 & 7 & 10 & 6 & 10 & 10 & 6 \\
14 & 8 & 8 & 3 & 11 & 10 & 7 & 2 & 9 & 538 & 80 & 11 & 88 & 31 & 7 & 9 & 23 & 9 \\
26 & 18 & 10 & 4 & 18 & 30 & 11 & 5 & 18 & 130 & 658 & 18 & 162 & 88 & 18 & 15 & 21 & 25 \\
10 & 5 & 4 & 2 & 7 & 12 & 5 & 2 & 7 & 34 & 39 & 8 & 466 & 20 & 2 & 7 & 10 & 6 \\
7 & 3 & 5 & 2 & 12 & 6 & 2 & 2 & 17 & 22 & 38 & 3 & 35 & 663 & 3 & 5 & 7 & 46 \\
25 & 12 & 10 & 12 & 3 & 20 & 15 & 8 & 12 & 5 & 9 & 14 & 5 & 3 & 731 & 26 & 22 & 2 \\
72 & 24 & 57 & 34 & 34 & 34 & 30 & 33 & 26 & 10 & 9 & 29 & 18 & 7 & 34 & 466 & 96 & 8 \\
1 & 4 & 1 & 0 & 4 & 2 & 1 & 1 & 4 & 2 & 3 & 1 & 3 & 15 & 1 & 2 & 1 & 785 \\
\end{array}\]
CHAPTER 18. POINT ACCEPTED MUTATIONS AND DAYHOFF MATRICES

|    | 8  | 10 | 5  | 12 | 7  | 4  | 2  | 45 | 8  | 10 | 6  | 10 | 83 | 4  | 9  | 6  | 43 | 51 | 12 | 9  | 5  | 33 | 15 | 16 | 6  | 10 | 156 | 58 | 14 | 54 | 24 | 13 | 18 | 47 | 7  |
|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|

18.3.3 Percent Identity and PAM Distance

Recall, a 1-PAM mutation matrix measures a unit of evolution where the expected change is exactly 1% of the amino acids. It is clear that a 2-PAM evolution will not produce a 2% change since some changes may mutate back to the original amino acid. (A 2-PAM evolution is equivalent to two separate 1-PAM evolutionary units.) The percentage identity, \( p_i \), is defined to be the number of identical bases between two sequences in an alignment. It is related to the PAM distance, \( p \), by

\[
pi(p) = \sum_{i=1}^{20} f_i (M^p)_{ii}
\]

In the above formula we need both the mutation matrix and the frequency vector. The frequency vector can be derived from any mutation matrix. Recall the symmetry relation:

\[
f_i M_{ji} = f_j M_{ij}
\]

Setting \( j = 1 \) and rearranging

\[
f_i = \frac{f_i M_{i1}}{M_{i1}}
\]

On the other hand we know that \( \sum_{i=1}^{20} f_i = 1 \), consequently

\[
f_i = \frac{M_{i1}}{M_{i1} \sum_{j=1}^{20} \frac{M_{1j}}{M_{j1}}}
\]

The formula for \( pi(p) \) can be made independent of the frequencies \( f_i \) as:

\[
pi(p) = \frac{\sum_{i=1}^{20} \frac{M_{i1}}{M_{i1}} (M^p)_{ii}}{\sum_{i=1}^{20} \frac{M_{i1}}{M_{i1}}}
\]
18.3. BETTER DAYHOFF MATRICES

for any mutation matrix $M$, in particular for $M^p$ which is also a mutation
matrix.

The following function computes the percent identity for a given PAM
distance.

```plaintext
> PamToPerIdent := proc( pam:real, logPAM1:array(real,20,20) )
> description
> 'compute the percentage identity that a pam distance will leave';
> if pam < 0 then error('invalid percentage range') fi;
> num := den := 0;
> M := exp(pam*logPAM1);
> for i to 20 do
> num := num + M[i,1]*M[i,i]/M[1,i];
> den := den + M[i,1]/M[1,i]
> od;
> 100*num/den;
> end:
```

By definition, the following should be 99% 

```plaintext
> PamToPerIdent(1,logPAM1);
99.0000
```

```plaintext
> PamToPerIdent(250,logPAM1);
16.9051
```

the classical PAM value gives about 17% identity. Many believe that ap-
proximately 20% identity is needed to detect homology based on sequence
data alone.

Computing the inverse function, PerIdentToPam, is somewhat more di-
ficult. Since it is very unlikely that a closed form solution exists, we use
Newton’s method for solving equations.

Newton’s method begins with an initial guess $x_0$ and computes $x_1, x_2, x_2, \ldots$
using
\[ x_{i+1} = x_i - \frac{g(x_i)}{g_x'(x_i)} \]
where \( g(x) \) is the equation to be solved in the unknown \( x \) and \( g_x'(x) \) is the derivative of \( g(x) \) with respect to \( x \). In our case, the equation is

\[ g(p) = \sum_{i=1}^{20} f_i (M^p)_{ii} - p \dot{v} = 0 \]

and our unknown is \( p \), the PAM distance. Computing the derivative of \( M^p \) with respect to \( p \) is not so complicated. Recall that \( M \) is a square matrix so \( \ln(M) \) is also a square matrix and using the series definition we can compute

\[
(M^p)'_p = (e^{p\ln(M)})'_p = (I + p\ln(M) + \frac{p^2\ln(M)^2}{2} + \ldots)'_p
\]

\[ = \ln(M) + p\ln(M)^2 + \frac{p^2\ln(M)^3}{2} + \ldots \]

\[ = \ln(M) e^{p\ln(M)} = \ln(M) M^p \]

\[ = e^{p\ln(M)} \ln(M) = M^p \ln(M) \]

(\( M \) and \( \ln(M) \) necessarily commute under matrix multiplication). As expected the derivative of powers of matrices has the same expression as for scalars.

Consequently the derivative of the equation is

\[ g_x'(p) = \sum_{i=1}^{20} f_i (\ln(M) M^p)_{ii} \]

This exercise also shows how to protect a Newton iteration from going astray by using binary search techniques. Newton's method may or may not converge on a solution. When it does, it does so quickly. However, often it does not converge at all.

> PerIdentToPam := proc( pi:real, logPAM1:array(real,20,20) )
> description 'compute the PAM distance which results in the given
> percentage identity';
check the argument range and trivial case

\[ \text{if } pi \leq 0 \text{ or } pi > 100 \text{ then error('invalid percentage range')} \text{ fi;} \]
\[ \text{if } pi=100 \text{ then return(0) fi;} \]

Compute the amino acid frequencies as described before

\[ AF := \text{CreateArray(1..20)}; \]
\[ M := \text{exp(logPAM1);} \]
\[ \text{for } i \text{ to } 20 \text{ do } AF[i] := M[i,1]/M[1,i] \text{ od;} \]
\[ AF := AF/\text{sum(AF)}; \]

Now check that the percentage identity is not less than its asymptotic value.
That is, for \( PAM \to \infty \), the percentage identity does not decrease to zero
but to the percentage identity of two random sequences. This value is

\[ \lim_{p \to \infty} pi(p) = \sum_{i=1}^{20} f_i^2 \]

\[ \text{asy := AF*AF;} \]
\[ \text{if } pi/100 \leq \text{asy then} \]
\[ \text{error('pi cannot be less than the asymptotic value',100*asy) fi;} \]

For reasons which escape the scope of this tutorial, the following is a good
initial guess for the PAM value.

\[ \text{pam := -100*ln(pi/100-asy);} \]

Next, the PAM value will be bound below by \( lo \) and above by \( hi \). Setting
the above bound to 5 times the initial guess is sufficient.

\[ lo := 0; \text{ hi := 5*pam;} \]

We will now iterate until we reach the desired accuracy, i.e. the difference
between the bounds is close to the machine epsilon.

\[ \text{while } hi-lo > hi*DBL_EPSILON*10 \text{ do} \]
\[ \text{mp := exp(pam*logPAM1);} \]
mip := logPAM1*mp;
num := -pi/100;
den := 0;
for i to 20 do
  num := num + AF[i]*mp[i,i];
den := den + AF[i]*m1p[i,i]
od;

Reset lo or hi as appropriate

if num >= 0 then lo := pam else hi := pam fi;
incr := -num/den;
pam := pam + incr;

if abs(incr)^2 < abs(pam)*DBL_EPSILON then break fi;

If the value falls outside the bounds, then the Newton’s value is not converging, so we just compute the midpoint of the bounds as the next guess.

if pam <= lo or pam >= hi then pam := (lo+hi)/2 fi;
od;
pam
end:

Again, by definition, the following should evaluate to 1.

PerIdentToPam(99,logPAM1);
1.0000

A 15% identity corresponds to PAM 278.5236.

PerIdentToPam(15,logPAM1);
278.5236
18.4 Estimating Mutation Matrices

This section describes a method for computing mutation matrices, and hence Dayhoff matrices, which is more accurate and based on better theoretical grounds. This method will derive the information from a sample of matches of sequences (see “Analysis of Mutation During Divergent Evolution” M. Cohen, S. Benner and G. Gonnet, [8]). We will assume that we have a large enough sample which has been inspected and approved by an expert in the area. Although we could construct sample sets automatically, it is important that a person analyses this sample and weeds out unsuitable sequences which do not represent mutations by evolution.

We will further require that this sample be selected so that all its matches have approximately the same PAM distance. For example, let us suppose that all the matches have been selected so that their distance is between 35 and 45 PAM units. Chapter 21 – *The Pairwise Comparison of Amino Acid Sequences* explains how to compute PAM distances of matches.

Suppose that we run the following experiment:

i) Generate a random sequence $S_0$.

ii) Randomly mutate $S_0$ with a 40-PAM mutation matrix generating a new sequence $S_1$.

iii) Compare $S_0$ and $S_1$ and build a comparison matrix $C$, such that every time that we have an amino acid $i$ in $S_0$ mutating into an amino acid $j$ in $S_1$ we add 1 to $C_{ij}$ (and if it does not mutate, i.e. stays the same, we add 1 to $C_{ii}$).

It is easy to see that the expected value of $C$ is given by

$$C \approx M^{40} \times N$$

where $N$ is a diagonal matrix and $N_{ii}$ is the number of times amino acid $i$ appears in $S_0$. This approximation, by the law of large numbers, will be
more accurate as we increase the number of sample points (we increase the length of \( S_0 \) and hence increase \( N \)).

The above equation is crucial for the estimation of \( M \) when we can tabulate a matrix \( C \). Doing some trivial algebra, we find that

\[
M \approx (C \times N^{-1})^{1/40}
\]

Since we do not know from our data which sequence evolved we will add 1/2 to \( C_{ij} \) and 1/2 to \( C_{ji} \) instead of adding 1 to \( C_{ij} \).

The matrix \( N \) is obtained from counting all the amino acids analysed. Multiplying \( C \) by \( N^{-1} \) is equivalent to dividing each column of \( C \) by the sum of the column. (Note the columns of a mutation matrix add up to 1.)

Now, if we were certain about the PAM distance of the sample, in this case 40, we could compute \((C \times N^{-1})^{1/40}\). However, we may not have a lot of trust in the estimated PAM distances. This does not pose a problem since we can use the definition of a 1-PAM matrix as an “anchor”:

\[
\sum_i f_i (1 - M_{ii}) = 0.01
\]

The problem reduces to finding \( \alpha \) such that \((C \times N^{-1})^{1/\alpha}\) is a 1-PAM matrix. This value \( \alpha \) will be an estimate of the average PAM distance of the sample.

The following Darwin function computes an estimate of a 1-PAM matrix based on a sample of matches stored in a file. This function also receives as parameters a minimum value for the similarity, a minimum value for the length of the match and the minimum and maximum value for the PAM distance that we want to select.

```plaintext
> EstMutMat := proc( filename:string, MinSim:real, MinLength:real,
>                     Min{PAM}:real, Max{PAM}:real )
>                      description 'estimate a mutation matrix from matches read from a file';
>     C := CreateArray(1..20,1..20);
>     M := CreateArray(1..20,1..20);
>
>     # Print some appropriate message to describe the present run.
```

```plaintext
> ```
To read a file from Darwin, line by line we first must establish a pipe via the OpenPipe command. This changes the standard input from the keyboard to the specified file. The file can be read line by line by issuing ReadLine calls within the body of a loop.

> OpenPipe('cat filename');

Initialize various counters to zero.

> totm := totma := totlm := toteq := totmut := 0;

This is now the main reading loop. The loop will be exited when the ReadLine call finds an end-of-file in which case returns the string EOF.

> do m := ReadLine();
>     if m=EOF then break fi;

It is worthwhile to check that we read a match from the file.

>     if not type(m,Match) then error('invalid input in', filename) fi;
>     totm := totm+1;

If the values of the match do not fall within the acceptable ranges, then skip this match.

>     if m[Sim] < MinSim or max(m[Length1],m[Length2]) < MinLength or
>         m[PamNumber] > MaxPam or m[PamNumber] < MinPam then next fi;
>     totma := totma+1;

The function DynProgStrings prepares an alignment for printing. In this case, we are interested in the strings of the aligned sequences as they would be printed. By scanning these two strings we can count the identities and the mutations.
> sm := DynProgStrings( m, SearchDayMatrix(m[PAmNumber],DMS) );
> lm := length(sm[2]);
> totlm := totlm + lm;

For each character in the matched strings add in the C matrix.

> for i to lm do
>     i1 := AToInt(sm[2,i]); i2 := AToInt(sm[3,i]);
>     if i1>0 and i1<21 and i2>0 and i2<21 then
>         if i1=i2 then toteq := toteq+1;
>         C[i1,i1] := C[i1,i1]+1
>     else totmu := totmu+1;
>     C[i1,i2] := C[i1,i2]+1/2;
>     C[i2,i1] := C[i2,i1]+1/2
>     fi
> end;

Print the collected counts.

> printf(‘%d matches read, %d selected with a total of %d positions,’,
>         totm, totma, totlm );
> printf(‘%d exact matches, %d mutations and %d deletions

> toteq, totmu, totlm-toteq-totmu );

Compute the sum of the columns, but since the matrix C is symmetric, it is easier to compute the sum of the rows.

> Cols := CreateArray(1..20);
> for i to 20 do Cols[i] := sum(C[i]) od;

Now compute the frequencies of amino acids into F and normalize C into M.

> tot := sum(Cols);
> F := Cols/tot;
> for i to 20 do for j to 20 do M[i,j] := C[i,j]/Cols[j] od od:
To compute the 1-PAM matrix we need to find the exponent $1/\alpha$ that will make $M^{1/\alpha}$ a 1-PAM matrix. The following iteration converges quite rapidly to the desired solution. It is based on estimating $1/\alpha$ as if it were linear on the probability of no change. This is not true, but both values are highly correlated. Hence by estimating and further correcting we converge to the right exponent.

> lnM := ln(M);
> alpha := 1;

Now iterate until a break is executed.

> do nochange := 0;
> Compute the probability of no change and if it is sufficiently close to 1%, exit the loop.
> for i to 20 do nochange := nochange + F[i]*(1-M[i,i]) od;
> if abs(nochange-0.01) < DBL_EPSILON then break fi;

If it is not accurate enough, it must be corrected.

> corr := 0.01/nochange;
> alpha := alpha/corr;
> lnM := lnM*corr;
> M := exp(lnM);
> od;
> printf('The PAM of the aggregated sample was %5g\n\n',alpha);

Return the logarithm of a 1-PAM matrix and the frequency vector as a list with two elements in it.

> [lnM,F]
> end:

Now run this function with the file SamplePAM40.

> res := EstMutMat( SamplePAM40, 80, 100, 35, 45 ):
Sample matches from file SamplePAM40, MinSimil=80, MinLength=100
PAM bounded between 35 and 45
838 matches read, 838 selected with a total of 244069 positions,
170528 exact matches, 65473 mutations and 8068 deletions

The PAM of the aggregated sample was 33.8193

The PAM estimate of the sample is slightly lower than expected. This
indicates that the original Dayhoff matrix tends to overestimate PAM dis-
tances.

Now we will compute a new Dayhoff matrix, with PAM 250, so that we
may compare it with the original one. Since we have the logarithm of a
1-PAM mutation matrix in res[1], we can used the CreateDayMatrix com-
mand:

> NewD := CreateDayMatrix(res[1], 250):
> print( NewD );
DayMatrix( Peptide, pam=250, Simil: max=15.035, min=-6.237,
max offdiag=5.554, del=-19.814-1.396*(k-1) )
C 12.6
S  0.3  2.0
T -1.1  1.3  2.4
P -3.1  0.9  0.1  6.8
A -0.7  1.2  1.1  0.8  2.4
G -2.1  0.6 -0.6 -1.4  0.8  5.9
N -1.7  1.0  0.5 -1.1  0.0  0.4  3.2
D -3.6  0.1 -0.4 -1.6 -0.1  0.6  2.2  4.8
E -4.2 -0.3 -0.5 -1.1 -0.1  0.1  1.0  3.4  4.2
Q -3.1 -0.4 -0.4 -0.1 -0.7 -1.2  0.5  0.6  1.7  3.9
H -1.3 -0.4 -0.9 -0.7 -1.3 -1.7  1.4  0.1 -0.2  2.2  6.1
R -2.1 -0.4 -0.5 -0.9 -1.2 -0.9  0.2 -1.1 -0.2  2.0  1.5  5.1
K -3.5 -0.3 -0.1 -1.1 -0.8 -1.1  0.9  0.1  0.9  2.0  0.7  3.5  4.2
M -1.9 -1.5 -0.1 -2.0 -0.9 -3.4 -2.3 -3.2 -2.6 -1.5 -2.3 -2.0 -1.7  4.8
I -1.8 -1.4  0.1 -2.1 -0.5 -3.6 -2.4 -3.5 -3.0 -2.5 -2.7 -2.9 -2.7  3.0
L -2.3 -2.0 -1.0 -1.6 -1.6 -4.4 -3.1 -4.3 -3.6 -1.9 -1.9 -2.5 -2.8  3.2
18.5. Other Similarity Matrices

Although similar, the new matrix has some significant differences with the original one.

18.5 Other Similarity Matrices

Darwin has several other matrices available to interested users. These can be loaded with the function `Matrices`.

```r
> Matrices()
```

**BLOSUM50, 60, 62, 70.** Sequence similarity among protein sequences is typically measured by similarity matrices, such as the Dayhoff PAM matrices based on evolutionary rates, which give scores for all possible exchanges of one amino acid with another. Henikoff and Henikoff have derived substitution matrices using about 2000 blocks of aligned subsequences from 504 groups of related proteins. In FASTA and BLAST searches, these matrices have shown improvements in alignments over the other substitution matrices, using query sequences from each of the 504 groups [19].

**PAM0, 50, 100, 150, 200, 250.** Various “enhanced” Dayhoff matrices computed for different PAM values. See §18.3.

**Dayhoff1978.** The original mutation matrix given in by Dayhoff et. al. [7].

**GENETIC10, 50, 100, 150, 200, 250.** These matrices are derived directly from the genetic code. They assume that the only source of mutations are random mutations in any of the three bases [15].

**Grantham.** Grantham’s structural dissimilarity matrix.

**Machlaclan.**
PIMA.

RDDH250. See the paper *Amino acid substitutions in structurally related proteins: a pattern recognition approach*, [24].

UNITARY, UNITARY2.
Chapter 19

Gap Penalties
Mutations occur at the DNA level and through the mechanism of gene expression these changes may manifest themselves in the protein encoded by the gene. The commonly used model for mutations at the DNA level partitions events into three categories:

1. Point accepted mutations. Base \( i \) mutates to base \( j \).

2. Insertions. A subsequence of DNA is inserted into a sequence.

3. Deletions. A subsequence of DNA is deleted from a sequence.

In the simplest theoretical model, all sequence mutations are explained as deletions and insertions of genetic material. A point accepted mutation corresponds to a deletion of a base followed by the insertion of a base at the same position in the sequence. Gene shuffling can be viewed as a series of deletion and insertion sequences. However, such a simplistic model does not take into account the relative probabilities of such events occurring. Intuitively, the probability of a single point mutation seems as though it would be much higher than the probability of an deletion event followed by an insertion event at the same position in the sequence. For this reason, we distinguish between point mutations and insertion/deletion events (\textit{indels}).

As is the case with our model for point mutations, our model for deletions and insertions is entirely symmetric. Given an alignment of two sequences \( A \), \( B \) containing at least one insertion or deletion event such as the following:

\[
\begin{align*}
\text{Sequence } A : & \quad \text{ALAEGLGVIACIGEKLDREAGITEKVVFEQTKVIADNVKD} \\
\text{Sequence } B : & \quad \text{CKNLGLETIVCTNN________________INTSKAVAALSPDY}
\end{align*}
\]

we can not determine whether sequence \( A \) has undergone an insertion or whether sequence \( B \) has undergone a deletion unless we know the ancestor of \( A \) and \( B \). Because we can not resolve this issue without an assumed origin, we treat both events in the same fashion.
19.1 Gap Probabilities and Penalties

Our algorithms for aligning amino acid sequences are based on the test of whether the two sequences evolved from a common ancestral sequence or not. This model allowed us to derive scores (used by the dynamic programming algorithm) from the mutation probabilities. In this section, we derive gap penalty costs from the probabilities of a gap occurring.

The gap model we use here is the one which is implicit in the vast majority of the present literature. This assumes that gaps are accidents at the DNA level which produce a random insertion or deletion of $k$ amino acids with probability $q(k)$. This probability is independent of the amino acids being inserted or deleted and of any neighbouring amino acids. It is a natural assumption that this probability also depends on the PAM distance $t$ between the two sequences, in particular, gaps in similar sequences are less frequent than gaps in distant ones. Then

$$Pr\{k\text{-gap at distance } t\} = q(k, t)$$

Gap penalties used with dynamic programming are typically of the form $a + bk$. In some cases the values of $a$ and $b$ depend on the PAM distance $t$. This gap penalty function used for dynamic programming will be denoted by $d(k, t)$. For example

$$d(k, t) = a + bk$$

The linear gap penalty function is very popular, in part because it is possible to compute dynamic programming alignments very efficiently using Gotoh’s algorithm[18].

The relation between $d(k, t)$ and $q(k, t)$ is simply

$$d(k, t) = 10 \log_{10} q(k, t)$$

This relation is not well known, so we will prove it here. Readers not interested in the mathematical theory of alignments should skip the rest of this section.
The purpose of aligning sequences with dynamic programming is to find
the alignment which maximizes the probability of the two sequences having
evolved from a common ancestor as opposed to being just random sequences.
Let $A$ and $B$ be two sequences with lengths $n_A$ and $n_B$. Let $A_i$ be the $i^{th}$
amino acid in $A$ (similarly for $B$). Let $f_i$ be the frequency of amino acid $i$
as defined in the previous section. The probability that the two sequences
are random is

$$P_R = Pr \{ \text{random} \} = \prod_{i=1}^{n_A} f_{A_i} \times \prod_{i=1}^{n_B} f_{B_i}$$

For example, given an alignment

$$
\begin{align*}
A_1 & \quad A_2 & \quad - & \quad - & \quad A_3 \\
X_1 & \quad X_2 & \quad X_3 & \quad X_4 & \quad X_5 & \quad X_6 \\
B_1 & \quad B_2 & \quad B_3 & \quad B_4 & \quad B_5 & \quad B_6
\end{align*}
$$

where $X$ denotes the unknown ancestral parent sequence. The probability
of this alignment being the consequence of having a common ancestor is

$$P_A = Pr \{ A, B \text{ evolved from } X \} =$$

$$\sum_{X_1=1}^{20} f_{X_1} (M^{t/2})_{A_1X_1} (M^{t/2})_{B_1X_1} \times \sum_{X_2=1}^{20} f_{X_2} (M^{t/2})_{A_2X_2} (M^{t/2})_{B_2X_2} \times$$

for all the possible $X_1$ which evolved into $A_1$ on one side and into $B_1$ on the
other, and similarly for $X_2$

$$\times q(3, t) f_{B_3} f_{B_4} f_{B_5} \times$$

where this term denotes the probability of a gap of length 3 on one sequence
times the probability of the sequence $B_3B_4B_5$ on the other and

$$\sum_{X_6=1}^{20} f_{X_6} (M^{t/2})_{A_3X_6} (M^{t/2})_{B_6X_6}$$

for the alignment of $A_3$ against $B_6$. For this alignment the quotient of the
probabilities is

$$\frac{P_A}{P_R} = \frac{(M^{t})_{B_1A_1}}{f_{B_1}} \times \frac{(M^{t})_{B_2A_2}}{f_{B_2}} \times q(3, t) \times \frac{(M^{t})_{B_4A_3}}{f_{B_6}}$$
once property grouped and simplified. To find the alignment which maximizes this probability we transform the maximization of a product (of all positive values) into the maximization of a sum by computing the logarithm of each term. To find the alignment which maximizes a sum, we use the dynamic programming algorithm. For historical reasons and to work with simpler numbers, these logarithms are multiplied by 10, which has no effect on the maximization process. Notice that the terms like

$$D[A_1, B_t] = 10 \log_{10} \left( \frac{(M^t)_{B_tA_1}}{f_{B_t}} \right)$$

are exactly the values of the standard Dayhoff matrices. The deletion cost for the above example, under dynamic programming, must be

$$d(3, t) = 10 \log_{10} q(3, t)$$

It is obvious how to generalize this result for any number or length of gaps.

The relationship can be applied in two directions: either we can estimate the probabilities from a sample and compute the appropriate gap penalties or we can compute the probabilities that are implied by a gap penalty function. A linear gap penalty function implies an exponential distribution of probabilities.

$$d(k, t) = a + bk = 10 \log_{10} q(k, t)$$

implies

$$q(k, t) = 10^{a/10} (10^{b/10})^k$$

For example, if $a = -10$ and $b = -3$, then

$$q(k, t) = 0.1000 \times (0.5012)^k$$

This further implies that the probability of a gap, of any length, is

$$\sum_{k=1}^{\infty} q(k, t) = \sum_{k=1}^{\infty} 10^{a/10} (10^{b/10})^k = \frac{10^{a/10}}{10^{-b/10} - 1}$$
in our example, the probability of having a gap would be 10.05%. Finally, the expected length of a gap is

\[ \frac{\sum_{k=1}^{\infty} kq(k,t)}{\sum_{k=1}^{\infty} q(k,t)} = \frac{1}{1 - 10^{b/10}} \]

in our example, the expected length of a gap would be 2.0011.
Chapter 20

A General Introduction to Dynamic Programming
20.1 Longest Common Subsequence

20.2 Dynamic Programming with a Weighted Scoring Matrix

20.3 No Cost for End Gap Alignment

20.4 Local Alignment

20.5 Affine Gap Penalties

20.6 Convex Gap Penalties

20.7 General Gap Penalties

20.8 Aligning DNA versus DNA
Chapter 21

A Maximum Likelihood Pairwise Alignment of Amino Acid Sequences
The pairwise alignment of amino acid sequences in Darwin is performed via dynamic programming using the Dayhoff matrices from Chapter 18 and the gap penalty costs derived in Chapter 19. This chapter describes the Darwin routines and structures available for the various methods of aligning sequences.

21.1 Dynamic Programming

This section gives a short introduction to dynamic programming. Readers not interested in the algorithm underlying the alignment routines should proceed to the next section.

Dynamic programming is a standard algorithmic technique in computer science and is particularly useful for problems such as sequence alignment. Intuitively, dynamic programming works bottom to top computing solutions to subproblems and storing them in a table. In this way, every subproblem is solved exactly once and when a subproblem is re-encountered the answer need not be recomputed.

There are two qualities a problem must possess for dynamic programming to be applicable:

1. Optimal substructure. The optimal solution to the problem must contain within it optimal solutions to subproblems.

2. Overlapping subproblems. The number of distinct subproblems is small.$^1$

It is not hard to see that most sequence alignment problems exhibit both of these behaviours. We will proceed by giving a simple example of how dynamic programming works.

---

$^1$Small usually means the number of possible unique solutions is bounded by a polynomial in the size of the input.
21.1. DYNAMIC PROGRAMMING

21.1.1 The DynProg Function

We want to find the alignment between the following two amino acid sequences that yields the highest score. For this example, we will use a Dayhoff matrix computed at PAM 250.

```plaintext
> CreateDayMatrix(logPAM1, 250);  # compute the Dayhoff matrix DM
> x := 'TCIYGH';
> y := 'TWMRH';
```

The function `DynProg` performs dynamic programming in Darwin.

**Calling Sequences:**

```
DynProg(seq1, seq2, DM)
DynProg(seq1, seq2, DM, l1, l2)
```

**Parameters:**

- `seq1, seq2` : string
- `DM` : DayMatrix
- `l1, l2` : posint

**Returns:** list – three elements (1) score, (2), length of match for sequence 1 and length of match for sequence 2.

**Synopsis:** If no lengths are given as arguments (`l1, l2` are not passed), the `DynProg` function computes the highest similarity score for the two sequences (or two subsequences of `seq1` and `seq2`). This is referred to as the best **global alignment**.

If the lengths are given as arguments, then `DynProg` finds the alignment of length $\max(l_1, l_2)$ with maximum score. The alignment is guaranteed to use $l_1$ elements of `seq1` and $l_2$ of `seq2`. This is referred to as the best **local alignment**.

We begin by tracing through an alignment of sequences $x$ and $y$. First, we ask Darwin to compute the maximum local alignment (we omit the lengths
of x and y.

\[ > \text{DynProg}(x, y, \text{DM}); \]
\[ [3.9785, 3, 3] \]

The score of the alignment is 18.0274 and it uses only 3 amino acid from both x and y. Although \texttt{DynProg} does not give us the alignment, we can deduce easily what matches occurred. The best way to visualize the alignment process is to build a matrix of dimension \texttt{length(x)+1} by \texttt{length(y)+1}.

\textbf{Imagine a beautiful, aesthetic sequence of arrays showing dynamic programming performing an elegant dance over the strings x and y. This will be filled in later.}

The overall score for the alignment is calculated as follows:

\[
\text{Score}(x, y, \text{DM}) = \text{DM}[\text{Sim}, \text{AToInt}(T), \text{AToInt}(T)] + \\
\quad \text{DM}[\text{Sim}, \text{AToInt}(C), \text{AToInt}(W)] + \\
\quad \text{DM}[\text{Sim}, \text{AToInt}(I), \text{AToInt}(M)] \\
= 2.5168 + -1.0187 + 2.4804 \\
= 3.9785
\]

The \texttt{DynProg} routine judiciously decided to cut the alignment at the third position because the cost of matching amino acid Y against amino acid R is \(-1.8160\) and matching H against G is \(-1.4169\). Inclusion of either or both matches decrease the overall score.

\begin{verbatim}
x: T C I Y G H y: T W M R H
2.52 -1.02 2.48 -1.82 -1.42
\end{verbatim}

The remaining alternative is to insert a gap somewhere in the sequence. However, any such gap will incur a penalty of at least \texttt{DM[FixedDel]}=\(-19.8137\). It is easy to verify that all of the following alignments score poorer than the alignment which keeps only the first three bases.
21.1. Dynamic Programming

Alternative #1:

\[
x: \quad T \quad C \quad I \quad Y \quad G \quad H \\
y: \quad T \quad W \quad M \quad R \quad H \\
\begin{array}{cccc}
2.52 & -1.02 & 2.48 & -1.00 \\
6.05 & -19.8 & (gap)
\end{array}
\]

Alternative #2:

\[
x: \quad T \quad C \quad I \quad Y \quad G \quad H \\
y: \quad T \quad W \quad M \quad R \\
\begin{array}{cccc}
2.52 & -1.02 & 2.48 & 0.62 \\
-21.2100 & (gap)
\end{array}
\]

Alternative #3:

\[
x: \quad T \quad C \quad I \quad Y \quad G \quad H \\
y: \quad T \quad W \quad M \quad R \quad H \\
\begin{array}{cccc}
2.52 & -1.02 & 2.48 & -1.82 \\
6.05 & -19.8 & (gap)
\end{array}
\]

Alternative #4:

\[
x: \quad T \quad C \quad I \quad Y \\
y: \quad T \quad W \quad M \quad R \quad H \\
\begin{array}{cccc}
2.52 & -1.02 & 2.48 & -1.82 \\
2.1697 & -19.8 & (gap)
\end{array}
\]

21.1.2 The BackDynProg Function

The DynProg function begins with the leftmost position of both strings and works towards the rightmost positions. Notice however that for some strings, higher scores are possible if one or more bases from the left hand side of the strings are removed. For example,

\[
\begin{array}{l}
> u := 'FRCD': \\
> v := 'SPAD': \\
> \text{DynProg}(u, v, \text{DM}); \\
[1.4750, 4, 4] \\
> u_1 := 'CD': \\
\quad \text{# cut the first two positions} \\
> v_1 := 'AD': \\
> \text{DynProg}(u_1, v_1, \text{DM}); \\
[5.2435, 2, 2]
\end{array}
\]
One way to find this “left starting point” is to apply dynamic programming in the reverse direction on the strings. The left tail of the resulting alignment indicates a good place for a normal “forward” dynamic programming function to begin. The function `BackDynProg` does exactly this in Darwin.

```plaintext
> BackDynProg(u, v, DM);
[5.2435, 2, 2]
> rev_u := 'DCRF';
> rev_v := 'DAPS';
> DynProg(rev_u, rev_v, DM);
[5.2435, 2, 2]
```

### 21.1.3 Global Alignments

If we pass the lengths of the strings as arguments to `DynProg`, the routine finds the best global alignment. That is, the routine does not allow the number of bases in the alignment to fall below $\max(\text{length}(x), \text{length}(y))$. Let us compare the results of using `DynProg` without the lengths specified (local alignment) and with the lengths specified (global alignment). The *Tryptophan* bases W score very well when aligned against themselves (a score of 14.2). The alignment begins very well. The *Tryosine* base Y also scores extremely well when conserved (a score of 7.8). The remaining part of the alignment scores very poorly (all scores are negative). A local alignment algorithm would return only the first six bases from x and y.

```plaintext
> CreateDayMatrices();
> x := 'WWWWYWCDYPHQILWYWWC';
> y := 'WWWWYWPDDGCSTAGPPP';
> DynProg(x, y, DM);
[78.5205, 6, 6]
```

However, the global alignment scores much lower.

```plaintext
> DynProg(x, y, DM, length(x), length(y));
[41.2944, 19, 19]
```
21.2 The Match Structure

The routines DynProg and BackDynProg are low-level commands operating directly on string objects. The alignment routines through the remaining part of this chapter require an offset into a Darwin database be given for the sequences. The routines available for accessing information contained in the Darwin sequence databases are explained in Chapter 8 – Genetic Databases.

The structured type Match holds a partial or totally defined alignment between two amino acid sequences. A match can have as little information as two offsets, for example:

```plaintext
> DB:=ReadDb('Sample/SH2');
> m1 := Match(367, 1338); # offset for sequence #1 and #2.
m1 := Match(367,1338)
```

Offset 367 from DB[string] marks the beginning of the sequence in the first entry of the SH2 database. Offset 1338 is the beginning of the sequence for the second entry (see page 128). Instead of giving the offset directly to Match, we could have used the Sequence and Entry structures to find these offsets for us.

```plaintext
> m1 := Match(op(Sequence(Entry(1))), op(Sequence(Entry(2))));
m1 := Match(367,1338)
> m1[Offset1]; # offset of first sequence
367
> m1[Offset2]; # offset of second sequence
1338
> Entry(m1);
Enter(1,2)
> Offset(m1);
Offset(367,1338)
```

Table 21.1 gives a complete list of the selectors available for this type.
<table>
<thead>
<tr>
<th>Selector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sim</td>
<td>Similarity score of the match</td>
</tr>
<tr>
<td>Offset1</td>
<td>Offset of the first sequence in the database</td>
</tr>
<tr>
<td>Offset2</td>
<td>Offset of the second sequence in the database</td>
</tr>
<tr>
<td>Length1</td>
<td>Length of the match of the first sequence</td>
</tr>
<tr>
<td>Length2</td>
<td>Length of the match of the second sequence</td>
</tr>
<tr>
<td>PamNumber</td>
<td>estimate of the PAM distance between the sequences</td>
</tr>
<tr>
<td>PamVariance</td>
<td>estimate of the PAM variance between the sequences</td>
</tr>
</tbody>
</table>

Table 21.1: Selectors for the Match structured type.

21.3 Matching Routines

This section contains all of the Darwin routines for performing pairwise alignments of two amino acid sequences.
The first Darwin routine for creating an alignment between sequences is the \texttt{GlobalAlign} routine.

**Calling Sequences:**

\texttt{GlobalAlign}(m, DM)

**Parameters:**

- $m$ : Match
- $DM$ : DayMatrix

**Returns:** Match

**Synopsis:** Via dynamic programming and the Dayhoff matrix $DM$, \texttt{GlobalAlign} creates the alignment between sequences $m_1$ and $m_2$ contained in the Match structure $m$.

The similarity scoring in the dynamic programming is done relative to $DM$, therefore, the PAM distance of the alignment is the PAM distance of $DM$.

If the Length1 and Length2 fields of $m$ structure have not been set (i.e. they equal 0), \texttt{GlobalAlign} finds the lengths which maximize the score. Note this score is not necessarily the global best score but the maximum score found by proceeding left to right through $m_1$ and $m_2$ and cutting the right tails (if necessary). If the lengths are defined, \texttt{GlobalAlign} finds the alignment which maximizes the similarity score and forces the overall length of the alignment to be $max(length(m_1), length(m_2))$.

The \texttt{GlobalAlign} function is used when

1. we have a Match with no lengths,
2. we have a Match without a score,
3. we have a Match with a similarity score for a similarity matrix that we would like to redo with a different matrix.
CHAPTER 21. A MAXIMUM LIKELIHOOD PAIRWISE ALIGNMENT OF AMINO ACIDS

```r
> DB:=ReadDb('Sample/SH2');
> CreateDayMatrices(); # calculate matrix DM
> m1 := Match(op(Sequence(Entry(1))), op(Sequence(Entry(2))));
> m1 := GlobalAlign(m1, DM); # DM is PAM 250
m1 := Match(1127.6,367,1338,492,618,250)
> print(m1);
```


\footnotesize

lengths=492,618 simil=1127.6, PAM_dist=250, offsets=367,1338,

identity=41.3%, similarity=23.5%
ID=ABL1_CAEL AC=P03949; DE=TYROSINE-PROTEIN KINASE
ABL-1 (EC 2.7.1.112) (FRAGMENT). OS=CAENORHABDITIS ELEGANS.
ID=ABL2_HUMAN AC=P42684; DE=TYROSINE-PROTEIN KINASE ABL2
(EC 2.7.1.112) (TYROSINE KINASE ARG). OS=HOMO SAPIENS (HUMAN).

NNE---------------------------------------------------------------
::
MGQVGRVGEAPGLQQPQPRIGRSSAARPSGRRDPAGRTTETGFNIFTQHDHFASCVDGFEGDKTGGRSPEALHR

---------------WCEAR------------------------------------------LYSTRKNDASNQRRLGEIGWPS

|::: |

GCDVEPQLNEAIRWSSKENVLLGATESDPNLFWVALYDFVASGDNLSTIKGKRLVGLYNQNGWSEVRKNGQGWPS

FGAPYNLSLDKYTWYHGKIERSDSEAAILGSGITGSLVRESETSIGYITSVRHGDGRVHFHYRINVNDTEKMFTQEVKFP

|:::|

YITPVNSLEKHSWYHGPVSRSAAEYLLSSLINGSLFVRESESSPQQLSISLRYEGRVYHYRINTTADGKVVTAESRF

LGELVHHHSVHADGLICLMMYPASKKDKGRGLFSLSPNPDEWDLRSEIMHMKLLGGQYGDYYEGYWKHDCTIAVF

|:: |

LAEVLHHHSVADGLLVTLHYPACKNPKPT_VYGVSPIHDKNEMERTDITMKHKLGGQYGEVYVGVKKYSITVAVF

LKDAMPMLHEFLAEAAIMKDLLHKNLVRLLGVCTHEAPFYITEFMCNGLLLEYLRRRTDKSLLPILLVQMASQIAGNM
21.3. MATCHING ROUTINES

The similarity score for this alignment is extremely high at 1127.6. This implies that the probability of both sequences coming from a common ancestor, as opposed to being a random alignment, is $10^{112.76}$ times more likely (Section 18.1.3 on page 192).²

The presentation of the alignment information is self-explanatory. The line between the sequences is intended to give users a quick graphical indication of the quality of the alignment. Each character in this middle line corresponds to the quality of the match.

<table>
<thead>
<tr>
<th></th>
<th>exact match</th>
<th>very good match</th>
<th>good match</th>
<th>poor match</th>
<th>very poor match</th>
</tr>
</thead>
</table>

²There are only about $10^{80}$ atomic particles in the universe.
An intuitive rule is that the closer the character is to the vertical bar, the better the match is.

21.3.2 The LocalAlign Function

When two sequences align well except at the ends, it is sometimes desirable to ignore these tails and align the two subsequences (subsets of the original sequences) which align best. The alignment from the previous subsection between the tyrosine protein kinases ABL1_CAENEL (C. Elegans) and ABL2_HUMAN (Humans) is extremely poor at the beginning. Only three bases align before an extremely long gap which is interrupted by a five base alignment before yet another long gap.

The LocalAlign function will perform such a subsequence alignment or local alignment. It is an implementation of the classic Smith-Waterman algorithm [25], a straightforward variant of dynamic programming with some nice properties making it extremely fast.

\[
\begin{align*}
\text{DB} &:= \text{ReadDb('Sample/SH2')} ; \\
\text{CreateDayMatrices}() ; & \quad \text{# calculate matrix DM} \\
\text{m1} &:= \text{Match(op(Sequence(Entry(1))), op(Sequence(Entry(2)))))} ; \\
\text{Glob}_m1 &:= \text{GlobalAlign(m1, DM)} ; \\
\text{Glob}_m1 &:= \text{Match(1127.6, 367, 1338, 492, 618, 250)} ; \\
\text{Loc}_m1 &:= \text{LocalAlign(m1, DM)} ; \\
\text{Loc}_m1 &:= \text{Match(1330.5, 378, 1477, 481, 479, 250)} ; \\
\text{print}('') ;
\end{align*}
\]

lengths=481,479 simil=1330.5, PAM_dist=250, offsets=378,1477,
identity=52.8%, similarity=29.3%

ID=ABL1_CAEL AC=P03949; DE=TYROSINE-PROTEIN KINASE ABL-1 (EC 2.7.1.112)
(FRAGMENT). OS=CAENORHABDITIS ELEGANS.

ID=ABL2_HUMAN AC=P42684; DE=TYROSINE-PROTEIN KINASE ABL2 (EC 2.7.1.112)
(TYROSINE KINASE ARG). OS=HOMO SAPIENS (HUMAN).

TRKNDASNQRLGEIGWVPSNFIAPYNSLDKYTWYHKISRSDEAILGSGITGSFLVRESETSIGQYTISVRHDGRVFH
The similarity score has climbed by more than 200 points (it is now $10^{133.05}$ more likely these sequences share a common ancestor than being simply a random alignment). Comparing the two alignments, one can see that the first two gaps (plus three extra bases of low quality alignment) have been removed. The two gaps in the orginal alignment created with GlobalAlign
where of lengths 91 and 37 respectively. These contributed

\[-19.814 - 1.396 \times (91 - 1) + -19.814 - 1.396 \times (37 - 1) = -215.524\]

to the overall score.

21.4 The LogDelLocalRefine Function

The LogDelLocalRefine function is must the same as the LocalAlign routine from the previous section. It finds the best local alignment for a PAM distance but using a logarithmic deletion cost.

**Calling Sequences:**

\[\text{LogDelLocalRefine}(m, DM)\]

**Parameters:**

- \(m\) : Match
- \(DM\) : DayMatrix

**Returns:** Match

**Synopsis:** Via dynamic programming and the Dayhoff matrix \(DM\), LogDelLocalRefine finds the highest scoring local alignment between sequences \(m_1\) and \(m_2\) contained in the Match structure \(m\).

The similarity scoring in the dynamic programming is done relative to \(DM\), therefore, the PAM distance of the alignment is the PAM distance of \(DM\). The gap scoring function is a logarithmic deletion function.

```plaintext
> DB := ReadDb('Sample/SH2');
> CreateDayMatrices();
> m := Match(op(Sequence(Entry(1))), op(Sequence(Entry(2))));
> align := LogDelLocalRefine(m, DM);
> print(align);
```
The \texttt{LogDelLocalRefine} routine is slower than \texttt{LocalRefine} due to its use of the logarithmic deletion function. However, this deletion function tends to produce better results especially when attempting to detect longer distance homologies [?].

## 21.5 Estimating PAM Distances

The second type of computation we are interested in performing with alignments is the estimation of their PAM distance and variance. We require a little background before continuing. In the previous section, we discussed dynamic programming alignments where the score represents the logarithm of a probability (multiplied by 10). These computations were done for a single similarity matrix, hence for a single PAM distance. In other words, we are estimating the probability of two sequences having diverged from a common ancestor given a distance from this ancestor. It is natural to ask whether we can estimate this distance from the alignment itself. Since the scores are probabilities, we can immediately compute an estimate of the PAM distance by maximum likelihood. In other words, we select the PAM distance which gives an alignment with maximal score. As with all maximum likelihood estimators, this will be an unbiased estimator. Let $S_p(a, b)$ be the score of aligning the sequences $a$ and $b$ at a PAM distance $p$. The maximum likelihood estimator, $q$ is such that

$$S_q(a, b) = \max_p S_p(a, b)$$

This estimation can be done by brute force or with Brent's minimization algorithm. In both cases we require several Dayhoff matrices (for various PAM distances). Since it is likely that we will compute more than one PAM distance in a normal session, it is more economical to pre-compute a dense set of similarity matrices for various PAM distances. The function \texttt{CreateDayMatrices} does exactly this, and assigns its result to the global
variable DMS (§18.3).

Finding the PAM distance which gives the maximum score requires between 13 and 15 alignments for arrays with up to 1000 entries.

Maximum likelihood however, does not allow us to compute the variance of the PAM distance. By assuming that the distribution of PAM distances for a random match is uniform between all possible values (for practical purposes, say up to PAM 1000), we can compute the PAM distance as an expected value:

\[
E[p] = \frac{\int_0^{1000} p 10^{S_p(a,b)/10} dp}{\int_0^{1000} 10^{S_p(a,b)/10} dp}
\]

Recall that the score is 10 times the logarithm (base 10) of a probability. This observation allows us to compute, not only the first moment of \( p \), the expected PAM distance, but all moments. In particular

\[
E[p^2] = \frac{\int_0^{1000} p^2 10^{S_p(a,b)/10} dp}{\int_0^{1000} 10^{S_p(a,b)/10} dp}
\]

The variance of the PAM distance can then be computed, and by using standard statistical tools we can estimate our confidence in \( E[p] \). Let

\[
sd(p) = \sqrt{E[p^2] - E[p]^2}
\]

and the PAM distance is between \( E[p] - 1.96 \times sd(p) \) and \( E[p] + 1.96 \times sd(p) \) 95\% of the time.

The above integrals can be computed by standard methods of numerical integration. Due to the shape of this distribution, and its sharp decrease away from its maximum, the integration does not need to be done through the entire range, but only around its maximum value.
21.6 The FindBestPam Function

The function FindBestPam computes both the average and the variance of
the PAM distance of a given alignment. To accomplish this it needs an array
of similarity matrices calculated at a rather dense range of PAM distances.
The range and coarseness of the array of matrices will have some influence
on the accuracy of the computation. By using the variable DMS, set with
the command CreateDayMatrices (§18.3), these computations tend to be
very accurate. For example

```r
> CreateDayMatrices():                  # assign the range of similarity matrices to DMS.
> DB := ReadDb('cbrg/DB/SwissProt');
> m := Match(op(Sequence(Entry(8948))), op(Sequence(Entry(8972))));
> m := LocalAlign(m, DM);             # align at 250 PAM
> m := FindBestPam(m, DMS);            # find best PAM number and its variance
```

The PAM distance is in the range delimited by

```r
> 1.96 * sqrt(m[PamVariance]);
9.7225
> m[PamNumber] - "; m[PamNumber] + ";
11.5157
30.9607
```

with 95% confidence.

For this particular example, the match is short and hence the uncertainty
of the PAM distance is large.

These functions are implemented using efficient methods and located within
the kernel of the system. The time (in seconds) required to find the align-
ment and then find the best PAM distance for this alignment are as follows:

```r
> rtime(LocalAlign(m, DM));
0.01975501
> rtime(FindBestPam(m, DMS));
0.02198303
```
21.7 The LocalAlignBestPam Function

The function `LocalAlignBestPam` alternates between two different optimizations. Given a Match structure, it will call `LocalAlign` (to improve the local alignment) followed by a call to `FindBestPam` (to find the best PAM distance) until the similarity score no longer increases.

```r
> DB := ReadDb('cbrg/DB/SwissProt');
> CreateDayMatrices();
> m := Match(op(Sequence(Entry(8588))), op(Sequence(Entry(8577))));
> m := LocalAlignBestPam(m, DMS);
> print(m);
lengths=97,110 simil=275.3, PAM_dist=85.9883, offsets=5978923, 5974267,
  identity=40.4%, similarity=12.3%
ID=CY2_RHODI AC=P00083; DE=CYTOCHROME C2 PRECURSOR. OS=RHODOPEUDEMOMAS
  VIRIDIS. RES=1.6
ID=CY22_RHOPA AC=P00091; DE=CYTOCHROME C2. OS=RHODOPEUDEMOMAS PALUSTRIS
  (STRAIN ATCC 17007 / 2.1.37).
  hhhhhhhhhht tt tt tt hhhht hhhhhhhhh hhh
QDAASGEQFVQLVCHSIIGPGAKNVGPVGLNLGRHSGTIEGFAYSDANKNS___GITWTEVFREYIRDPCA____
QDAKAGEAVFKQCMTCIR___ADKNNVPGPALEGVVGKGAAGTYSPLNHNSGEACLGLVWTADNINYNLMPNAFLKKF
  t tt hhhhhhhhh
  __________KIPGKTMIFAGVKDEQKVSDDLIAI Y
LTKGKADQAVGVTMTFKLWNEQNYRKKVAYL
```

The results of a `LocalAlignBestPam` should be compared with a call to `LocalAlign`. Recall `LocalAlign` aligns at one particular PAM distance. Here we calculate it at PAM 250.

```r
> ref := LocalAlign(m, DMS);
> ref := Match(166.1, 5978923, 5974267, 72, 72, 250, 171.449)
> print(ref);
lengths=72, 72 simil=166.1, PAM_dist=250, offsets=5978923, 5974267,
```
21.8. THE DRAWSIMPAM FUNCTION

The function DrawSimPam produces a graph of the similarity score against different PAM distances for a given alignment. This gives the user an intuitive notion of how the similarity changes with changing distances, how precisely defined the maximum is, etc. For example:

> DB := ReadDb("cbrg/DB/SwissProt"): CreateDayMatrices():
> m := Match(op(Sequence(Entry(8588))), op(Sequence(Entry(8577))));
> m := LocalAlignBestPam( m, DMS );
> DrawSimPam(m);

The maximum, 275.4, is located at pam=84

The results are contained in Figure 21.1.
Figure 21.1: The DrawSimPam Function for the Match model. The x axis is the similarity scores while the y axis is the PAM distance.
21.9 The GetOffset Function

So far we have described how to match sequences in the database. When we want to match sequences which are not in the database, we must first translate the strings to database offsets. This is done with the function GetOffset. The GetOffset function requires that DB be currently assigned to a sequence database. We apply this function to strings s1 and s2 before placing them in the Match structure.

```plaintext
> CreateOrigDayMatrix(); # compute the Dayhoff matrices.
> DB := ReadDb('Sample/SH2'); # assign to DB.
> s1 := 'MSRYEKMFLNRMNGAFVPFVTCDPNAESEQYKIMETLVEGADALELGIPFSDP':
> s2 := 'MLLSVNPPLIFPFIVAGDPSPEVTDLALALLEEAGADLLELGVYPYSDP':
> m3 := Match(GetOffset(s1), GetOffset(s2));
> m3 := Match(283090972, 283090904)
> print(LocalAlignBestPam(m3, DMS));
lengths=39,39 simil=164.1, PAM_dist=85.2921, offsets=283090990,283090914,
    identity=48.7%, similarity=20.5%
FVPFVTCDPNAESEQYKIMETLVEGADALELGIPFSDP
!11111111111111111111111111111111
FIPFIVAGDPSPEVTDLALALLEEAGADLLELGVYPYSDP
```

The vast majority of the time, users may completely ignore offsets of a sequence. However, there are some situations (in particular, in the all against all matchings) where one may wish to determine which of the two sequences in a Match structure are native to the sequence database and which is the foreigner.

One must take care using the GetOffset function. Depending on available space, Darwin may insert the foreign sequence either before or after the location of DB[string]. If it is placed before DB[string], this may result in negative offsets.
CHAPTER 21. A MAXIMUM LIKELIHOOD PAIRWISE ALIGNMENT OF AMINO ACIDS
Chapter 22

Searching for Genes
<table>
<thead>
<tr>
<th>Selector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sim</td>
<td>Similarity score for the match.</td>
</tr>
<tr>
<td>NucOffset</td>
<td>Offset of the nucleotide sequence in NucDB.</td>
</tr>
<tr>
<td>PepOffset</td>
<td>Offset of the nucleotide sequence in PepDB.</td>
</tr>
<tr>
<td>NucLength</td>
<td>Length of the nucleotide sequence.</td>
</tr>
<tr>
<td>PepLength</td>
<td>Length of the peptide sequence.</td>
</tr>
<tr>
<td>PamNumber</td>
<td>The estimate PAM distance for the match.</td>
</tr>
<tr>
<td>PamVariance</td>
<td>The estimated PAM variance for the match.</td>
</tr>
<tr>
<td>IntronScoring</td>
<td>The function for scoring introns.</td>
</tr>
<tr>
<td>NucGaps</td>
<td>Gaps in the nucleotide sequence from the alignment.</td>
</tr>
<tr>
<td>PepGaps</td>
<td>Gaps in the peptide sequence from the alignment.</td>
</tr>
<tr>
<td>Introns</td>
<td>The list of suspected introns.</td>
</tr>
</tbody>
</table>

Table 22.1: Selectors for the NucPepMatch structured type.

Assume we have just sequenced the following two chunks of RNA from some prokaryote.

```plaintext
> chunk1 :=
>   'UAUUGCAUAUCGCCAUUGCCGUUCUGUCUCGGCCCCAAUUGCUAUUGAUCUGCCUGACU'.
>   'CGUAGAACUGCGCCGCUAGAUGUACCGGUUUUCACAUGAAGCCCGUAGA'.
>   'GCUUCAAACAAAAUCACUAUGCCAUAAACCCUGAUAUGCGGUUGCUUCCAGACCCUACCAC'.
>   'CGCAGAACCACUGCGCAAGCGAAGAUAAGAACAGCAUGGCGUGCAAGAAG'.
>   'UCCUCAGGGGUCAUGAUUGGCUACCCUACGCAUCCCUGCGGAAAGCCUUUGAGA'.
> chunk2 :=
>   'GAAAACUGCGCGCCGCUAUTCUGUGUCUCCAUCCGUGUUCACGCGGCGUGCCGUCUU'.
>   'GGAGUGGGAAGGCUUCAAGCGCAAGCGCUGAAGGUGGCGCCGCUAUGGGAUGCG'.
```

We assume the 311 and 114 base chunks to be part of the same mRNA molecule, but we do not know

- whether both chunks belong to a single mRNA molecule (they obviously do not overlap),
- whether chunk1 follows chunk2 or vice versa,
22.1. SEARCHING FOR HYPOTHETICAL PEPTIDES

- how many bases are missing between the two chunks,
- what peptide(s) the chunks might code for.

We are going to answer these questions as follows:

1. Search the SwissProt protein database for candidate proteins being reasonably similar to one derived from some encoding of our chunks.

2. Once candidates in SwissProt have been detected, find the proper alignment with the original chunks using direct nucleotide versus peptide dynamic programming.

22.1 Searching for hypothetical peptides

Since we do not know the correct reading frame, we have to translate both chunks in all three reading frames, resulting in six hypothetical peptides to be searched for.

To do the translation, we write a function NucToPep. It takes a nucleotide sequence as input and returns the peptide sequence it encodes. The universal genetic code is built into Darwin as the function GenCodeToInt.

```
> print (GenCodeToInt);
GenCodeToInt: Usage: GenCodeToInt( UUU:string )
Genetic code function. Translates 3 bases (three letters or a name of
    length 3) to the corresponding amino acid number (1 to 20, unknown=21,
    stop=22).
```

Because we want to align all amino acids derived from all three reading frames, even if there are sequencing errors that invalidate parts of the hypothetical peptides, the function continues encoding even if it encounters a stop codon. The stop codon is translated to the unknown amino acid X.

```
> NucToPep := proc (nuc: string)
Create an empty peptide name of appropriate length.
```
>  pep := CreateString(trunc (length (nuc) / 3));
>  for i to length (pep) do

Use the genetic code function to translate the codon; translate stop codons to X.

>  p := GenCodeToInt (nuc[3*i-2..3*i]);
>  pep[i] := If (p = 22, 'X', IntToA (p))
>  od;
>  pep
>  end:

We now can create an array pep containing all hypothetical peptide sequences our chunks code for.

>  pep := [NucToPep(chunk1), NucToPep(1+chunk1), NucToPep(2+chunk1),
>          NucToPep(chunk2), NucToPep(1+chunk2), NucToPep(2+chunk2)];

The hypothetical peptide derived from the third reading frame of chunk2 is:

>  pep[6];
NIGGRSLWSSIGFAALALGWEFQPASRRWRGYS

Next we search SwissProt for similar proteins and store all the matches with pep[i] in ms[i]. Since we have to assume that there are some sequencing errors, none of our six hypothetical proteins will be real, and we do not expect very good similarity to any known protein. We therefore set the similarity score threshold to 70 at pam 250.

>  DB := ReadDb('DB/SwissProt');
>  DM := CreateDayMatrix (NewLogPAM1, 250);
>  ms := CreateArray(1..6);
>  for i to 6 do
>    ms[i] := AlignOneAll (pep[i], DB, DM, 70)
>  od:

The matches found are best shown in a table with one row per database entry and one column per hypothetical peptide. For each entry and each
hypothetical peptide, the table lists the range of aligned positions in the
SwissProt protein.

The following code makes use of the fact that the matches in ms[i] are
sorted by database offset (and entry number).

Create six counters for the six match lists.

```plaintext
> k := CreateArray(1..6, 1):
> do
>  of5 := DB[TotChars];
>  for i to 6 do
>    if k[i] <= length (ms[i]) then
>      of5 := min (of5, ms[i,k[i],Offset1])
>    fi
>  od;
```

Find the current minimum offset of all match lists.

```plaintext
> if of5 = DB[TotChars] then break fi;
```

Get the entry corresponding to that offset and the offset of its first amino
acid.

```plaintext
> entry := GetEntryNumber(of5);
> seq := Sequence (Entry(entry));
```

If at the beginning, print two title lines.

```plaintext
> if sum (k) = 6 then
>   printf (’Entry| 0+chunk1| 1+chunk1| 2+chunk1|’);
>   printf (’ 0+chunk2| 1+chunk2| 2+chunk2|\n’);
>   fi;
>   printf (’%5d’, entry);
```

Print amino acid position range for all matches of this entry.
```plaintext
for i to 6 do
  if k[i] <= length (ms[i]) and
    GetEntryNumber(ms[i,k[i],Offset1]) = entry then
    m := ms[i,k[i]]; p := m[Offset1] - seq[i];
    printf ('%d-%d\n', p+1, p+m[Length1]);
  k[i] := k[i] + 1
  else
    printf ('|')
  fi
od;
printf ('\n')
```

<table>
<thead>
<tr>
<th>Entry</th>
<th>0+chunk1</th>
<th>1+chunk1</th>
<th>2+chunk1</th>
<th>0+chunk2</th>
<th>1+chunk2</th>
<th>2+chunk2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2614</td>
<td>3-64</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3503</td>
<td>110-133</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3703</td>
<td>472-527</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8669</td>
<td>2-53</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9200</td>
<td></td>
<td>27-79</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10083</td>
<td></td>
<td></td>
<td>1348-1428</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12881</td>
<td></td>
<td>188-235</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12899</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>58-82</td>
<td></td>
</tr>
<tr>
<td>12900</td>
<td></td>
<td>183-221</td>
<td>151-184</td>
<td></td>
<td>267-291</td>
<td></td>
</tr>
<tr>
<td>12901</td>
<td></td>
<td>183-221</td>
<td>151-184</td>
<td></td>
<td>267-290</td>
<td></td>
</tr>
<tr>
<td>12902</td>
<td></td>
<td>197-235</td>
<td>165-198</td>
<td></td>
<td>281-306</td>
<td></td>
</tr>
<tr>
<td>12903</td>
<td></td>
<td>188-224</td>
<td></td>
<td></td>
<td>270-294</td>
<td></td>
</tr>
<tr>
<td>12904</td>
<td></td>
<td></td>
<td>163-196</td>
<td></td>
<td>282-307</td>
<td></td>
</tr>
<tr>
<td>12905</td>
<td></td>
<td>226-278</td>
<td></td>
<td></td>
<td>322-345</td>
<td></td>
</tr>
<tr>
<td>12906</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>269-293</td>
<td></td>
</tr>
<tr>
<td>12907</td>
<td></td>
<td>187-223</td>
<td></td>
<td></td>
<td>269-293</td>
<td></td>
</tr>
<tr>
<td>12909</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>323-346</td>
<td></td>
</tr>
<tr>
<td>12910</td>
<td></td>
<td>195-231</td>
<td>161-194</td>
<td></td>
<td>278-303</td>
<td></td>
</tr>
<tr>
<td>12911</td>
<td>170-228</td>
<td>168-240</td>
<td>136-239</td>
<td></td>
<td>249-274</td>
<td></td>
</tr>
<tr>
<td>13427</td>
<td></td>
<td></td>
<td>1-42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14063</td>
<td></td>
<td></td>
<td>97-150</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>----</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>19427</td>
<td>71- 152</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23318</td>
<td>289- 331</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23351</td>
<td>289- 331</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23900</td>
<td>799- 869</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24199</td>
<td>1383-1414</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25060</td>
<td></td>
<td>2214-2289</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33357</td>
<td>19- 77</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33358</td>
<td>19- 77</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33359</td>
<td>19- 77</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>35217</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>252- 287</td>
<td></td>
</tr>
<tr>
<td>36131</td>
<td>198- 291</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36132</td>
<td></td>
<td>54- 147</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>36660</td>
<td>750- 809</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37919</td>
<td>281- 357</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>37920</td>
<td>281- 357</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40388</td>
<td>261- 310</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40696</td>
<td>589- 670</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>43854</td>
<td>738- 787</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The table shows us that the hypothetical peptides derived from the second and third reading frame of the first chunk and from the third reading frame of the second chunk are similar to known proteins.

We want to have a closer look at those proteins which match both chunks. The set of entries which match both chunks is the intersection of the set of entries which matches the first chunk and the set of entries which matches the second chunk. The set of matched entries for each of the six hypothetical peptides can be extracted from the matches.

The Darwin function \texttt{zip} manipulates elements of lists directly and returns a list with the manipulated elements. It avoids creating empty arrays and filling them with \texttt{for} loops. The function \texttt{f} uses \texttt{zip} to extract the entry numbers of \texttt{Offset1} from a list of matches and converts them to a set.

\begin{verbatim}
> f := x -> {op (zip ((y->GetEntryNumber(y[Offset1]))(x)))};
\end{verbatim}
Remember that \( ms \) contains a list of lists of matches. If we apply \( f \) on each element of \( ms \), we get a list of sets of SwissProt entries. \( e[i] \) will contain the set of all SwissProt entries referenced by \( ms[i] \) and hence being similar to \( pep[i] \).

\[
\begin{align*}
> & e := \text{zip}(f, ms); \\
> & e := [(12911, 36680), \{2614, 3503, 3703, 8669, 12881, 12900, 12901, 12902, 12903, \\
> & 12905, 12907, 12910, 12911, 19427, 23318, 23351, 23900, 24199, 33357, 33358, 33369, \\
> & 37919, 37920, 43854}, \{9200, 10083, 12900, 12901, 12902, 12904, 12910, 12911, 13427, \\
> & 14063, 25060, 36131, 36132, 40388, 40696}, \{\}, \{\}, \{12899, 12900, 12901, 12902, \\
> & 12903, 12904, 12905, 12906, 12907, 12909, 12910, 12911, 35217]\]
\end{align*}
\]

\( me \) will hold all entries which match both chunks by matching any hypothetical peptide of each chunk.

\[
\begin{align*}
> & me := (e[1] \cup e[2]) \cap e[3] \\
> & \quad (e[4] \cup e[5]) \cup e[6]); \\
> & me := \{12900, 12901, 12902, 12903, 12904, 12905, 12907, 12910, 12911\}
\end{align*}
\]

Convert this set to an Entry data structure and print the description and species of all these entries:

\[
\begin{align*}
> & \text{me} := \text{Entry}(\text{op} (\text{me})); \\
> & \text{me} := \text{Entry}(12900, 12901, 12902, 12903, 12904, 12905, 12907, 12910, 12911)
\end{align*}
\]

\[
\begin{align*}
> & \text{PrintInfo} (\text{me}, 'DE', 'OS'); \\
\text{Entry 12900:} \\
\text{DE=GLUCOSE-6-PHOSPHATE ISOMERASE, CHLOROPLAST (GPI) (EC 5.3.1.9)} \\
\text{PHOSPHOGLUCOSE ISOMERASE (PGI) (PHOSPHOHEXOSE ISOMERASE) (PHI).} \\
\text{OS=CLARKIA UNGUICULATA.} \\
\text{Entry 12901:} \\
\text{DE=GLUCOSE-6-PHOSPHATE ISOMERASE (GPI) (EC 5.3.1.9) (PHOSPHOGLUCOSE} \\
\text{ISOMERASE) (PGI) (PHOSPHOHEXOSE ISOMERASE) (PHI).} \\
\text{OS=ESCHERICHIA COLI.} \\
\text{Entry 12902:} \\
\text{DE=GLUCOSE-6-PHOSPHATE ISOMERASE (GPI) (EC 5.3.1.9) (PHOSPHOGLUCOSE}
\end{align*}
\]
22.1. SEARCHING FOR HYPOTHETICAL PEPTIDES

ISOMERASE (PGI) (PHOSPHOHEXOSE ISOMERASE) (PHI).
OS=HAEMOPHILUS INFLUENZAE.

Entry 12903:
DE=GLUCOSE-6-PHOSPHATE ISOMERASE (GPI) (EC 5.3.1.9) (PHOSPHOGLUCOSE ISOMERASE) (PGI) (PHOSPHOHEXOSE ISOMERASE) (PHI) (NEUROLEUKIN) (NLK).
OS=HOMO SAPIENS (HUMAN).

Entry 12904:
DE=GLUCOSE-6-PHOSPHATE ISOMERASE (GPI) (EC 5.3.1.9) (PHOSPHOGLUCOSE ISOMERASE) (PGI) (PHOSPHOHEXOSE ISOMERASE) (PHI).
OS=KLUYVEROMYCINES LACTIS (YEAST).

Entry 12905:
DE=GLUCOSE-6-PHOSPHATE ISOMERASE (GPI) (EC 5.3.1.9) (PHOSPHOGLUCOSE ISOMERASE) (PGI) (PHOSPHOHEXOSE ISOMERASE) (PHI).
OS=LEISHMANIA MEXICANA.

Entry 12907:
DE=GLUCOSE-6-PHOSPHATE ISOMERASE (GPI) (EC 5.3.1.9) (PHOSPHOGLUCOSE ISOMERASE) (PGI) (PHOSPHOHEXOSE ISOMERASE) (PHI).
OS=SUS SCROFA (PIG).

Entry 12910:
DE=GLUCOSE-6-PHOSPHATE ISOMERASE (GPI) (EC 5.3.1.9) (PHOSPHOGLUCOSE ISOMERASE) (PGI) (PHOSPHOHEXOSE ISOMERASE) (PHI).
OS=SACCHAROMYCINES CEREVISIAE (BAKER'S YEAST).

Entry 12911:
DE=GLUCOSE-6-PHOSPHATE ISOMERASE (GPI) (EC 5.3.1.9) (PHOSPHOGLUCOSE ISOMERASE) (PGI) (PHOSPHOHEXOSE ISOMERASE) (PHI).
OS=ZYMONOMAS MOBILIS.

The above tables lead to the following conclusions:

- Because there are similarities for more than just one reading frame, the chunks most likely contain sequencing errors.
- Both chunks seem to belong to the same mRNA sequence, and chunk2 obviously follows chunk1 in that sequence.
• The mRNA seems to be transcribed from some gene for phosphoglucone isomerase.

### 22.2 Direct nucleotide versus peptide alignment

To find the probable location of sequencing errors, the gap between the chunks and the most similar protein, we want to align the RNA sequence built from both chunks directly with all the proteins that match both chunks, using nucleotide versus peptide dynamic programming.

While peptide versus peptide functions deal with a single peptide database in variable DB, all nucleotide versus peptide (NP) functions require two databases to be loaded: a nucleotide database, accessible by variable NucDB, and a peptide database, accessible by variable PepDB.

Most peptide versus peptide functions have a corresponding NP function. The structure that holds an NP match is called NucPepMatch, finding the best possible alignment for a given pam distance is called LocalAlignNucPep, finding the best possible alignment at the optimal pam distance is called BestLocalAlignNucPep.

```plaintext
> print (NucPepMatch, LocalAlignNucPep, BestLocalAlignNucPep);
NucPepMatch: Usage: NucPepMatch(  )
Structure to hold nucleotide/peptide matches.
- Selectors:
  Sim: real, NucOffset: integer, PepOffset: integer,
  NucLength: integer, PepLength: integer, PamNumber: real,
  PamVariance: real, IntronScoring: {function, string, 0},
  NucGaps: list(posint..posint), PepGaps: list(posint..posint),
  Introns: list([real, posint..posint])
- Formats:
  NucPepMatch (NucEntries: structure, PepEntries: structure)
  NucPepMatch (NucOffset, PepOffset)
  NucPepMatch (Sim, NucOffset, PepOffset, NucLen, PepLen)
  NucPepMatch (Sim, NucOffset, PepOffset, NucLen, PepLen, PamNumber)
```
NucPepMatch (Sim,NucOffset,PepOffset,NucLen,PepLen,PamNumber,PamVariance)  
NucPepMatch (Sim,NucOffset,PepOffset,NucLen,PepLen,PamNumber,PamVariance,  
  IntronScoring())  
NucPepMatch (Sim,NucOffset,PepOffset,NucLen,PepLen,PamNumber,PamVariance,  
  IntronScoring(),NucGaps,PepGaps,Introns).  

LocalAlignNucPep: Usage: LocalAlignNucPep( npm:NucPepMatch, D:DayMatrix )  
Return the NucPepMatch between the nucleotide and the peptide of npm with the  
  highest score.  

BestLocalAlignNucPep: Usage: BestLocalAlignNucPep( m:NucPepMatch )  
Apply LocalAlignNucPep and FindNucPepPam until a maximum is found.  

To do nucleotide versus peptide dynamic programming, the actual steps are:  

1. **Load a peptide database into variable PepDB.** Since we already loaded  
   SwissProt into DB, we can simply assign it.  

   > PepDB := DB:  

2. **Find the offset of the peptide sequence to be aligned.** As an example  
   we want to align chunk1 with the peptide sequence that matches its  
   first reading frame only.  

   > first := e[1]: for s in e[2..6] do first := first minus s od:  
   > pepOffset := Sequence (Entry(first[1]))[1]:  

3. **Load a nucleotide database into variable NucDB.** Even if we will not use  
   a sequence from it right now, we load the funghi division of the EMBL  
   database. We will use funghi sequences later on in this Darwin session.  

   > NucDB := ReadDb('DB/EMBL.fun'):  

4. **Find the offset of the nucleotide sequence to be aligned.** In our example,  
   the sequence is in chunk1, which is not in the database. We find the  
   offset of this chunk with respect to EMBL.fun.  

   > nucOffset := GetOffset (chunk1, NucDB):
5. Create an NucPepMatch from these offsets.

\[ m := \text{NucPepMatch (nucOffset, pepOffset)}; \]
\[ m := \text{NucPepMatch(-940568940, 25135271)} \]

6. Use one of the NP dynamic programming functions to align the sequences. The most convenient way to create the Dayhoff matrices required for those is

\[ \text{CreateDayMatrices();} \]

To align the sequences at 250 pam, use

\[ \text{LocalAlignNucPep (m, DM);} \]
\[ \text{NucPepMatch(65.8523, -940568787, 25136030, 150, 50, 250, 0)} \]

To get the optimal alignment at optimal pam, use

\[ \text{m1 := BestLocalAlignNucPep(m);} \]
\[ \text{m1 := NucPepMatch(66.4941, -940568787, 25136030, 150, 50, 235.6046, 2687.1169, 0, [], [ ])} \]

Note that nucleotide versus peptide dynamic programming aligned only 59 amino acids yielding a similarity score of 66.5, while 59 amino acids of the hypothetical sequence were aligned with a similarity score of 113.7. This is because the hypothetical peptide contains an X that aligns at zero cost where the nucleotide contains a stop codon that would contribute

\[ \text{m1[StopSimil];} \]
\[ \text{Error, (in NMatch_select) Invalid selector, StopSimil} \]

to the total score. This is an example how aligning hypothetical peptides fails in handling stop codons. In fact, this doubtful match would not have been considered as a candidate by NP dynamic programming because it would not have reached the similarity threshold used above.
7. Print the NucPepMatch.

> print (mi);

See figure ???. Both percent identity and similarity are rather small, the estimates of pam distance and variance are large. All this indicates a very doubtful match.

The first two lines of a printed NP match contain information about the alignment, the next lines information about the sequences in the database:

- **offsets** are the database offsets of the sequences,
- **lengths** are the lengths of the aligned nucleotide and peptide sequences,
- **simil** is the similarity score obtained from dynamic programming,
- **pam** is an estimate of the PAM distance between the two sequences,
- **var** is an estimate of the variance of the PAM distance,
- **StopSimil** is the similarity score used for aligning stop codons,
- **codons** is the number of amino acids/codons that were aligned,
- **stops** is the number of stop codons that were aligned,
- **identity** is the percentage of aligned positions which are identical,
- **similarity** is the percentage of aligned positions not identical but with a positive value in the Dayhoff matrix.

Then the alignment is shown in five rows:

- the coordinates of the original nucleotide sequence (first base is 1),
- the nucleotide sequence itself,
• the hypothetical peptide sequence imposed by the alignment, marking
  the three bases of the codon with a $<$, the amino acid itself (or $\$\$ for a
  stop codon), and a $>$,

• the quality of the match indicated by a character just like in peptide
  alignments:

<table>
<thead>
<tr>
<th></th>
<th>exact match</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>very good match</td>
</tr>
<tr>
<td>:</td>
<td>good match</td>
</tr>
<tr>
<td>.</td>
<td>poor match</td>
</tr>
<tr>
<td>#:</td>
<td>very poor match</td>
</tr>
</tbody>
</table>

• the peptide sequence from the database.

For nucleotide indels longer than 4 bases, the length is shown in parentheses.

We now want to find the optimal alignment/pam of the concatenated
chunks with all the entries in me.

> chunk := chunk1.chunk2;
> chunkOffset := GetOffset (chunk, NucDB):

The second statement again finds the offset of our chunk with respect to
NucDB. We now are ready to BestLocalAlignNucPep our RNA sequence
with all SwissProt entries we are interested in. For each sequence offset
$s$ defined by the list of entries in me, we create an NucPepMatch $m$ of the
nucleotide at offset chunkOffset (our two concatenated chunks) with the
peptide at offset $s$, then BestLocalAlignNucPep the match and append it
to the list of results. For Sequence to work on entries in PepDB, we need to
assign it to DB:

> DB := PepDB:
> res := []:
> for s in Sequence (me) do
>  m := NucPepMatch (chunkOffset, s);
>  res := append (res, BestLocalAlignNucPep(m))
> od:
We sort the matches by similarity score (highest first):

```r
> res := sort (res, x -> -x[Sim]);
```

```r
res := [NucPepMatch(1504.9172, -940149966, 8954634, 423, 150, 4, 0.1652, 3, 0.5986, 0, [100
.. -1, 198.. -1, 275.. -1, 313.. -1, 390..390], [106..113], []), NucPepMatch(376.3618, -940149966, 8946239, 423, 152, 88.2429, 122.6461, 0, [100.. -1, 198.. -1, 275.. -1, 313.. -1, 393..393], [86..87, 108..115], []), NucPepMatch(376.3590, -940149966, 8945323, 423, 152, 87.5727, 121.5655, 0, [100.. -1, 198.. -1, 275.. -1, 313.. -1, 387..387], [86..87, 108..115], []), NucPepMatch(350.2784, -940149966, 8944425, 423, 152, 82.4499, 133.0105, 0, [100.. -1, 198.. -1, 239.. -1, 313.. -1, 390..390], [81..81, 100..100, 108..115], []), NucPepMatch(350.1464, -940149966, 8953764, 420, 152, 94.6648, 140.3467, 0, [100.. -1, 198.. -1, 275.. -1, 313.. -1, 393..393], [86..88, 109..116], []), NucPepMatch(336.6448, -940149966, 8950944, 423, 153, 84.4001, 115.0486, 0, [70.. -1, 76..78, 86..86, 100.. -1, 198.. -1, 233..235, 275.. -1, 313.. -1, 390..390], [24..25, 87..89, 109..116], []), NucPepMatch(330.6823, -940149966, 8947208, 423, 153, 88.6275, 124.4639, 0, [100.. -1, 198.. -1, 233..235, 275.. -1, 313.. -1, 390..390], [22..22, 87..89, 109..116], []), NucPepMatch(307.8176, -940149966, 8949040, 420, 157, 87.1001, 118.8091, 0, [79.. -1, 86..86, 100.. -1, 198.. -1, 243..244, 259.. -2, 275.. -1, 313.. -1, 387..387], [27..27, 74..77, 92..94, 114..121], []), NucPepMatch(290.7973, -940149966, 8948096, 423, 155, 102.2532, 156.4797, 0, [100.. -1, 208..209, 275.. -1, 313.. -1, 393..393], [74..77, 89..90, 111..118], []))
```

By printing the best match, we see that there are probably sequencing errors at positions 102-103, 200-201, 277-278, 392 and that about 25 bases are missing between the two chunks. See figure ??.

```r
> print (res[1]);
```

Of course, we could have been searching the database with `LocalAlignNucPep` only, without first looking for hypothetical peptides. However, scanning an entire protein database with NP dynamic programming is more expensive than scanning it with peptide versus peptide dynamic programming over the hypothetical proteins. The following simple experiment reveals this: We first measure the time it requires to align the three hypothetical peptides with the peptide of the best match above. To make it use a significant amount of time, we repeat it 10 times.
> ofs := Sequence (me)[i];
> PepTime := time ()
> for rep to 10 do
> for i to 3 do
>   LocalAlign (Match (GetOffset (pep[i]), ofs), DM)
> od
> od:
> PepTime := time () - PepTime;
PepTime := 0.6900

We then measure the time it requires to do the same with direct NP alignment.

> ofs1 := GetOffset (chunk1, NucDB):
> NucTime := time ()
> for rep to 10 do
>   LocalAlignNucPep (NucPepMatch (ofs1, ofs), DM)
> od:
> NucTime := time () - NucTime;
NucTime := 15.5600

NP dynamic programming is therefore about 23 times slower than standard peptide versus peptide dynamic programming. Of course, it is much more accurate and has the following advantages:

- Sequencing errors do not damage the entire alignment.
- Nucleotide indels need not be aligned to codon boundaries.
- Stop codons are handled properly. In NP alignments, the similarity score for matching an amino acid with a stop codon is a parameter of a match (called StopSimil).
- Introns causing frame shifts can be detected.

As an example for intron detection, we align the gene for ligninase isozyme H8 of Phanerochaete chrysosporium (in EMBL.fun) with the corresponding protein (in SwissProt).
22.2. DIRECT NUCLEOTIDE VERSUS PEPTIDE ALIGNMENT

> DB := NucDB ('ligninase', 'h8', 'chrysosporium');
Entry(2170,2171)

> nucOffset := Sequence ('') [1];
> DB := PepDB ('ligninase', 'h8', 'chrysosporium');
Entry(20766)

> pepOffset := Sequence ('') [1];

> print (BestLocalAlignNucPep (NucPepMatch ((nucOffset, pepOffset))));

The result is shown in figure ??: The GT-AG rule for introns states that introns start with the dinucleotide GT and end with AG. Keeping this in mind, we immediately see where exactly the introns are. Note that 4 of the 7 introns cause frame shifts and hence frame encoding and peptide versus peptide alignment will never work.

In our current model, introns are scored as indels. If an intron becomes too long, the scoring for it as an indel may avoid inclusion of an exon and shorten the match. An example is the gene for yeast's ribosomal protein L29. We want to align this gene with the protein it encodes. We first search for the gene in NucDB.

> DB := NucDB ('yeast', 'ribosom', 'L29');
Entry(1823,1824,3231,3361,4639,4640,6187)

> PrintInfo ('', 'DE', 'AC');
Entry 1823:

DE=Neurospora crassa crp-1 gene for ribosomal protein homologous to yeast cyH2 gene encoding L29
AC=X06320;
Entry 1824:

DE=Neurospora crassa crp-1 mRNA for ribosomal protein homologous to yeast cyH2 gene encoding L29
AC=X13254;
Entry 3231:
DE=S.cerevisiae chromosome III complete DNA sequence
AC=X59720; S43845; S49180; S58084; S93798;

Entry 3361:
DE=Yeast CYH2 gene for ribosomal protein L29
AC=X01573; K01162;

Entry 4639:
DE=S.cerevisiae CYH2 gene encoding ribosomal protein L29, 5’ end.
AC=M19490;

Entry 4640:
DE=yeast (S.cerevisiae) ribosomal protein L29, (CYH2) gene, complete
cds.
AC=K01162;

Entry 6187:
DE=S.pombe rpgL29 gene for ribosomal protein L29
AC=X57207;

Since this gene is contained several times in the database, we choose the one
with accession number X01573.

> nucEntry := Entry(3361);
> nucOffset := Sequence (nucEntry)[1]:

To find the SwissProt entry expressed by this gene, we look at the DR tag
of the EMBL entry. It contains references to other databases.

> PrintInfo (nucEntry, 'DR');
Entry 3361:
DR=SWISS-PROT; P02406; RL2A_YEAST.

> DB := PepDB:  pepEntry := SearchDb ('P02406;');
pepEntry := Entry(31443)

> pepOffset := Sequence (pepEntry)[1]:
> m := NucPepMatch (nucOffset, pepOffset):
Our first attempt is to get a single best alignment at distance 1 pam (remember that we are aligning a protein with its gene).

```plaintext
> pam1 := SearchDayMatrix(1, DMS);
pam1 := DayMatrix(Peptide, pam=1.00432, Sim: max=18.820, min=-37.514,
    max offdiag=-11.346, del=-37.626-1.396*(k-1))

> m1 := LocalAlignNucPep(m, pam1);
m1 := NucPepMatch(1648.1674,7042031,21681479,399,133,1.0043,0)
```
Chapter 23

Datastructures for Genetic Datasets
Chapter 24

Phylogenetic Trees
Chapter 25

Phylogenetic Trees

The creation of phylogenetic or evolutionary trees is one of the oldest problems in biology.

25.1 Introduction

There are basically two methodologies within Darwin for building phylogenetic trees.

25.2 Scoring Functions

25.2.1 Parsimony Based Methods

DNA Based Scoring Methods

Fitch’s Algorithm

Eck and Dayhoff’s Algorithm

Generalized Parsimony

Felsenstein’s Algorithm
Darwin's Version

Amino Acid Based Scoring Methods

Kinoshuro's Algorithm

25.2.2 Maximum Likelihood Methods

25.2.3 Distance Based Methods

25.3 Searching for Tree Topologies

25.3.1 Tree Enumeration

25.3.2 Initial Topologies

25.3.3 Local Search Techniques
Chapter 26

Multiple Sequence Alignments
26.1 Scoring a Multiple Sequence Alignment

26.1.1 Sum of Pairs

26.1.2 Travelling Salesman Tours

26.1.3 Other Methods

26.2 Constructing a Multiple Sequence Alignment

26.2.1 Multi-dimensional Dynamic Programming

26.2.2 Bootstraping Techniques

26.2.3 Building a Multiple Sequence Alignment Using a Phylogenetic Tree

26.2.4 The Darwin Technique

26.2.5 The New Darwin Technique

26.3 Gap Adjustment

Calling Sequences:
\[
\text{CreateMultiAlign}(malign, method)
\]

Parameters:
- \textit{malign} : MultiAlign
- \textit{method} : string

Returns: MultiAlign

Synopsis:
### Table 26.1: Selectors for the MultiAlign structured type.

<table>
<thead>
<tr>
<th>Selector</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq</td>
<td>The list of sequences</td>
</tr>
<tr>
<td>Labels</td>
<td>The list of labels (same order as Seq)</td>
</tr>
<tr>
<td>Align</td>
<td>The alignment of the sequences in Seq</td>
</tr>
<tr>
<td>Tree</td>
<td>The phylogenetic tree formed from Align.</td>
</tr>
<tr>
<td>PAS</td>
<td>The probabilistic ancestral sequence.</td>
</tr>
<tr>
<td>AllAll</td>
<td>The results of an “all vs. all” on Seq.</td>
</tr>
<tr>
<td>Title</td>
<td>The title for the alignment.</td>
</tr>
<tr>
<td>SIA</td>
<td>Surface/Inside/Active site prediction.</td>
</tr>
<tr>
<td>Parse</td>
<td>A parse prediction.</td>
</tr>
<tr>
<td>Info</td>
<td>Additional information and annotation.</td>
</tr>
<tr>
<td>Score</td>
<td>The score of the alignment.</td>
</tr>
<tr>
<td>Max</td>
<td>The maximum possible score (via TSP ordering).</td>
</tr>
</tbody>
</table>
Chapter 27

Probabilistic Ancestral Sequences
27.1 Algorithms for Probabilistic Sequence Creation

27.2 Comparing Two Probabilistic Sequences

27.3 Consensus Sequence Algorithms
Chapter 28

Recognizing Proteins by Weight
In some cases, recognition of proteins can be done by fragmenting the protein according to certain pattern and using the molecular weights of the fragments as a trace. This method is not effective to find the composition of an unknown protein, but it is effective in locating an unknown sample if its sequence is recorded in a protein database.

One of the ways of breaking a protein into smaller pieces according to a certain pattern is by using enzymes which digest the protein. For example, trypsin breaks a protein after every Arginine (R) or after every Lysine (K) not followed by a Proline (P). It is not very difficult, given the rules, to write a function which will do the theoretical digestion of a sequence. The function for trypsin is

```plaintext
> DigestTrypsin := proc( s: string )
>     description 'break a peptide sequence as if digested by trypsin';
>     ls := length(s);
>     res := NULL;
>     i := 1;
>
>     for j to ls-1 do
>         if s[j] = 'R' or s[j] = 'K' and s[j+1] <> 'P' then
>             res := res, s[i..j];
>             i := j+1
>         fi
>     od;
>
>     [res, s[i..ls]]
> end:
```

the fragments will be defined between i and j

```plaintext
> for j to ls-1 do
>     if s[j] = 'R' or s[j] = 'K' and s[j+1] <> 'P' then
>         res := res, s[i..j];
>         i := j+1
>     fi
> end;
```

collect the last fragment

```plaintext
> [res, s[i..ls]]
> end:
```

If we would subject the protein

```plaintext
> p := 'YKVTLVDSRREGDAEDQGLDLKPSRAGACSTCAGKIVSGDLDDQIEK0':
```

to the action of trypsin, we would obtain 7 fragments:
28.1. THE COMPARISON ALGORITHM

The molecular weight of fragments can be found experimentally by mass spectrometry methods to a good level of accuracy. More importantly, these methods typically require very small samples in the order of fractions of pico-moles.

In Darwin we can compute the theoretical molecular mass of a protein sequence by using the function GetMolWeight.

```plaintext
> dp := DigestTrypsin(p);
> print(GetMolWeight);
GetMolWeight: Usage: GetMolWeight( s:{string,array(string)} )
Compute the molecular weight of an amino acid sequence.

> GetMolWeight(dp);
[309.3440, 829.8990, 174.1880, 209.1640, 867.9860, 1446.5000, 75.0520]
```

The problem of identifying a sampled protein can be reduced to digesting the protein with an enzyme, finding the molecular weights of each of the pieces and then comparing this set of weights to what would be obtained from the digestion of each protein in the database. The process can be repeated with several different enzymes to increase its selectivity.

The purpose of this chapter is to describe an algorithm to perform this matching against the database in an efficient way. Secondly we are interested in estimating when a match of weights is significant. This algorithm is available in Darwin under the name SearchMassDb. Readers interested just in its use should skip to the example section. As we have done with other algorithms, the next sections describe the algorithms and their theory.

28.1 The comparison algorithm

If our mass measures were perfect and our sequence database contained all searched sequences, this would be an almost trivial problem. Search a vector
of weights against all possible vectors of weights computed from the sequence database. This problem is known in computer science as multidimensional search. This is, unfortunately, not the case for the following reasons:

(a) The recording of molecular mass is subject to a relative error, in general less than 1% but not exact enough as to identify even very short sequences of amino acids.

(b) The searched sequence may not be verbatim in the database, maybe a close relative of the sequence is. In this case the searched sequence and the target could differ due to mutations, insertions and deletions. This will cause some molecular weights to be different.

(c) The mutations in the database sequence can cause the digestion to be different, splitting into more or fewer fragments. This will cause a complete mismatch of weights involving such fragments.

(d) Impurities in the sample and in the digesters may produce spurious data in the searched sample.

(e) The fragmentation (digestion) although in general accurate, is not 100% deterministic. Partial digestion or incorrect ones are also possible.

For all these reasons we have to choose a matching method which will tolerate errors both in the sample and in the database.

The algorithm we will use, for a single digestive enzyme, can be stated in relatively simple terms:

(i) Find a set of molecular weights of the digested protein (usually found by experimental means).

(ii) Digest (theoretically) every sequence in the database and find the molecular weights of the fragments.
28.2. PROBABILITY FOUNDATIONS

(iii) Compute the probability that a match of the given weights against the computed ones happens at random.

(iv) Record the $m$ lowest probabilities.

This algorithm returns the $m$ most likely candidate sequences from the database. Analysis of these sequences and their probabilities will normally reveal whether we have found a match, a hint or just random noise.

28.2 Probability foundations

In this section we will derive the formulas to compute the probabilities which will be used in determining the most significant similarities and finally to determine whether these similarities are significant or not.

First we will abstract the problem in the following way. Suppose that we select $k$ small intervals, each one of length $\epsilon$ in the range 0 to 1. Suppose that $\epsilon$ is small enough so that we can distribute the $k$ intervals at random without a significant danger of overlapping them. We have now $k\epsilon$ of the interval covered and $1-k\epsilon$ uncovered. The second step consists of choosing $n$ random points ($n \geq k$) in the unit interval. We want to determine the probability that all the $k$ intervals receive at least one of the $n$ random points.

Let $a_1, a_2, ..., a_k, b$ be formal variables.

$$G_{k,n,\epsilon} = (a_1 \epsilon + a_2 \epsilon + ... + a_k \epsilon + b(1 - k\epsilon))^n$$

is a generating expression of all the events of this experiment, where $a_i$ corresponds to an object in box $i$ and $b$ corresponds to an object outside all the boxes. For example, the coefficient in $G_{k,n,\epsilon}$ of $a_1^2b^{n-2}$ gives the probability of two of the points falling in box 1 and all the rest falling outside all the other boxes. The coefficient of $b^n$ gives the probability of all points falling outside of all the intervals.

The probability we want to compute is the one which includes all the terms with each $a_i$ to the power 1 or higher. These terms can be computed directly
by computing which are the coefficients independent of \( a_i \) (\( a_i \) to the power 0) and subtracting this from the original \( G_{k,n,\epsilon} \). To compute the probability, all the \( a_i \) and \( b \) can be set to 1. For example,

\[
G_{2,n,\epsilon} = (a_1 \epsilon + a_2 \epsilon + b(1-2\epsilon))^n
\]

The coefficient in \( a_1 \) to the power 0 is just equivalent to substituting \( a_1 = 0 \). So the component of \( G_{2,n,\epsilon} \) which has at least a linear factor in \( a_1 \) is

\[
G_{2,n,\epsilon}^* = (a_1 \epsilon + a_2 \epsilon + b(1-2\epsilon))^n - (a_2 \epsilon + b(1-2\epsilon))^n
\]

Repeating the same for \( a_2 \) in \( G_{2,n,\epsilon}^* \) we find

\[
G_{2,n,\epsilon}^{**} = (a_1 \epsilon + a_2 \epsilon + b(1-2\epsilon))^n - (a_2 \epsilon + b(1-2\epsilon))^n - (a_1 \epsilon + b(1-2\epsilon))^n + (b(1-2\epsilon))^n
\]

Substituting \( a_1 = a_2 = b = 1 \) and simplifying we obtain

\[
P_{2,n,\epsilon} = 1 - 2(1 - \epsilon)^n + (1 - 2\epsilon)^n
\]

\( P_{2,n,\epsilon} \) is the probability that when selecting \( n \) points both intervals of length \( \epsilon \) include at least one point each. A more general analysis will show that

\[
P_{k,n,\epsilon} = 1 - \left( \frac{k}{1} \right)(1 - \epsilon)^n + \left( \frac{k}{2} \right)(1 - 2\epsilon)^n - \ldots - \left( \frac{k}{i} \right)(-1)^i(1 - ki\epsilon)^n
\]

\[
= \sum_{i=0}^{k} \left( \frac{k}{i} \right)(1 - i\epsilon)^n
\]

Although exact, this formula is extremely ill conditioned for computing these probabilities when \( \epsilon \) is very small and \( n \) and \( k \) are relatively large. This cannot be ignored. Assuming that \( n\epsilon \) remains of moderate size (neither too large nor too small), the following asymptotic approximation can be used

\[
P_{k,n,\epsilon} = (1 - e^{-n\epsilon})^k \left( 1 + \frac{k\epsilon(e^{n\epsilon} - k)}{2(e^{n\epsilon} - 1)^2} - O(\epsilon^2) \right)
\]

For all practical purposes the most significant term, \((1 - e^{-n\epsilon})^k\), is an excellent approximation for the probability.
If we have \( k \) weights from a sampled protein and we match them against an unrelated digested protein which splits into \( n \) fragments, we can view the weights of these \( n \) fragments as random.

For example, suppose that we are given \( k = 3 \) weights of fragments: \( w_1 = 441 \), \( w_2 = 893 \) and \( w_3 = 1415 \) found by mass spectrometry from an unknown protein. Suppose that the theoretical digestion of a protein in the database would give the weights \( v_1 = 410 \), \( v_2 = 892 \), \( v_3 = 925 \), \( v_4 = 1218 \) and \( v_5 = 1421 \). If we require an exact match, none of the values will match. If we accept a tolerance radius \( r = 1 \), then \( v_2 \) is within the range of \( w_2 \), 
\[
|w_2 - v_2| = |893 - 892| \leq 1.
\]
If we increase the tolerance to radius \( r = 6 \), then two weights will be within range, \( |w_2 - v_2| \leq 6 \) and \( |w_3 - v_5| \leq 6 \). Finally we need to increase the tolerance to radius \( r = 31 \) to include all the given weights in ranges. This information can be summarized by a table

<table>
<thead>
<tr>
<th>( n )</th>
<th>( k )</th>
<th>( r )</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>31</td>
</tr>
</tbody>
</table>

If the weights \( v_i \) were from a totally unrelated protein, they could be viewed as random values in the given range. Let's assume, for the above example, that the weights were selected in the range 400 to 1500. We can now establish an equivalence between this example and the previously computed probabilities for each of the entries in the table. For example, the second line corresponds to hitting one interval with radius 1 out of 5 random choices. The third line corresponds to hitting two intervals each one within radius 6 out of 5 random points.

Before we compute these probabilities with the above formulas, we need to consider the nature of error in molecular weights. These errors are not of an absolute magnitude, rather they are relative to the mass we are measuring.
I.e., an error of 1 for a mass of 1000 is equivalent to an error of 10 for a mass of 10000. Since our probability was derived on the assumption that all the intervals were of the same length, it is easiest to apply a transformation to the masses, such that a constant relative error becomes a constant absolute error. In terms of our example, an error of 1 for the mass of \( w_1 \), 441 would be equivalent to an error of \( 1415/441 \approx 3.2 \) for \( w_3 \) with mass 1415. A standard transformation, when we want to linearize relative values, is to use the logarithms of the measures. So instead of working with the weights, we will work with the logarithms of the weights. A tolerance of \( \epsilon \) in the logarithms is equivalent to a relative tolerance of \( e^{\pm \epsilon} \approx 1 \pm \epsilon \) in the relative values.

\[
| \log w_i - \log v_j | \leq \epsilon
\]

implies that

\[
1 - \epsilon \approx e^{-\epsilon} \leq \frac{w_i}{v_j} \leq e^\epsilon \approx 1 + \epsilon
\]

or

\[
\frac{|w_i - v_j|}{v_j} \leq \epsilon
\]

Finally, since our intervals in the theoretical derivation were based on random variables distributed in \((0,1)\), we must make the relative errors relative to the size of the entire interval, or divide them by \( \log w_{\text{max}} - \log w_{\text{min}} \) where \( w_{\text{max}} \) and \( w_{\text{min}} \) are the maximum and minimum weights that we will consider in the sample.

In our example, \( w_{\text{max}} = 1500 \) and \( w_{\text{min}} = 400 \) so the radius of the error for \( w_1, v_1 \) is

\[
\frac{\log 441 - \log 410}{\log 1500 - \log 400} \approx 0.055
\]

the error for \( w_2, v_2 \) is

\[
\frac{\log 893 - \log 892}{\log 1500 - \log 400} \approx 0.00084
\]

and for \( w_3, v_5 \) is

\[
\frac{\log 1415 - \log 1421}{\log 1500 - \log 400} \approx -0.0032
\]
28.2. PROBABILITY FOUNDATIONS

The above results give the minimal radius of the intervals for a hit to occur, the corresponding interval is twice this value. The following table shows the converted values and the computed probabilities

\[
\begin{array}{ccc}
  n & k & \epsilon & P_{k,n,\epsilon} \\
  5 & 0 & 0 & 1 \\
  5 & 1 & 0.0017 & 0.0084 \\
  5 & 2 & 0.0064 & 0.00080 \\
  5 & 3 & 0.11 & 0.056 \\
\end{array}
\]

Surprisingly, even though \(w_2\) is very close to \(v_2\), the event which considers both \(w_2, v_2\) and \(w_3, v_5\) at a greater distance, is 10 times more rare than the match of \(w_2, v_2\) alone.

We have now all the ingredients for our algorithm. For each protein in the database we compute its digestion in, say, \(n\) fragments, and the weights of these fragments. For each searched weight \(w_i\) we find the minimum distance to one of the database weights \(v_j\) and set \(d_i = 2|\log w_i - \log v_j|/(\log w_{\text{max}} - \log w_{\text{min}})\). Now we order the distances in increasing values \(d_1 \leq d_2 \leq d_3 \leq ...\). For distance \(d_i\), \(i\) weights are within distance \(d_i\) of database weights. For each \(d_i\) we compute \(P_{i,n,d_i}\). The best match is considered the \(i\) for which \(P_{i,n,d_i}\) is minimal. This probability identifies the database sequence. Finally we keep track of the \(m\) lowest probabilities (\(m\) usually between 5 and 20).

This function will be called \texttt{SearchMassDb} and receives as parameters a set of weights, a function which does the digestion, as shown earlier with \texttt{DigestTrypsin}, and the value \(m\).

\begin{verbatim}
> SearchMassDb := proc( weights:array(real), Digest:procedure, m:posint )
>    description 'Search the database for similar mass profiles';
>    if not type(DB, database) then 
>      error('a protein database must be loaded with ReadDb') fi;
>    create the output array (with maximum probability entries)
\end{verbatim}
CHAPTER 28. RECOGNIZING PROTEINS BY WEIGHT

\[ \text{BestMatches} := \text{CreateArray}(1..m, [1]); \]

sort the input weights and compute weight bounds

\[ \text{w} := \text{sort}(\text{weights}); \]
\[ \text{lw} := \text{length}(\text{w}); \]
\[ \text{wmax} := \text{w}[\text{lw}] \times 1.02; \]
\[ \text{wmin} := \text{w}[1] \times 0.98; \]
\[ \text{d} := \text{CreateArray}(1..\text{lw}); \]
\[ \text{IntLen} := \log(\text{wmax}) - \log(\text{wmin}); \]

for each sequence in the database do the digestion and compute the weights

\[ \text{for } e \text{ to } \text{DB[TotEntries]} \text{ do} \]
\[ \quad s := \text{String(Sequence(Entry(e)))}; \]
\[ \quad v := \text{Digest}(s[i]); \]
\[ \quad v := \text{sort}(\text{GetMolWeight}(v)); \]

compute the \( d_i \) values

\[ \text{for } j \text{ from } j \text{ to } \text{lv}-1 \text{ while } v[j+1] < w[i] \text{ do od}; \]
\[ \quad \text{if } j > 0 \text{ then} \]
\[ \quad \quad d[i] := 2 \times \text{abs}( \log(w[i]) - \log(v[j]) ) / \text{IntLen}; \]
\[ \quad \quad \text{else } d[i] := 1 \text{ fi}; \]
\[ \quad \text{if } j < \text{lv} \text{ then} \]
\[ \quad \quad d[i] := 2 \times \text{abs}( \log(w[i]) - \log(v[j+1]) ) / \text{IntLen}; \]
\[ \quad \quad \text{if } d[i] > d[i] \text{ then } d[i] := d[i] \text{ fi}; \]
\[ \quad \text{fi}; \]
\[ \quad \text{od}; \]

compute \( n \), the number of weights between \( w_{\text{max}} \) and \( w_{\text{min}} \)
28.3.  EXAMPLE AND PROBABILITY ESTIMATES

\begin{verbatim}
> n := 0;
> for j to lv do
>     if v[j] >= wmin and v[j] <= wmax then n := n+1 fi od;
> if n < 2 then next fi;

sort the interval widths and compute the most rare event

> d := sort(d);
> p := 1;
> for k to lw do
>     if d[k] > 0 then
>         pk := ( 1 - exp(-n*d[k]) ) ^ k;
>         if pk < p then kmin := k;  p := pk fi
>     fi
> od;

Instead of working with probabilities directly, we will work with the log-
arithm of these probabilities. To make these measures similar to the simi-
larity scores of matches, we will compute $-10 \log p$. This measure will now be
comparable to scores obtained from alignments using the standard Dayhoff
matrices.

> sim := -10 * log10(p);
> if sim > BestMatches[1,1] then

new smallest probability found, insert and reorder BestMatches

>         BestMatches[1] := [sim,e,n,kmin];
>         BestMatches := sort( BestMatches, x -> x[1] );
>     fi
> od;
> BestMatches
> end:
\end{verbatim}

28.3  Example and probability estimates

Suppose we are given the molecular weights:
> ws := [ 511.3, 563.1, 717.2, 743.2, 836.4, 842.5, 1014.4, 
   1169.4, 1387.5, 1509.7, 1524.0 ]:

from the results of digesting an unknown protein with trypsin. To compute 
the 5 most significant matches we run the command

> res := SearchMassDb( ws, DigestTrypsin, 5 )
> res := [[69.1127, 25358, 3, 5], [70.1624, 779, 28, 5], [70.3967, 8895, 5 
   , 6], [71.4526, 11384, 11, 4], [148.9464, 11424, 15, 9]]

Each element of res has 4 components: the similarity score, the entry num-
ber, the number of selected weights from the database (n) and the number 
of matched weights from the sample. We can print the above in a more 
readable format:

> for m in res do
  >    printf('Sim %.if,  %s,  %s\n\n', m[1],
  >        GetEntryInfo(Entry(m[2]),'DE')[2],
  >        GetEntryInfo(Entry(m[2]),'OS')[2]) od;
Sim 69.1,  PROPHAGE P2  OGR PROTEIN.,  ESCHERICHIA COLI.

Sim 70.2,  ANGIOTENSIN-CONVERTING ENZYME PRECURSOR, TESTIS-SPECIFIC 
(EC 3.4.15.1) (ACE-T) (DIPEPTIDYL CARBOXYPEPTIDASE I) (KININASE II).,
HOMO SAPIENS (HUMAN).

Sim 70.4,  DNA-BINDING PROTEIN II (HB) (HU).,  BACILLUS SUBTILIS,
AND BACILLUS GLOBIGII.

Sim 71.5,  COAGULATION FACTOR IX (EC 3.4.21.22) (CHRISTMAS FACTOR)
(FRAGMENT).,  OVIS ARIES (SHEEP).

Sim 148.9,  FATTY ACID-BINDING PROTEIN, LIVER (L-FABP)., 
Ginglymostoma cirratum (NURSE SHARK).

How can we determine if the above similarities are significant or not? 
Doing this directly is far too complicated, it would require to consider the
total number of sequences and the total number of choices of fragments to match. A Montecarlo method is much more appropriate in this case. We will run the search for various randomly generated sequences and tabulate their values. For this example we first generate as many random weights as we had in the sample, uniformly distributed in the same range

\begin{verbatim}
> rw := copy(ws):
> wmin := min(ws) * 0.98; wmax := max(ws) * 1.02;
> wmin := 501.0740
> wmax := 1554.4800

set counters to collect statistics

> BestSim := Stat();

and run 5 experiments

> to 5 do
> for i to length(rw) do rw[i] := wmin + Rand()*(wmax-wmin) od;
> res2 := SearchMassDb(rw,DigestTrypsin,1);
> UpdateStat(BestSim,res2[1,1]);
> od:

The results of this simulation are

> print(BestSim);

number of sample points=5
mean = 77.3 +- 5.5
variance = 39 +- 61
skewness=1.45134, excess=0.375441
minimum=72.0419, maximum=87.6283

Now we can state that for this database, similarity scores smaller than

> BestSim[Mean] + 1.96*sqrt(BestSim[ Variance]);
89.5599
are not very surprising or interesting. Our most significant match in the previous example is

\[ \frac{(\text{res}[5,1] - \text{BestSim}[\text{Mean}])}{\sqrt{\text{BestSim}[\text{Variance}]}} \]

11.4762

standard deviations away from the average, and that is very significant. Hence we could conclude that the weights 511.3, 563.1, 717.2, etc. come from the digestion of the fatty acid-binding liver protein of sharks or similar sequences.
Part III

The Appendices
Appendix A

Installation Instructions
Create a directory where you would like to store Darwin and move to this directory.

```plaintext
mkdir Darwin
cd Darwin
```

Next store the `Darwin` file you received by e-mail in this directory.

```plaintext
\begin{code}
  mv darwin.mach.tar
\end{code}
```

Here `\{tt mach\}` is the machine architecture you requested (e.g. `\{tt hp\}` is Hewlett-Packard, `\{tt xap\}` is Digital Alpha, `\{tt solaris\}` is Solaris).

Unpack `Darwin` with the Unix `\{tt tar\}` command:

```plaintext
\begin{code}
  tar xpf darwin.mach.tar
\end{code}
```

Now you are ready to execute `Darwin`. If you run

```plaintext
\begin{code}
  darwin.mach <samp.session
\end{code}
```

you should see the results of a small sample session with `Darwin`.

To run the command from another place use:

```plaintext
\begin{code}
  <absolute-path>/darwin.mach -l <absolute-path>/lib
\end{code}
```
in order for \texttt{Darwin} to correctly find the library.
You may want to make the above line an alias or a command
in your execution paths. Note that \texttt{Darwin} must find the
file \{\texttt{lib darwininit}\} (short for ‘‘darwin initialization’’)
on startup. This file is located in the \{\texttt{lib} \} (library)
directory in your \{\texttt{Darwin} \} directory.

There are three special flags which can be passed when starting a
\texttt{Darwin}-session.

\begin{verbatim}
> darwin -l library/directory
\end{verbatim}

<table>
<thead>
<tr>
<th>Flag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-l</td>
<td>Specifies the location of the Darwin library</td>
</tr>
<tr>
<td>-s</td>
<td>Specifies that Darwin should be run in server mode.</td>
</tr>
<tr>
<td>-q</td>
<td>Specifies that Darwin should be run in quiet mode.</td>
</tr>
</tbody>
</table>

Table A.1: The flags for executing Darwin.
Appendix B

Naming Conventions
The following is a short document sketching the Darwin naming convention. We can group the different Darwin constructs into five categories:

1. built-in types
2. structured types
3. commands
4. built-in functions, and
5. library functions.

We give a short but reasonably precise set of rules for naming types, structures, routines etc. for each of these categories. This document is primarily meant for Darwin developers.

English dictionaries provide the legal hypenation pattern for a word, e.g.

\[ \text{ap\cdot prox\cdot i\cdot mate} \]

usually in bold face. This does not necessarily correspond to the syllables of the word (these are typically given in the international pronunciation) e.g. Oxford English Dictionary (OED).

We will use the syllables of a word to create abbreviations for names which are too long in Darwin. The convention is as follows:

\* When names are abbreviated in Darwin, we use the first syllable of a word according to the OED. If this abbreviation is either (1) too short for uniqueness, (2) unaesthetic or (3) extremely unprounceable, the second syllable of the word is added. Subsequent syllables are added until problems (1) - (3) disappear.

There are small number of computer and biologic abbreviations common to both literatures. These abbreviations do not follow the \* principle above but may be used throughout the system and the onus lies on the user’s shoulders to identify their meanings. In general, this list should be kept as small as possible. There is a delicate balance between the advantages of having short names in the system and the disadvantages of having too many abbreviations. These lists should be made available to the user through on-line help. In Darwin, they will be made available as follows:
> ? abbreviations;
> help abbreviations;
> ? abbr;

Where possible, both the abbreviated form and the full length version should be recognized by the system. For example, suppose we have the function \texttt{LoadDB} (load a database file). The synonym \texttt{LoadDatabase} should also be recognized.

We give these lists here:

\textit{Computer Science:}
\begin{itemize}
  \item DB for \texttt{database, eval for evaluated, IPC for Inter Processor Communication,}
\end{itemize}

\textit{Biology:}
\begin{itemize}
  \item DNA for deoxyribonucleic acid, RNA for ribonucleic acid, Nuc for Nucleid acid (either DNA or RNA), AA for amino acid, PAM or Pam for point accepted mutation, ...
\end{itemize}

\section{Built-in Types}

\textit{Rules:}

(1) All built-in types should have names consisting of only lower case letters.

(2) Only very common computer science names should be abbreviated. These abbreviations should be noted in the abbreviations list above. The only example of this is \texttt{uneval} (short for \texttt{unevaluated}).
B.2 Structured Types

Rules:

(1) Only the first letter of each word should be capitalised.

(2) Structured type names should be kept reasonably short. When abbreviations seem appropriate, they should take place according to the * rule above. The abbreviations should be included in the abbreviations list.

(3) Abbreviations used in selector names should correspond to abbreviations used throughout the system. It makes no sense to use the abbreviation Sim for Similarity throughout the Darwin system and then require users to select with Simil on DayMatrix structures.

(4) Nouns should be singular. Thus Entry not Entries.

(5) When the structure is used in tandem with a routine (like with stats/Update, DrawPlot/Plot, Grid/CreateGrid), the name for the structure should be comprised of the noun used in naming the associated routine followed by Struct. For example, change Plot to PlotStruct, change stats to StatStruct (Rule 4 and 5).

(6) Selector names should be case insensitive. Thus, the selector PamNumber for structured type DayMatrix should be changed to PAMNUMBER and pammumber.

(7) Selector names should reflect the type of data they return. If they return a simple type, they should have names formatted according to the naming conventions for simple types. If they return structured types, they should have names formed according to these rules. The only exception to these rules is that they must be case insensitive.
B.3  Commands

Rules:

(1) We follow computer science history for naming conventions as close as possible. This is already basically true for the language.

(2) We always use lower case letters. Thus, error is error and \texttt{RETURN} is \texttt{return}. Both of these functions act as commands in Darwin enough so to warrant them being classified under this rubric and not the section of \textit{built-in function}.

B.4  Built-In Functions

By \textit{built-in function} we mean any function that has the option \texttt{builtin} line. This includes the mathematical functions and many other low level commands for searching, sorting, storing, and input/output.

Rules:

(1) Mathematical functions are named according to Abramowitz and Stegun conventions \cite{Abramowitz-Stegun}.

(2) We stay with the conventions of the language C when the function is sufficiently similar to the C routine (again \texttt{printf}, \texttt{printf}, \texttt{scanf}).
(3) We stay with the conventions of the language Maple when the function is sufficiently similar to a function from that language (or an exact copy).

(4) We only abbreviate with words that are common in computer science languages such as DB for database. All abbreviations must be kept in the Abbreviations List made available to users through help abbreviations.

(5) If the routine has a common application in another field (such as the NBody function does in physics), this name can be chosen. It would be preferable to give it the more abstract mathematical name when such a name exists.

(6) If none of the above cases apply, we use the conventions of Library Functions below.

B.5 Library Functions

Library functions should be named according to the following convention.

Rules:

(1) The name should consist of at most five parts.

\[ \text{<domain>_<adverb><verb><adjective><noun}> \]

(2) The verb should reflect the action in a meaningful way ie. Draw, Load, Save, Print. For performing string searches, the verb should be Search. If we are aligning sequences, it should be Align. If we are loading information from a file, it should be Load. If we are creating a graph-
ics file, it should be Draw. If we require a generic verb (such as Create, Build, Do, Compute, Make, we recommend Make be chosen. For example, ReadRawFile loads a “raw file”. This should be changed to LoadRawFile. The function DrawHistogram creates a histogram. This should be changed to MakeHistogram.

(3) The noun will typically be the object if you were to say the sentence completely. It will typically be a type or structured type. If the routine works on a particular type, then this type should be placed in the name as the noun. For example, ReadDb should be changed to LoadDatabase.

(4) A noun should be chosen that represents the generic object and is mathematical in nature i.e. you would not choose DrawLarson but instead DrawTree.

(5) The adjective should only be included when its absence does not distinguish between the objective of two or more routines. There are currently two routines in Darwin to create phylogenetic trees: DarwUnrootedTree (un-rooted) and DrawTree (rooted). We would choose the most generic mathematical name (Rule 4). This would lead to the choices DrawRootedTree and DrawUnrootedTree.

(6) The first and only the first letter of each word should be capitalised unless it is part of an abbreviation common in the biology/biochemistry literature i.e. DNA, RNA, PAM and maybe AA. See list of abbreviations below.

(7) The adverb indicates a qualified action. For example, currently the function ApproxTextSearch looks for an approximate match of a string against a body of text. This should be changed to ApproximateSearchString (Approximate is the adverb, Search is the verb, String is the object type.)
(8) The **domain** is a special identifier used to indicate that the routine that follows (ie. the `<adverb><verb><adjective><noun>`) applies to a special type of object. There are only two special domains in Darwin: **Inter Processor Communication** abbreviated to **IPC** and **Nuclear Peptide** abbreviated to **NucPep**.

The **domain** is essentially only a short form for the name. The information could be encoded as **adjectives** and **adverbs** in the name of the function but this leads to very long names. As a concrete example, the routines **GlobalAlign, LocalAlign, LocalAlignBestPam** etc. are used to match amino acid sequences with amino acid sequences. Most of the routines have analogs in the the nucleotide/peptide matching arena. We would rename **LocalAlignBestPam** to **AlignBestPamMatch** according to the new naming conventions but when we enter the nucleotide/peptide setting this becomes **AlignNucPepBestPamMatch** which is a little too long (well over twenty letters) and rather cumbersome. Instead, we suggest **NucPep.AlignPamMatch** for various reasons. Now the same function name is retained except prefixed by the new domain. (It is still too long but this requires that **ShakeAlignBestPamMatch** be abbreviated, possibly **ShakeAlignPamMatch**.

The same problems arise with the **IPC** protocols.

(9) Abbreviations should be avoided. When function names are too long, the adverb and adjective should be the first to be abbreviated. All abbreviations should follow the * rule above.

(10) Underscore characters should be avoided except to separate `<domain>` from the rest of the name. Of course, underscore characters are need for polymorphism but this poses no problem with our conventions.

(11) Nouns should be singular.
(12) Functions which perform “conversion” require a bit of extra attention. In the worst case, we might require the following extension to the grammar for names:

<adverb><verb><adjective><noun>To<adverb><verb><adjective><noun>

For example, the *Strings* function takes an offset from DB[*string*] and returns the associated sequence from the database. First we would apply rule 7 to change *Strings* to *String*. In the full form, this would become *OffsetTo*String. However, the type specification in the *Strings* function is *offset : integer*, so reasonably intelligent readers will immediately know that *Strings* takes an *Offset* and returns a *String*. Here we may leave out the *OffsetTo* entirely. When any ambiguity arises, we suggest the above *<obj>*To*<obj>* syntax.

## B.6 Name Conversion List

The following list contains the old Darwin 1.6 (and earlier) name and the new Darwin 2.x name.

**From To**

- AAAP AAAToInt
- ACS AC
- AP AToInt
- AaCount GetAaCount
- AaFrequency GetAaFrequency
- AddGF AddGrid
- AlignGaps AdjustGaps
- AlignedIntrons GetIntrons
- AlignedPeptide GetPeptides
AllMatches GetAllMatches
AminoP AminoToInt
ApprTextSearch ApproxSearchString
BBBC NucToCode
BBBP NucToInt
BP NToInt
BBBP NuctoInt
BackDynProgr BackDynProg
BaseP NucleicToInt
BestPamMatch GetPam
BestPamShake LocalAlignBestPam
BestStringMatch SearchString
BisectTree DrawBisectTree
CBBB CodeToNuc
CleanMSA RemoveGaps
CT_Species AddSpecies
CloseGF CloseGrid
ColorMap GetColorMap
ColorTree CreateColoredTree
Complement GetComplement
CompressGF CompressGrid
ConvertSP SpToDarwin
ConvertToDF DbToDarwin
CreateDB CreateAcGrid
CreateGF CreateGrid
DF DB
DMDMS CreateDayMatrices
DnaFile database
DNAPepDayhoffM ApproxDnaDayMatrix
Dayhoff CreateOrigDayMatrix
DayhoffM CreateDayMatrix
DelFixed FixedDel
DelIncr IncDel
DigestSeqs DigestSeq
Distribution DrawDistribution
DnaFile database
DotPlot DarwDotplot
DynProgr DynProg
ERROR error
EndOverlayPlot StopOverlayPlot
Entries Entry
Entropy FindEntropy
EntryInfo GetEntryInfo
EntryNumber GetEntryNumber
EquadTree DrawEqualRadialTree
ExponFit ExpFit
ExpFit2 BoundedExpFit
ExtCallFrame CreateCProgram
FlushGF FlushGrid
FragSearch SearchFrag
GetBetween GetLcaSubtree
GetIndex FindTreeFitIndex
GetLabels GetTreeLabels
GetMatchSequence GetAlignSequences
GetTreeLength TotalTreeDistance
GetPath GetPathDistance
GridFile grid
Histogram DrawHistogram
IDS ID
IPCconnect ConnectTcp
IPCdisconnect DisconnectTcp
IPCread ReadTcp
IPCReceive ReceiveTcp
IPCsend SendTcp
IPCReceiveDATA ReceiveDataTcp
IPCSendDATA SendDataTcp
IndexSP CreateSpGrid
LabelTree ChangeLeafLabels
LarsonTree DarwUnrootedTree
LeafNames GetLeafLabels
LinRegr LinearRegression
LinearIntron CreateLinearIntron
LinearIntron0 CreateLinearZeroIntron
LinearIntron1 CreateLinearProbIntron
LoadFile ReadDb
LongestRep FindLongestRep
MAalignment MultiAlign
MachineUsage GetMachineUsage
MapGF MapGrid
MassDyn DynProgMass
MassDynAll DynProgMassDb
MassProfile SearchMassDb
Maximize MaximizeFunc
MinSqTree MinSquareTree
Minimize MinimizeFunc
Minimize2D Minimize2DFunc
Minimizex DiscouMinimize
MolWeight GetMolWeight
MostFrequent GetMostFrequentGrams
MoveGap MoveGap
MultAlign CreateMultiAlign
NewArray CreateArray
NewString CreateString
NextGF GetNextGrid
NPAlignMatch AlignNucPepMatch
NPAllMatches GetAllNucPepMatches
NPBackDynProgr NucPepBackDynProg
NBPamMatch FindNucPepPam
NPamShakeBestLocalAlignNucPep
NPDynProgr NucPepDynProg
NPMatch NucPepMatch
NPMultiAllMatches ParallelAllNucPepMatches
NPOneAllMatch AlignNucPepAll
NPRefine GlobalNucPepAlign
NPRefineShake LocalAlignNucPep
NPRegions NucPepRegions
NPStringMatch DynProgNucPepString
Offsets Offset
OneAllMatch AlignOneAll
OpenGF OpenGrid
OrderedSearch SearchOrderedArray
PA IntToA
PAAA IntToAAA
PAmino IntToAmino
PB IntToN
PBPP IntToNNN
PBase IntToNucleic
PIdentToPam PAMToPam
PamToPI PamToPIdent
ParExec ParExecute
ParExec2 ParExecuteSmall
ParTest ParExecuteTest
PatEntries PatEntry
PepPepSearch SearchPepAll
PhyloTree CreatePhyloTree
PickTree FindLabeledSubtree
PlotPam DrawSimPam
PlotOptions Plot
PosInfo GetPosition
PositionDF GetOffset
PrintSeqsInTree PrintTreeSeq
ProbDynProgr ProbDynProg
ProfileEnter EnterProfile
ProfileExit ExitProfile
QueryAll AllQueryGrid
QueryGF QueryGrid
RETURN return
RandTree CreateRandMultAlign
RandomPermut CreateRandPermutation
RandomSeq CreateRandSeq
RandomTrees CreateRandTrees
Refine GlobalAlign
RefineLog LogDelLocalRefine
RefineShake LocalAlign
SameTree IdenticalTrees
Scale DayMatrixScale
SearchDF SearchDb
SearchPepDF SearchSeqTagDb
SearchText CaseSearchString
SelfMatch GetSelfAlign
Sequences Sequence
ShortestPath ConShortestPath
ShortestPath2 ShortestPath
Smooth SmoothData
SplatTree DrawSplatTree
SprintMatch DynProgStrings
System TimedCallSystem
StackedBar DrawStackedBar
Stats Stat
string name
Strings String
SummarizeTree CollapseNodes
TSP ComputeTSP
TSP3 ComputeCubicTSP
TSP4 ComputeQuadraticTSP
TreeOrder FindCircularOrder
TrulyRandom SetRandSeed
UUUP GenCodeToInt
UnCompressGF UncompressGrid
UnLabelTree UnlabelLeaves
UnionStats UnionStat
View ViewPlot
Violations FindSpeciesViolations
WriteMSA WriteMsa
anitparallel GetAntiparallel
appendto AppendFile
clearw ClearStat
currentOfs CurrentOff
dpuTime DpuTime
eigenvalues Eigenvalues
externcall CallExternal
findkey FindKey
function uneval
gausselim GaussElim
gcm GenCode
kGramRegion GramRegion
kGramRegionScore GetGramRegionScore
kGramSite GramSite
kGramSiteScore GetGramSiteScore
load ReadLibrary
numeric real
plot DrawPlot
srand SetRand
rand Rand
read ReadProgram
readBRK ReadBrk
readDSSP ReadDssp
readfile ReadRawFile
readpipelines OpenPipe
readstat ReadLine
readstatAt ReadOffsetLine
searchtext SearchString
specfunc specuneval
srand SetSeed
string name
system CallSystem
text string
update UpdateStat
writeto WriteFile
Predict PredictSecStruct
B.6. NAME CONVERSION LIST

NDF NucDB
Simil Sim
string symbols
Text string
PDF PepDB
MaxSimil MaxSim
MinSimil MinSim
GetPam FindBestPam
Entries_print Entry_print
Entries_Offset Entry_Offset
Entries_Strings Entry_String
Entries_Sequences Entry_Sequence
Entries_Match Entry_Match
Entries_select Entry_select
Offsets_Entries Offset_Entry
Offsets_PatEntries Offset_PatEntry
Offsets_print Offset_print
Offsets_Sequences Offset_Sequence
Offsets_Strings Offset_String
Offsets_Match Offset_Match
PatEntries_Entries PatEntry_Entry
PatEntries_Offset PatEntry_Offset
PatEntries_Strings PatEntry_String
PatEntries_print PatEntry_print
PatEntries_Sequences PatEntry_Sequence
PatEntries_Match PatEntry_Match
Sequences_Entries Sequence_Entry
Sequences_Offset Sequence_Offset
Sequences_PatEntries Sequence_PatEntry
Sequences_print Sequence_print
Sequences.Strings Sequence.String
Sequences.Match Sequence.Match
Match.Offsets Match.Offset
Match.Entries Match.Entry
Match.Strings Match.String
Entries.IDS Entry.ID
Match.IDS Match.ID
Sequences.IDS Sequence_ID
Offsets.IDS Offset.ID
PatEntries.IDS PatEntry.ID
IDS.Offsets ID.Offsets
IDS.Entries ID.Entry
IDS.Match ID.Match
IDS.Sequences ID.Sequence
Entries.ACS Entry.ACS
IDS.Entries ID.Entry
IDS.Match ID.Match
IDS.Sequences ID.Sequence
IDS.Strings ID.String
Match.ACS Match.AC
Entries.ACS Entry.ACS
Offsets.ACS Offset.AC
Sequences.ACS Sequence.AC
PatEntries.ACS PatEntry.AC
ACS.Offsets AC.Offset
ACS.Entries AC.Entry
ACS.Match AC.Match
ACS.Sequences AC.Sequence
ACS.Strings AC.String
IDS.ACS ID.AC
B.7 Scripts for Updating Code

```bash
!/bin/csh
grep "\<$1\>" hallett/Darwin/lib/* hallett/Darwin/* hallett/Book/* | more
echo ":s/\<$1\>/\$2/g"
echo ":/\<$1\>"
vi 'grep -l "\<$1\>" hallett/Darwin/lib/* hallett/Darwin/* hallett/Book/*'
```
Bibliography


[26] UNIX System Labs, Inc. Unix operating system.