
Perl for Bioinformatics

Arun Jagota

Department of Computer Science, University of California, Santa Cruz

© Arun Jagota, 2001

All rights reserved. This book is protected by Copyright ©. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, electronic, mechanical, photocopying, recording, or otherwise, without prior permission of the copyright owner.

Preface

Informatics in the genomic setting involves the storage, search, and manipulation of textual data such as DNA and protein sequences and related matter such as structural or functional annotations, authors, cross-links to other sequences or structures, and more. (See the **Genbank**, **EMBL**, and **PDB** databases to get a better idea.) While some of the search and manipulation needs of a user may be met by generic web interfaces, often they won't be. Some needs may not have been anticipated when the web interface was constructed, others perhaps were too complex to support for use over the web.

Perl is an ideal language for such needs. With **Perl** one will sometimes be able to solve a data search or manipulation problem in a matter of hours if not minutes. (Solving the same problem in **C/C++/Java** could take much, much longer.) Here are some particular problems in this setting for which **Perl** is a great fit.

- Searching sequence databases with regular expression patterns.
- Parsing entries in databases (e.g., reading a **Genbank** entry of a gene and extracting its exons).
- Converting database entries from one format to another (e.g., converting from **Genbank** to **EMBL** format).
- Using standard sequence analysis tools written in **Perl**. (See the various Perl packages supported by the **BioPerl** project.)

This short book introduces **Perl** to the bio or computer scientist interested in or working in bioinformatics. Chapter 1 covers data types. Chapter 2 covers control structures. Chapter 3 covers input and output. Chapter 4 covers regular expressions. Chapter 5 covers handy functions on strings. All these chapters contain illustrative examples from bioinformatics. These chapters cover only those features of **Perl** that are particularly important to know in the context of search and manipulation of biomolecular data. In particular, **Unix**-specific features such as those involving **Unix** file, directory and process management are omitted. Chapter 6 touches on subroutines. Chapter 7 presents several Perl scripts for various common bioinformatics tasks. Chapter 8 covers the **BioPerl** project.

There is a common myth that **Perl** is a **Unix**-only tool and people on **Windows** have no use for it. Not so! A pretty nice version of **Perl** comes for the **Windows** platform, and free! (Indeed my students and I did bioinformatics programming assignments in **Perl** in a **Windows** environment and this worked well.)

If you need more convincing, read

L. Stein. How Perl saved the Human Genome Project. *The Perl Journal*. The full text is online at http://bio.perl.org/GetStarted/tpj_ls_bio.html

Prerequisites

Prior familiarity with **C**, **C++**, or **Java** will definitely help the reader.

1 Data Types

In any programming language one minimally needs the ability to store and operate on *numbers* and on *arrays of numbers* (vectors, matrices). The former are called *scalar data*, while the latter *composite data*. Most other languages also support a separate notion of a *character* (which permits one to store and operate on single characters such as `a`, `b`, etc). A *string* of characters is then supported in these languages by forming an array of characters. Thus in these languages a string is composite data.

Perl, like these languages, supports numbers and arrays of numbers and the usual operations on them. Unlike most other languages, however, **Perl** also supports a string as a *scalar data type*, together with built-in operations on strings. One reason for this choice is that **Perl** (unlike languages such as **Fortran/C/C++**) is a true string processing language with a lot of built-in support for strings and operations on them. In order to realize this effectively, it makes sense to treat strings as fundamental data objects, i.e. scalar data.

Scalar Types

A *scalar data type* is one whose values are not decomposable into parts. An integer is a scalar data type, while an array of integers is not. **Perl** has just two scalar types: *number* and *string*. A scalar variable has the form **\$name**.

Numbers and Operators

Numbers in **Perl** are **C**-like. All numbers are represented internally as real (double-precision float). Here are some numeric constants:

1 1.5 3.21 -6.5e24 1.2E3

Perl supports **C**-like numeric operators. Here are the main ones.

Operator type	Operators	Examples
Arithmetic	+, -, *, /, %, **, ++, --	1+2, 10/3, 3**8, 8%3, \$x++
Comparison	<, <=, ==, >=, >, !=	\$X < \$Y, \$X == \$Y
Logical	&&, , !	(\$x<\$y)&& (\$y<\$z)
Assignment	=, +=	\$a=5; \$b+= \$a;

4 Regular Expressions

For manipulating string data such as DNA and protein sequences, what makes **Perl** a hands-down winner over every other programming language is its unique and powerful support for *regular expressions*.

A regular expression is a *pattern* to be matched against a string. Regular expressions are especially useful for searching sequence databases for certain patterns. Here are some examples. In each of them the "database" is assumed to be a file containing DNA sequences, one per line, and the query pattern is hard-coded in the example.

```
while (<>) {
  if (/ATG/)
    {print $_;}
}
```

```
while (<>) {
  if (/([CT]AG)/)
    {print $_;}
}
```

```
while (<>) {
  if (/GT.*AG/)
    {print $_;}
}
```

Print all lines containing ATG in an attempt to find all lines containing the start codon.

Prints all lines containing YAG in an attempt to find all lines containing acceptor site boundaries.

Prints all lines containing ...GT...AG.... in an attempt to find all lines containing introns.

Many false positives could result from running these programs. These might be cut down somewhat by using the known wider consensus sequences for the donor and acceptor sites. Specifically,

```
while (<>) {
  if (/([CT]AGGT[AG]AGT.*[CT]{11}[ACGT][CT]AG[GA])/)
    { print $_; }
}
```

[CT]AGGT[AG]AGT is the donor site consensus sequence, with the **GT** being the first two bases in the intron. **[CT]{11}[ACGT][CT]AG[GA]** is the acceptor site consensus sequence, with the **AG** being the last two bases in the intron. **.*** matches the rest of the intron.

It is tempting to extend this to try to recognize entire genes! Would it not be great if gene finding were that simple? Unfortunately, it is not. On the other hand, a simple **Perl** script could be put together to screen out regions of the genome that for sure do not contain a gene.

Rather than have the regular expression applied by default to **\$_**, one can have it applied to any string via the **=~** operator. Here are some examples.

```
if ($sequence =~ /CGC/) {...}
```

```
if (<STDIN> =~ /GT.*AG/)
  {...}
```
