初学者のためのPerlによる
バイオプログラミング入門

松井 求・斎藤 輪太郎 著
冨田 勝 監修

はじめに

ポストゲノムの時代を迎え分子生物学の実験データが大変な勢いで蓄積されている．これら膨大な量の公共データと自分が出した実験データを比較，解析し，有用な情報を抽出するためにはコンピュータを用いることが必要不可欠である．すでに現在までに様々な目的に応じた有用なソフトウェアが開発され，多くは無料で配布されている．既存のプロトコルにそった解析で間に合うのであればこれらのソフトウェアを用いる事で十分な結果が見込まれるであろう．しかし独自の仮説に基づいた解析を行う場合，あるいは大量のデータを相手にする，複数のソフトウェアを組み合わせて用いるといった場合はそれだけでは不十分であろう．柔軟にソフトウェアを組み合わせてフローを自動化したり，自前で計算をこなすような独自のプログラムが必要となってくる．このような用途に関してプログラミング言語に求められるのは外部のソフトウェアを自由に操作する事ができること，そして自前で計算やテキスト処理をこなせる能力をもつことである．さらには初心者でも簡単に習得でき，他の広い分野にも応用できる汎用的な言語であればさらによい．

このような要望にもっとも適している言語は Perl であろう．Perl はテキスト処理や CGI など，現在ある大量の Web コンテンツを維持する為に無くてはならない言語であるが，そのまま”テキストデータ”であるゲノム情報，あるいは生化学データなどを解析することが非常に得意な言語である．しかしながら，他言語の経験のあるプログラマや情報学者を対象にしたプログラミングの解説書は多数出版されているものの，プログラミングの経験のない分子生物学者にとってはかなり高度な内容となってしまっている．忙しい研究の合間に，あるいは実習の合間にプログラミングを習得していくのには必要最小限の練習問題を順に解いていき，身につけていくのがよい方法であると思われる．

本書は慶應義塾大学湘南藤沢キャンパス (SFC) で行なわれている実習形式の講義，”ゲノム解析プログラミング実習”で用いられている講義資料を元として加筆，修正したものである．講義の対象は主にプログラミング経験のない学部 1,2 年生であり，従って本書の対象はプログラミングの知識を前提としない分子生物学者および分子生物学を専攻とする学生となる．第 1,2 部で Perl プログラミングの基礎をひととおり網羅した．各確認事項ごとに練習問題，及び解答をつけたので一通ずつ解いていくことで効率よく必要事項を学ぶ事ができると考える．3 部でゲノム配列解析，遺伝子発現データの解析，生化学データの解析の実習問題をあげた．この問題を通して実用的なプログラミング能力を習得していただきい．また本書に掲載したプログラム，およびサンプルデータはすべて以下の Web ページに載せた．参考にしていただければと思う．http://www.../

最後に本書を執筆するにあたってお世話になった方々にお礼申し上げたい．まず鈴木治夫氏，永安悟史氏にはお忙しい中本書執筆にご協力いただいた．そして慶應義塾大学先端生命研究室の皆さん，特に岡田祐輝氏には多くの貴重なご指摘，ご意見をいただいた．この場を借りて感謝申し上げる．また本書の出版にあたってピアソン・エデュケーションの藤村行俊氏には大変お世話になった．深く感謝を申し上げる．

2006 年 12 月 24 日 松井求

目 次

第1章　Perlプログラミングを始めるまえに

1　準備するもの

必要なものはPCだけです．Perlプログラミングは Linux, Mac, Windows といったほとんどすべてのOS[1]で可能です．本書では Linux での作業を想定して述べていきますが，他の OS でもほとんど同様にできるはずです．例えば Linux は始めから Perl が含まれており，それはプロンプトを開いてすぐに確認することができます．Mac OSX にも同様にはじめから Perl が含まれていてやはり Terminal 上で全く同様に行うことができます．Windows でプログラミングを行う場合には若干のツールのインストールが必要かもしれません．ActivePerl あるいは Cygwin 等を使うと便利でしょう．詳しくはマシン管理者に問い合わせてみてください．

本書を学習するにあたっては即実用に役立つようにとの目的からあえて詳しい説明を省いた箇所があります．より深く学習されたい方はぜひリファレンスを手元に用意するとよいと思います．あるいはそういった資料はネット上に充実しています．参考にしてみてください．

2　login, logout

解説 Linux ではコマンドによってコンピュータに命令を与えます．以下に最低限知っておくべきコマンドをあげます．どれも重要なコマンドですのでひとつづつ動作を確認してみましょう．

表 1.1: 代表的なコマンド

コマンド	意味	例文
ls	ディレクトリの中身を表示	ls ディレクトリ
cd	カレントディレクトリの移動	cd ディレクトリ
cp	ファイルのコピー	cp 元ファイル 新規ファイル
mkdir	ディレクトリの新規作成	mkdir 新規ディレクトリ
less	ファイルの中身を表示	less ファイル
mv	ファイル，ディレクトリの移動	mv 元ファイル 新規ファイル
rm	ファイルの削除	rm ファイル
emacs	emacs(エディタ) の起動	emacs -nw ファイル
exit	ログアウト	exit
chmod	パーミッション変更	chmod 755 ファイル
perl	Perl インタプリタの起動	perl HelloWolrd.pl

問題 1(★☆☆☆☆) ホームディレクトリの下に perl_exer というディレクトリを作成してください．以後の作業はこのディレクトリで行いましょう．

[1]Perl はプラットフォームに非依存，かつフリーな言語です．以下に Perl の環境を用意する上で有用な文献，Web ページをあげます．

3 Perl とは

解説 コンピュータは指令に従って計算を行う機械です．使用者はなんらかの方法でコンピュータに指令を出さなくてはいけません．そのひとつの方法としてコンピュータに対する指令を順番にファイルに書き込みコンピュータに手渡すという方法があります．この場合コンピュータはそのファイルを読み込み，書いてある指令を順にこなしていくわけですが，指令の書き方/文法は様々なものがあります．その文法のひとつが Perl 言語です．

問題 2(★☆☆☆☆) perl というコマンドを打ってみましょう．すると入力待ち状態になります．そこで以下のように記述し，計算結果を表示させてみてください (%はプロンプト)．8 と出力されれば正解です．

```
% perl
print 5+3;
_ _END_ _
```

4 Perl の書き方

解説 今見たように Perl は適宜起動して作業を行わせることができますが，スクリプトとして保存して繰り返し使用することも可能です．このような Perl スクリプトを書く上で必要なのがエディタです．Linux で代表的なエディタには Emacs と vi がありますが，ここでは Emacs を使います[2]．以下に基本的なコマンドをあげます．特に C+z ⇔ fg は効率のいいプログラミングに必須でしょう．

表 1.2: Emacs の主なコマンド

シェル内での移動	
emacs -nw	同ウィンドウ内で起動
emacs &	別のウィンドウで起動
C-z	Emacs の中断，シェルに戻る
fg	シェルから中断した Emacs に戻る
保存	
C-x C-s	ファイルへの保存
C-x C-w	別ファイル名で保存
編集	
C-k	カーソル上の文字から行末までの削除
C-w	リージョンの削除
M-w	リージョンのコピー
C-y	キルバッファの内容の取り出し
C-x u	操作の取り消し
C-x z	操作の繰り返し
C-s	カーソル以降の文字列検索
C-r	カーソル以前の文字列検索

[2]Windows ではメモ帳，xyzzy，秀丸等があります．使いやすいものを用いてください

問題 3(★☆☆☆☆) まず以下のようにして Emacs を起動してください.

```
% emacs -nw nucleotide.txt
```

そして以下の内容を書き込み，保存しましょう．すると nucleotide.txt というファイルができているはずです．less コマンドで観覧してみてください.

```
adenine
thymine
cytosine
guanine
```

5 Perl の動かし方

解説 Perl プログラムを実行するには perl インタプリタにファイル (perl スクリプト) を渡し，実行してもらう必要があります．まず Emacs を起動して以下の内容を書き込みましょう.

```
% emacs -nw test.pl  #emacs の起動，perl スクリプトの編集
```

書き込む内容

```
print "Hello, World!\n";
```

保存してシェルに戻ったら以下のコマンドを打ち込む.

```
% perl test.pl
```

これで test.pl という perl スクリプトが実行されます．結果を確認してみてください．ところで test.pl の中身は以降勉強していくとして，一般の perl スクリプトの先頭には次のように書いてある場合があります.

```
#!/usr/bin/perl

プログラム...
```

これはこのスクリプトを読み込んで実行するものとして perl インタプリタを指定している，という文です．ためしに

```
% which perl
```

というコマンドを打ってみましょう．/usr/bin/perl などという応答が帰ってくるはずです．これは/usr/bin というディレクトリの下に perl インタプリタが置いてあることを意味しています．つまり`#!/usr/bin/perl` と書いてあるスクリプトはそれに続く命令文を/usr/bin/perl に読み込んでもらうということになるのです.

さて，この機構を利用して実行するには先ほどとは違い以下のようにします.

```
% emacs test.pl -nw #編集
(% chmod 755 test.pl #実行権を与える)
% ./test.pl #実行
```

perl というコマンドがいらなくなったのは分かりましたか？実際にはどちらの方法でスクリプトを実行させてもかまいません．例えばインタプリタが置いてある場所はシステムごとに異なるので`#!`の行を省略するという考え方もありますし，それとの中間的な方法として

`#!/usr/bin/env perl`

と書く人もいます[3].

[3]chmod はパーミッションを変更するコマンドです.

第2章　Perl入門

1　標準出力

解説 print 文で文字列または数値を表示することができます．\n は改行を表します[1]．先ほど取り上げた以下の例は”Hello, World!”と画面上に出力させる命令文です．

```
print “ Hello, world!", "\n ”;
```

問題 1(★☆☆☆☆) 123 x 456 の答えを，”123 x 456 = ”という文字列の後に表示しましょう．

ヒント 2 * 3 で 2×3 が計算できます．また,(カンマ) で区切られた文字列は連続して出力されます．

2　スカラー変数

解説 "$名前 ”でスカラー変数になります．文字列と数値のどちらも同様にスカラー変数に代入することができます．変数に値を代入する演算子は"="で，これは右側の値を左側の変数に代入するという意味になります．

```
$nucleotide = "uracil";
print $nucleotide, “ \n ”;
```

問題 2(★☆☆☆☆) 変数$x に 86400，$y に 365 を代入し，$z = $x * $y として，$z の値を表示しましょう．

解説 変数の中の値を増やす方法がいくつかあります．$x = $x + 10 で$x の値が 10 増えます．また$x ++ は$x = $x + 1 と等価です．

```
$x = 3;
$x = $x + 5;
print $x, “ \n ”;
```

問題 3(★☆☆☆☆) $x に 10 を，$y に 5 を代入しましょう．そして$x に$x+$y を代入して$x の中身を出力しましょう．

3　配列変数

解説 ”@名前 ”で配列を扱うことができます．@array で array という名前の配列を扱うことができます．$array[3] で@array の中の 4 番目の要素を取り出すことができます．以下の例では cytosine が出力されるはずです[2]．

```
@nucleotide = (“ adenine ”, “ thymine ”, "cytosine ”, “ guanine ”);
print $nucleotide[2], “ \n ”;
```

[1]セミコロンに注意！

[2]Perl の中では 1 番目のものは 0 番目と認識されます．同様に 2,3,... 番目のものは 1,2,... となります．

問題 4(★☆☆☆☆) ”Sunday ”,“ Monday ”,“ Tuesday ”, …,“ Saturday ”を@week の中に入れましょう．そして，配列の中の 3 番目の要素を出力しましょう．

4 split 関数

解説 split を使うと，文字列を区切り文字で区切って，配列に格納することができます．

```
$str = “ a,b,c,d,e ”;
@array = split(/,/, $str);
print $array[2];
```

問題 5(★★☆☆☆) 文字列 ”Sun-Mon-Tue-Wed-Thr-Fri-Sat ”を$week_str に格納し，それを split で曜日ごとに区切って@week_array に格納し 3 番目の要素を表示しましょう．

5 ループ

5.1 while 文

解説 while 文を使うと，( ) で囲まれた条件が満たされている間，{ }で囲まれた箇所が何度も実行されます．

```
$x = 1;
while($x < 5){
   print “ Hello!\n ”;
   print “ Variable is now $x\n ”;
   $x ++;
}
```

問題 6(★★☆☆☆) while 文を使って以下のような出力をしてみましょう．

```
There are 2 sheep.
There are 4 sheep.
There are 6 sheep.
There are 8 sheep.
There are 10 sheep.
There are 12 sheep.
```

問題 7(★★☆☆☆) 1, 2, 4, 8, 16, …と 2 のべき乗を 100000 を超えるまで出力してみましょう．

問題 8(★★☆☆☆) while 文を使って $3 \times 5 \times 7 \times 9 \times 11 \times 13 \times 15$ を計算しましょう．

5.2 for 文

解説 for 文を使うと，( ) で囲まれた条件が満たされている間，{ }で囲まれた箇所が繰り返し実行されます．ただし ( ) の中身は以下の書式に従う必要があります．

書式 for(初期化式; 条件式; 増分式){ 処理 }

式は空白でもかまわないが小カッコの中にはセミコロンが必ず 2 つ含まれなければならない．また初期化式などを複数書きたい場合はカンマで区切る．

```
for( my($i, $j)=(0,0); $i<=10; $i++, $j = $i*$i ){
   print “ $i\t$j\n ”;
}
```

またこれは以下のループと同等です．

```
my($i, $j) = (0,0);
while($i<=10){
    my $j = $i*$i;
    print “ $i\t$j\n ”;
    $i++;
}
```

問題 9(★☆☆☆☆) $1, 2, 3, \ldots, 99, 100$ の数字を降べきの順 (100, 99, . . . , 1) に出力しましょう．

5.3 foreach 文

解説 foreach 文を使うと配列の要素を順番に取り出し処理することができます．

書式 foreach $変数 (@配列){ 処理 }

$変数は省略することが可能です．省略した場合，配列の要素は$_に代入されます．また@配列の代わりにリストを使用することも可能です．以下はリストを応用した例です．例文中の 0..100 は (0,1,2,...,99,100) というリストを表します．

```
foreach my $i(0..100){
    my $j = $i*$i;
    print “ $i\t$j\n ”;
}
```

問題 10(★★★☆☆)1 日が月曜日の 4 月のカレンダーを出力しましょう．

```
1 月
2 火
. .
. .
. .
```

もしくは

```
月 火 水 木 金 土 日
1 2 3 4 5 6 7 8
9 ...
}
```

のどちらでもかまいません．最初の例の場合はあらかじめ月曜から日曜までの曜日のリスト (配列) を用意し，%演算子を使うと余りが計算できることを利用すると解決できます．例えば$R = 10%7 で $R に 3 が代入されます．つまり 4 月 10 日は水曜日ということが分かります[3]．

[3]演算子については後ほど詳しく解説します．

6 ハッシュ

6.1 静的なハッシュ作成

解説 ”%名前 ”でハッシュを扱うことができます．%hash で hash という名前のハッシュを扱うことができます．ハッシュは ”辞書 ”のようなものです．以下のようにして ”辞書 ”を作成し利用することが出来ます．”辞書 ”でいう見出しをキー (key)，説明を値 (value) と呼びます．

```
%nucleotide = (
    "a"   =>"adenine",
    "t"   =>"thymine",
    "c" =>"cytosine",
    "g" =>"guanine"
);
print "$nucleotide{c}\n";
```

問題 11(★☆☆☆☆) いくつかのアミノ酸の略称と正式名称を対応させたハッシュ%amino を作成しましょう．いくつかのアミノ酸について略称と正式名称を並べて出力しましょう．

6.2 動的なハッシュ作成

解説 上の例文では静的に%nucleotide を作りましたが，さらに以下のような操作によってエントリーを追加することが出来ます．

```
%nucleotide = (
    "a"   =>"adenine",
    "t"   =>"thymine",
    "c" =>"cytosine",
    "g" =>"guanine"
);
$nucleotide{u} = "uracil";
print “ $nucleotide{u}\n ”;
print “ $nucleotide{a}\n ”;
```

問題 12(★☆☆☆☆) 上の問題で作った%amino に後からエントリーを追加しましょう．

6.3 ハッシュのキーの取り出し (keys 関数，values 関数)

解説 keys 関数を使うとハッシュのキーをリスト化することが出来ます．foreach 文と組み合わせることでハッシュの中身を順番に処理することが出来ます．また似た関数に values があります．これはハッシュの value をリスト化します．

```
%nucleotide = (
    "a"   =>"adenine",
    "t"   =>"thymine",
    "c" =>"cytosine",
    "g" =>"guanine"
);
foreach my $key(keys %nucleotide){
    print “ $key\t$nucleotide{$key}\n ”;
}
```

問題 13(★☆☆☆☆) %amino に含まれるエントリーを全て出力しましょう.

6.4 ハッシュのキーの取り出し (each 関数)

解説 each 関数を使うとキーと値を一組ずつ取り出すことが出来ます. each 関数は while 文と組み合わせて使うことに注意.

```
%nucleotide = (
    "a"   =>"adenine",
    "t"   =>"thymine",
    "c" =>"cytosine",
    "g" =>"guanine"
);
while(($key, $value) = each %nucleotide){
    print “ $key\t$value\n ”;
}
```

問題 14(★☆☆☆☆) さきの keys 関数を使ったプログラムを each 関数を使ったプログラムに書き換えましょう

6.5 ハッシュを用いたカウント

解説 ハッシュを用いると各要素の数を効率的にカウントすることが出来ます. 対象とする要素数が多い場合, あるいは未知の場合有効です.

```
my $seq = ‘ aacgtgatgtcgtagtacgatgc ’;
my %count;
$count{$_}++ for split ‘ ’, $seq;
for(keys %count){
    print “ $_\t$count{$_}\n ”;
}
```

問題 15(★★☆☆☆) 上の例文を実行し, アルゴリズムを理解しましょう

7 ファイルの読み込み

解説 open でファイルをオープンした後，while(<FILE>) でファイルを一行ずつ全部読み込むことができます．読み込んだ行は$_に格納されます．

```
open(FILE, “ nucleotide.txt ”);
while(<FILE>){
   print $_;
}
close FILE;
```

問題 16(★☆☆☆☆) nucleotide.txt の最後の一行を表示するプログラムを書きましょう．

解説 chomp を使用すると，行末の改行を取り除くことができます．

```
open(FILE, “ nucleotide.txt ”);
while(<FILE>){
   chomp;
   print $_;
   print “ \n ”; # 行末の改行が取り除かれているので，ここで改行を行わなければ
   # ならない．
}
close FILE;
```

問題 17(★☆☆☆☆) nucleotide.txt の各行をコンマでつなげて表示するプログラムを書きましょう．

出力結果

```
adenine,thymine,cytosine,guanine
```

問題 18(★☆☆☆☆) 以下の文章を nucleotide2.txt として作成し，一行目と二行目を入れ替えるプログラムを書きましょう．各行の区切り文字はタブとします．

fruit.txt

```
a    adenine
t    thymine
c    cytosine
g    guanine
```

出力結果

```
adenine    a
thymine    t
cytosine   c
guanine    g
```

問題 19(★★★☆☆) さらに opendir と readdir を使うことでディレクトリに含まれるファイルをすべて処理することができます．今いるディレクトリを開き，含まれるファイルをすべて表示するようなプログラムを作成しましょう．

8 演算子

8.1 算術演算子

解説 perl には加減乗除等の算術演算子がいくつか用意されています．基本的な演算子は一通り理解しておきましょう．同時に ”+= ”といった短縮表現及び++の意味も確認しましょう．[4]

表 2.1: 算術・論理演算子

演算子	意味	書式
$a + b$	足し算	$a + b$
$a - b$	引き算	$a - b$
$a \times b$	かけ算	$a * b$
a/b	割り算	a/b
$a \div b$ の余	剰余	$a\%b$
a^b	累乗	$a * *b$
$a \cap b$	論理積 (かつ)	$a\&\&b$
$a \cup b$	論理和 (または)	$a\|\|b$
$\neg a$	否定	$!a$

書式 演算と代入：`$hoge = $foo + $bar;`
短絡表現：`$hoge += $foo; #($hoge = $hoge+$foo と同じ)`
オートインクリメント：`$hoge ++;`

```
$add    = 5+3; print $add,“ \n ”;
$take   = 5-3; print $take,“ \n ”;
$time   = 5*3; print $time,“ \n ”;
$divid  = 5/3; print $divid,“ \n ”;
$add   += 2; print $add,“ \n ”;
$take  -= 2; print $take,“ \n ”;
$times *= 2; print $times,“ \n ”;
$divid /= 2; print $divid,“ \n ”;
```

問題 20(★☆☆☆☆) 0,1,2,. . . ,10 の整数をランダムに 100 回出力しましょう．ただし以下の条件を満たしたプログラムにしてください．
a) 0 の場合，ZERO という文字列に置き換えて出力してください
b) 条件分岐は使わずに演算子を用いましょう

ヒント 1 0~10 の間のランダムな整数を得たい場合，`$number = int(rand(10));\index{rand}`とすると良いでしょう．

ヒント 2 `$hoge = $foo || $bar;` とすると，`$foo` の値が 0 だった場合，`$bar` が`$hoge` に代入されます．一方`$foo` の値が 0 以外であった場合は`$foo` が`$hoge` に代入されます．

[4] ++はオートインクリメントという演算子で変数の値を一つ増やす作用をします．同様に–はオートディクリメントといわれ，変数の値を一つ減らします．

8.2 文字列演算子

解説 数値演算子と同様に文字列を操作する演算子もいくつか用意されています．特に文字列結合演算子（．ドット）はマスターしておきましょう．

表 2.2: 文字列演算子

意味	書式
文字列をつなげる	a . b
文字列を繰り返す	a x 3

```
$seq  = ‘ hoge ’. ”\n ”; print $seq;
$seq  = ‘ hoge ’ x 3; print $seq;
$seq .=  ”\n ”; print $seq;
$seq x= 3; print $seq;
```

問題 21(★☆☆☆☆) a から z まで順につなげた 26 文字の文字列を作りましょう．そしてそれを 10 回繰り返した 260 文字の文字列を出力しましょう．

8.3 比較演算子

解説 数学における等号/不等号と同様に数値を比較し，その結果を返す比較演算子が用意されています．以下の例文を実行し演算子の働きを確かめましょう．

表 2.3: 比較演算子

数式	意味	書式
	数値比較	
$a = b$	等しい	a==b
$a \neq b$	等しくない	a!=b
$a < b$	小さい	$a < b$
$a > b$	大きい	$a > b$
$a \leq b$	以下	$a <= b$
$a \geq b$	以上	$a >= b$
	比較	$a <=> b$
	文字列比較	
	同一文字列	a eq b
	同一文字列ではない	a ne b
	文字列の比較	a cmp b

注意 ・ == と = の違いに気をつける
・ => と − > は比較演算子ではない．>= との書き間違えに注意
・ 数値比較演算子と文字列比較演算子の違いに注意．eq と==を混合しないように

```
$judge  = (5==3); print $judge;
$judge  = (‘ b ’ eq ‘ b ’); print $judge;
$judge  = (15<=>5); print $judge;
$judge  = (‘ aa ’ cmp ‘ bb ’); print $judge;
```

問題 22(★☆☆☆☆) $first と$second にランダムに数を代入しましょう．そして二つの数字を比較した結果と共に出力してください．

9 条件分岐

9.1 if/elsif/else

解説 ”アルゴリズム ”とはある問題を解決する手順のことですが，以下の三つの要素でほぼどのようなアルゴリズムも実装可能です\footnote{興味のある人はダイクストラの構造化定理についてしらべみましょう}．if/elsif/else という三段階の条件分岐を正確に使いこなすことがうまい Perl 使いへの近道となります．

1) 手続き A → B → C と順番に式を実行していく

2) ループ A → B → C → A → B → C →. . . とある条件のもと式の塊をくりかえす

3) 条件分岐 A の結果が真なら次に B，偽なら C を行う

```
my $seq = ‘ acgtagtcgtgtga ’;
my $length = length $seq;
my $ans;
if($length >= 20){
    $ans = ‘ long ’;
}
elsif($length <= 5){
    $ans = ‘ short ’;
}
else{
    $ans = ‘ middle ’;
}
print “ The sequence is $ans\n ”;
```

問題 23(★☆☆☆☆) 閏年を判定するプログラムを作りましょう

9.2 last, next, redo

解説 ループを途中で強制的にスキップする，もしくは抜け出したいときがあります．これを実現する関数が三つほど用意されています．違いをよく確認しておきましょう

last 直ちにループを抜け出す

next 直ちに次のループに入り，next 以降のブロック要素は無視される．

redo next と同じく次のループに入る．ただし条件式はスルーされる．

次の例は redo を使った有名な文例です．あるファイルの文中，\(バックスラッシュ) で終わる行を次の行と連結して表示します．

```
while(<FILE>){
    if(/\\\n/){
            chomp;
            s/\\//g;
            $next = <FILE>;
            $_    .=  $next;
            redo;
        }
        print;
}
```

問題 24(★☆☆☆☆) 上の例を実際に実行し redo の代わりに next や last を使えない理由を考えましょう．

9.3 三項演算子

解説 A が真ならば B，偽ならば C という文を簡潔に表現することが出来ます．

```
my($foo, $bar) = (20, 30);
$foo = ($foo >= $bar)? $foo: $bar;
print $foo;
```

問題 25(★☆☆☆☆) 先ほど作った閏年判定プログラムを三項演算子を用いて書き換えましょう．

10 正規表現

10.1 マッチ演算子

解説 ある文字列があり，あるキーワードでマッチさせたいとき正規表現を用います．

書式 $sequence =~ /pattern/;

=~は二文字でひとつの演算子です．=と~の間に空白はあけてはいけません．また pattern は "/ "二つで囲みます．さらに pattern は変数を含むことが出来ます．この場合変数は展開された後，pattern

として参照されます.

```
my $seq   = ‘ aucugcugccaugguguagc ’;
my $kozak = ‘ gccaugg ’;
if($seq =~ /$kozak/){
    print “ Kozak\n ”;
}
else{
    print “ Non kozak\n ”;
}
```

問題 26(★☆☆☆☆) ランダムな塩基配列を作り，その中に atg と (tag/tga/taa) という二つのパターンが同時に存在するか調べ，結果を出力しましょう.

10.2 メタキャラクタ，ワイルドカード，後方参照

解説 Perl はあいまい検索を行うことが出来ます．よく用いる正規表現 (メタキャラクタ) はリファレンスにまとめましたので参考にしてください.

.（ドット）などメタキャラクタとなる文字は正規表現内では文字列としては認識されないため，文字列としての.（ドット）とマッチさせたいときは\.のように\(バックスラッシュ) でエスケープさせる必要があります．さらに小カッコで囲うとその内容は$1,$2,$3…と\1,\2,\3…に入ります．正規表現の外では$を，正規表現内では\1 を使います．また，複数の小カッコがあるとき，囲われた内容の入る変数は開きカッコの順番で決まります.

```
my $haiku = ‘ Tabiniyande YumehaKarenowo Kakemeguru ’;
my $data = ‘ Hideki ’s uniform number is 55. ’;
my $html = ‘ <p>hoge <I>hogehoge</I> hoge</p>  ’

if($haiku =~ /^(\S+)\s+(\S+)\s+(\S+)$/){
    print “ $1\n ”;
    print “ $2\n ”;
    print “ $3\n ”;
}
if($data =~ /(\d+)/){
    print “ $1\n ”;
}
if($html =~ /^<([^>]+)>(.*?)<\/\1>/){
    print “ $2\n ”;
}
if($haiku =~ /\b(\S+[aiu])\b/;){
    print “ $1\n ”;
}
```

問題 27(★☆☆☆☆) 300 塩基長のランダムな塩基配列を作り，atg の前後 6 塩基ずつを全て表示してください.

ヒント while(/ pattern /g) 処理 でマッチの結果を順に見ることが出来ます.

10.3 量指定子

解説 文字の連続に関して指定が出来ます.

```
my $mRNA = ‘ atgtgctgatcgtagctgaaaaaaaaa ’;
if($mRNA =~ /^([a-z]+)a{5,}$/){
        print “ $1\n ”;
}
else{
        print “ $mRNA\n ”;
}
```

問題 28(★☆☆☆☆) 300 塩基長のランダムな塩基配列を作り，c か g のみからなる 10bp 以上の長さの領域があるかどうかを判定しましょう.

11 関数

11.1 組み込み関数

解説 print，split 等すでにいくつ使ってきましたが，perl にはたくさんの関数が用意されています．最初は以下にあげたものを知っていれば十分です．このうちいくつかは後のセクションで扱います.

```
出力：          print, printf
算術計算：      log, sqrt, sin/cos, exp, abs, int, rand
文字列操作：    length, reverse, index/rindex, substr, lc/uc, chomp
                split/join, s///g, tr///, sprintf
配列操作：      shift, unshift, pop, push, sort, reverse, map, grep, splice
ハッシュ操作：  keys/values, each, exists/defined, delete
制御：          last, next, redo, exit, die, return
宣言：          my, sub
システム:       open/close, opendir/readdir, select, system, package, use
```

問題 29(★☆☆☆☆) 上にあげた関数の機能を調べましょう.

11.2 ユーザ関数

解説 組み込み関数と同様な関数をユーザが作ることが出来ます.

sub 関数名{ 処理 }

とすることで関数を宣言します．使うときは

&関数名 (引数 1，引数 2，…)

です．&と小カッコは条件によって省略できます

以下の例は相補鎖を求める関数です

```
sub complement{
    my $seq = shift;
    $seq =~ tr/atcgATCG/tagcTAGC/;
    $seq = reverse $seq;
    return $seq;
}

my $seq = ‘ ttaactgatgctgtcgatctagctcat ’;
print &complement($seq), “ \n ”;
```

問題 30(★☆☆☆☆)

a) reverse 関数を自分で実装しましょう.

b) 塩基を含む配列を与えるとエントロピーを計算する関数を実装しましょう

ヒント reverse 関数はループ及び配列の引数をうまく用いて作ると良いでしょうエントロピー H は P_i を塩基 i の出現頻度をしたとき

$$H = -\sum_{n=a,t,c,g} P(n)\log_2 P(n)$$

で定義されます. ただしここでは $0\log 0 := 0$ とします

12 リファレンス

12.1 明示的なリファレンス作成

解説 リファレンスは変数の住所のようなものです. 値自体ではなく名前のみをやり取りすることでメモリーの節約やプログラムの効率の向上に寄与します. また関数に配列や複数の変数を渡すケース, あるいは多元配列/ハッシュの実装に用いられます.

```
my $seq = ‘ atgctgtagctgtgatgctgatgcg ’;
my $seq_ref = \$seq;
my @array = (‘ a ’, ‘ b ’, ‘ c ’, ‘ d ’, ‘ e ’, ‘ f ’);
my $array_ref1 = \@array;
my %hash = (‘ a ’, 1, ‘ b ’, 2, ‘ c ’, 3);
my $hash_ref = \%hash;
sub hello{print “ Hello $_[0]\n ”}
my sub_ref = \&hello;

print $$seq_ref, “ \n ”;
print $array_ref->[1], “ \n ”;
print $hash_ref->{a}, “ \n ”;
print $sub_ref->(‘ john ’), “ \n ”;
```

問題 31(★☆☆☆☆) 先ほど作った reverse 関数を参照渡しに変更しましょう

12.2 無名配列

解説 上では\を用いてリファレンスを明示的に作成しましたが，[ ] もしくは{ }を用いてより直接的にリファレンスを作成することが出来ます．これらを無名配列/ハッシュ/関数といいます．

```
@array = (‘ a ’, ‘ b ’, ‘ c ’, ‘ d ’, ‘ e ’, ‘ f ’);
$array_ref1 = [@array];
$array_ref2 = [‘ a ’, ‘ b ’, ‘ c ’, ‘ d ’, ‘ e ’, ‘ f ’];
$hash_ref = {‘ a ’, 1, ‘ b ’, 2, ‘ c ’, 3};
$sub_ref = sub{print “ Hello $_[0] ”};

print $array[1], “ \n ”;
print $array_ref1->[1], “ \n ”;
print $array_ref2->[1], “ \n ”;
print $hash_ref->{a}, “ \n ”;
print $sub_ref->(‘ john ’), “ \n ”;
```

問題 32(★☆☆☆☆) 各自配列を作り，さらにそのリファレンスを作ってください．ただしリファレンス演算子を用いた方法，無名配列を用いた方法の二通りを行ってください．そのあと元の配列を変え，リファレンスの中身を見てみましょう．

12.3 多元配列，行列

解説 リファレンスを用いて多元配列を作成することが出来ます．

```
@array = ([1, 2, 3], [4, 5, 6], [7, 8, 9]);
$array_ref = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];

print $array[1][1], “ \n ”;
print $array_ref->[1][1], “ \n ”;
```

12.4 多元ハッシュ，構造体

解説 多元配列と同様に多元ハッシュも作成することが出来ます．さらにスカラー変数，配列，関数等を要素に含む構造体 like なデータ構造も実現できます．

```
%hash = (
    a => {1, 3, 2, 2},
    t => {1, 5, 2, 5},
    c => {1, 3, 2, 1},
    g => {1, 4}
)
$hash{g}{2} = 1;

print $hash{g}{2}, “ \n ”;
```

問題 33(★☆☆☆☆) スカラー変数，配列，ハッシュ，関数への各リファレンスを含む多元ハッシュを構築しましょう．そしてそれぞれ出力し動作を確認しましょう．

13 文字列操作

13.1 部分文字列の操作

解説 substr を使うと文字列の操作を行うことが出来ます.

```
my $seq = ‘ abcdefg ’;
my $part1 = substr($seq, 2, 2);
my $part2 = substr($seq, -2, 2);
print $seq, “ \n ” ;
print $part1, “ \n ”;
print $part2, “ \n ”;

substr($seq, 0, 2) = ‘ xy ’;
substr($seq, 0, 2, ‘ pq ’);

print $seq;
```

問題 34(★☆☆☆☆) 300 文字のランダム塩基配列を作りましょう．そしてコドンごとに頻度をカウントし，数の多い順に表示してください.

13.2 パターンの検索

解説 m//, index, rindex はパターンを検索する関数です．m//は実は正規表現で出てきた ”=~ /pattern/ ” のダブルスラッシュのことです．このように m は省略できますが，あえてつけた場合 ”=~ m#pattern# ” のようにパターンをスラッシュ以外の記号で囲うことが出来ます．index/rindex は文字列から最左 (右) のパターンマッチの位置を返す関数でパターンマッチに失敗したときは-1 を返します.

```
$seq = ‘ actgacgatgatgtgcatgc ’;
($forward, $behind) = /([atcg]{6})atg([atcg]{6})/ for $seq;
$l_pos = index($seq, ‘ atg ’, 0);
$r_pos = rindex($seq, ‘ atg ’);

print “ $forward\t$behind\n ”;
print “ $l_pos\t$r_pos\n ”;
```

問題 35(★☆☆☆☆) index/rindex は最左，もしくは最右の要素しか見つけることが出来ませんが，それだと目的にそぐわない場合があります．そこでパターンの見つかった場所全てを返す windex 関数を実装してみましょう．ループとうまく組み合わせるとよいでしょう.

13.3 置換

解説 s//, tr///はそれぞれ文字列の置換を行う関数です.

```
#$seq から塩基以外の文字をのぞく
$seq =~ s/[^a-z]//g;

#カウント (二つの式は同じ意味)
$count = $seq =~ tr/gcGC/gcGC/;
$count = tr/gcGC/gcGC/ for $seq;
```

問題 36(★☆☆☆☆) 置換演算子を用いて塩基配列中における GC の数をカウントしましょう.

14 配列操作

14.1 末端要素の操作

解説 配列の末端要素を操作する関数は 4 つ用意されています.

shift 先頭要素をひとつ削除

pop 末尾要素をひとつ削除

unshift 先頭に要素を追加

push 末尾に要素を追加

```
my @test1 = (‘ a ’, ‘ b ’, ‘ c ’);
my @test2 = (‘ d ’, ‘ e ’, ‘ f ’);
my $shift = shift @test1;
my $pop = pop @test1;
unshift( @test2, ‘ c ’ )
push( @test2, ‘ g ’ );

print “ $shift\n ”;
print “ $pop\n ”
print “ @test1\n ”;
print “ @test2\n ”;
```

問題 37(★☆☆☆☆) これらの関数を用いて配列の最初と最後の要素を入れ替えましょう.

14.2 要素の置換

解説 substr は文字列の操作を行う関数ですが, 同様に配列の操作を行う関数に splice があります. 書式はほぼ substr と同じです.

```
my @test = (‘ a ’, ‘ b ’, ‘ c ’, ‘ d ’, ‘ e ’, ‘ f ’, ‘ g ’);
my @test2 = splice(@test, 3, 2);
splice(@test, 1, 2, (‘ a ’, ‘ b ’));

print “ @test\n ”;
print “ @test2\n ”;
```

問題 38(★☆☆☆☆) splice を使って shift，unshift，push，pop と同じ動作をする関数を作ってみましょう.

14.3 スライス

解説 配列の要素をひとつとってくるときは$array[0] としますが，スライスを用いると同時に複数の要素を取ることが出来ます.

書式 ($first, $second) = @array[0,1];
$の代わりに@を使います．ハッシュも同様にスライスを適用可能です

```
my $seq = ‘ abcdef ’;
my @array = (‘ a ’, ‘ b ’, ‘ c ’, ‘ d ’, ‘ e ’, ‘ f ’);
my %hash = (‘ a ’, 1, ‘ b ’, 2, ‘ c ’, 3);
my @chr = ( split(‘ ’, $seq) )[0,1,2];
my @num = @hash{@array[2,1]};

print “ @chr\n ”;
print “ @num\n ”;
```

問題 39(★☆☆☆☆) 配列を作成し，スライスを用いて要素の交換を行いましょう.

14.4 負の引数

解説 配列の最後の要素へは最後の要素の添え字を表す$#配列名を利用し

$hoge[$#hoge]
でアクセスすることが出来ますが，負の引数を用いて
$hoge[-1]
でもアクセスできます．同様に最後から n 番目の要素は
$hoge[-n]
です.

```
@array = (‘ a ’, ‘ b ’, ‘ c ’, ‘ d ’, ‘ e ’, ‘ f ’);
print $#array, “ \n ”;
print $array[$#array], “ \n ”;
print $array[-1], “ \n ”;
```

問題 40(★☆☆☆☆) スライスを用いて最初と最後の要素を入れ替えましょう.

14.5 ソート

解説 ソートを行う sort 関数はデフォルトでは辞書順に従った並べかえを行いますが，正確に数値の大きさ順にそろえる，複雑な並べ替えを行いたいといった場合，並べ替え方を指定することも出来ます[5]. 以下は塩基の長さについてまず綴りの短い順に並べ，同じ長さの名前に関してはアルファベット順に並べています.

```
my @nuc = (‘ adenine ’, ‘ guanine ’, ‘ thymine ’, ‘ cytosine ’, ’uracil’);
my @sorted = sort{
    length $a <=> length $b ||
    $a cmp $b
}@nuc;
```

問題 41(★☆☆☆☆)

`@number = (1, 5, 30, 22, 3, 9);`

を作成し以下の二つのソートの動作比較をしましょう

`sort @number`

`sotr{$a <=> $b}@number`

14.6 配列の編集

解説 map と grep は配列の編集に威力を発揮する関数です．以下に基本的な使用例をあげますが，是非 sort と共に使いこなせるようにしておきましょう.

```
my @even = map{$_*2}0..50;
my @quadric = map{[$_, $_**2]}0..20;
my @lion_king = grep{/lion king/i}@musical;

#以下は@data のデータを%other_data に記述されたデータにしたがってソートする効率の良い方法 ”Schwartz 変換 ”です.
my data_sorted =
map{$_->[0]}
sort{$a->[1] <=> $b->[1]}
map{[$_, $other_data{$_}]}@data;
```

問題 42(★☆☆☆☆) 要素のダブリを含む配列からダブリを除くようなプログラムを作成しましょう．ただし grep を用いた方法とハッシュを用いた方法の 2 通りを考えてください.

問題 43(★☆☆☆☆) 実際に Schwartz 変換を使ってみましょう

15 外部入出力，制御

15.1 ファイルへの書き出し

解説 読み込み用にファイルを開く場合は

[5] デフォルトのままだと例えば 12,3,23 をソートした場合 12, 23, 3 と並べてしまいます.

open FILE, hoge.txt;
としました．同様に書き出し用のファイルを開くことも出来ます．
open FILE, “ >output.txt ”;
これで output.txt というファイルが新たに生成され，データを書き込むことが出来ます．ただし，>では既存のファイルを上書きしてしまうためそれを避けるために>>を用いることもあります．>は無条件に新規ファイルの生成を行いますが，>>は同名のファイルがなかった場合は新規作成，あった場合は末尾にデータを追加します．

```
open FILE, “ >output.txt ” or die;
print “ one\n ”;
print FILE “ two\n ”;
select FILE;
print “ three\n ”;
select STDOUT;
print “ four\n ”;
```

問題 44(★☆☆☆☆) 上のコードを実行し，動作を確認しましょう．

15.2 @ARGV

解説 スクリプトへの引数は@ARGV に収められています．
コマンドライン上

```
% view_one_line.pl nucleotide.txt 5
```

スクリプト

```
#!/usr/bin/perl

open FILE, $ARGV[0] or die ”Can ’t open $ARGV[0]:$! ”;
my @line = <FILE>;
print “ $line[$ARGV[1]-1]\n ”;
```

問題 45(★☆☆☆☆) 複数の数値を引数として受け取り，平均を出力するスクリプトを作成しましょう．

15.3 外部コマンドの使用

解説 外部コマンドを使用する場合，system 関数を使う，あるいはバッククォーテオションを使うという二通りの方法があります．

```
system(“ ls -l ”);
my @who = ‘who‘;
for(@who){
    my @line = split;
    print “ $line[0]\t$line[4]\n ”;
}
```

問題 46(★☆☆☆☆) last コマンドの出力を受け取り，PC へのアクセス回数を表示させましょう．

16 囲い文字

16.1 クォーテーション

解説 クォーテーションは三種類あります.

“ /qq （ダブルクォーテーション）
文字列．変数/エスケープ文字が含まれていた場合，解釈される．
“ $hoge\n ” ← $hoge の中身が展開され，改行が付加される
qq#$hoge\n# ← 上に同じ．囲い文字は自由.

‘ /q （シングルクォーテーション）
文字列．ただし変数は展開せず ”そのまま ”に解釈される．
‘ $hoge\n ’ ← $hoge\n という文字列
q#$hoge\n# ← 上に同じ

‘/qx （バッククォーテーション）
中にコマンドを書く．するとコマンドは実行され，結果が返される．
$who =‘who‘; ← who というコマンドの実行結果が$who に入る
$who =qx#who# ← 上に同じ

qw 空白区切りでリストを作る．カンマ・囲いは不要.
@chr = qw(a b c d e f g);

```
my $foo =  ’bar ’;
my @continent = qw(Africa Antarctica Australia Eurasia
 North_America South_America);

print ‘ $foo\n ’;
print “ $foo\n ”;
print ‘who‘;
print “ $continent[1]\n ”;
```

問題 47(★☆☆☆☆) コマンドラインからファイル名を受け取り，.txt という拡張子をつけて適当に文字列を書き込みましょう.

問題 48(★☆☆☆☆) lynx と組み合わせて簡単な web robot を作成しましょう．例えば GenBank からファイルをダウンロードし，アノテーションを抽出するような web robot を作成しましょう.

16.2 [ ]

解説 [ ] は文脈によって以下のようなことなる意味を持ちます.

配列の添え字 $array[2]

スライスの添え字 @slice[3..5]

リファレンスの添え字 $array_ref->[6] or $$array_ref[6]

無名配列 [1, 1, 2, 3, 5, 8, 13, 21]

```
my $foo =  'a ';
my $bar =  'b ';

$ref1 = [$foo, $bar];
@ref2 = \($for, $bar);
```

問題 49(★☆☆☆☆) 例文で出来てきた$ref1 と@ref2 の違いを理解しましょう.

16.3 {}

解説 { }も文脈によって異なる意味を持ちます

ハッシュの添え字 $hash{$key}

スライスの添え字 @hash{$key1, $key2}

リファレンスの添え字 $hash_ref->{$key} or $$hash_ref->{$key}

無名ハッシュ {1, ‘ a ’, 2, ‘ b ’, 3, ‘ c ’}

ループ foreach(配列){処理}

ブロック do{処理}while(条件);

特に dowhile(); 文では最後のセミコロンを忘れないようにしてください.

```
my $hoge = ‘ foo ’;
{
    my $hoge = ‘ bar ’;
    print “ $hoge\n ”;
}
print  ”$hoge\n ”;
```

問題 50(★☆☆☆☆) 引数を保持し，再び与えられた引数が元の引数と合致するか判定するクロージャを実装しましょう.

16.4 ( )

解説 ( ) に関しては特にリスト・配列・無名配列の違いを理解すること，正規表現に関するグループ化と後方参照を理解することが重要です.

```
my $chr = ‘ 5a3b1 ’;
my @array;
@array = /((\d)a(\d(b))\d)/ for $chr;

printf “ %s\n ”, $_ for @array;
```

問題 51(★☆☆☆☆) 例文から後方参照のルールを確認しましょう．また配列と無名配列の違いも確認しておきましょう.

16.5 <>

解説 山カッコ演算子はファイルハンドルを受け取ると，それを読み込む演算子です．ハンドルが省略されると STDIN を対象とします．また山カッコ演算子のデフォルト区切り文字は\n，つまり改行です．そのために文章を一行ずつ読み込めるわけですが，この区切り文字は以下のように変えることが出来ます．

```
local $/ = “ \n> ”;
while(<FILE>){
処理
}
```

これでカッコの中だけ区切り文字が>(ただし直前に改行有り) に変えることが出来ます．以下はこれを利用して効率よく fst 形式のファイルを加工している例です．

```
open FST, hoge.fst or die;
{
    local $/ = “ \n> ”;
    while(<FST>){
        my($head, @seq) = split /\n/;
        my($seq = join ‘ ’, @seq) =~ s/[-a-zA-Z]//g;
        $head =~ s/>//;
        $fst{$head} = $seq;
    }
}
```

問題 52(★☆☆☆☆) ls -R の結果をファイルに書き出し，それを効率よく読み込み，ハッシュに格納しましょう．

16.6 / /

解説 正規表現の囲い文字です．正確にはマッチ演算子 m//の省略形です．m が省略されると囲い文字は//でなければいけませんが，m を省略しない場合以下の例文のようなことが出来ます．

```
my $matsui = ‘ GODZILLA ’;
my $slash = ‘ ////// ’;

print  ”55\n ” if $matsui =~ m!GOD!;
print  ”6slashes\n ” if $slash =~ m#\/\/\/\/\/\/#;
```

問題 53(★☆☆☆☆) 100 文字と 3 文字の文字列をそれぞれ作り，100 文字の文字列中に 3 文字の文字列が含まれるかチェックする．という作業を 100 回繰り返し，結果を表示しましょう．

17 デバグの定石，コメントアウト

17.1 デバグしやすいコード

解説 プログラムを作成する際には入力ミス，アルゴリズムの不備，想定外のエラーはつきものです．そういったエラーをバグといい，直すことをデバグといいます．デバグはプログラミングの重要な一行程

であり，以下に効率よくデバグできるかは効率のいいプログラミングには必須のスキルとなります.

- 中身を書く前にカッコを閉じる
- コメントをつける
- スペース・改行・インデントの幅を統一する[6]
- 抽象度の高いプログラムにする

```
1)
$hoge[ ]
↓
$hoge[5]

2)
substr( );
↓
substr($seq, 0, 3);

3)
while( ){

}
↓
while($num > 5){

}
↓
while($num > 5){
    print $num++;
}
```

問題 54(★☆☆☆☆) ここで使われているプログラミングのコツとはなんでしょうか.

17.2 コメントアウトの仕方，デバグの手順

解説 printout，コメントアウト，カット，__END__，exit，die
デバグのときにはこれらを用いるといいでしょう.
(a) コメントアウト #に続く一行はインタプリタに無視されます

```
#この行は無視されます
#この行は無視されます
print “ Hello World\n ”;
```

(b) POD の応用 広い範囲をコメントアウトするときに用います

[6]emacs を使用している場合は Perl-mode を使うと便利です.

```
=c
この行は無視されます
この行は無視されます
=cut
print “ Hello World\n ”;
```

(c) __END__ 以降のコードを無効にしたいときに用います．アンダーバーが計 4 つ使われていることに注意してください

```
print “ Hello World\n ”;
__END__
これから後は無視されます
```

(d) exit プログラムを強制的に終了したいときに用います．

```
print “ Hello World\n ”;
if($a != 1){
    exit;
    $a が 1 ではないときプログラムはここで終了します
}
```

(e) die エラーによる強制終了には die を用います

```
open FILE, ‘ ecoli.fst ’ or die ”Can ’t open ecoli.fst:$! ”;
ecoli.fst が開けないとき，その旨出力してからプログラムが終了します
```

問題 55(★☆☆☆☆) 上に挙げた方法の動作をそれぞれ確認しましょう．

17.3 ありがちなミス，対処法

解説 スクリプトを作成中に起こるありがちなバグというものがあります．しかしそれに対する対処法を知っているとデバグが楽になります．以下にエラーメッセージの例を示します．

事例 1 変数の宣言忘れ or タイプミス or $ 等のつけ忘れ

```
・Global symbol "$hoge" requires explicit package name
・Bareword "hoge" not allowed while "strict subs"
・Odd number of elements in hash assignment
・HASH(0x9441c18)
・Type of arg 1 to keys must be hash (not private array)
```

事例 2 FILE が開けなかった or 外部環境が整っていない or ヘッダーが変

```
・Can’t locate hoge.pm in @INC…
・Command not found.
```

事例 3 カッコの数があっていない

```
・Missing right curly or square bracket
・コードのないところでのエラーメッセージ
```

事例 4 無限ループ or 深すぎる再帰

```
・Deep recursion on subroutine "main::hoge"
・プログラムが止まらない.
```

事例 5 フォーマットの読み間違い

```
・substr outside of string
・Use of uninitialized value
```

問題 56(★☆☆☆☆) 上に挙げたエラーを実際に出力させてみましょう．プログラムが止まらなくなってしまったら Ctrl+C で強制終了させることができます.

18 モジュール

18.1 作り方

解説 プログラムの一部を別のファイルに記述し，後でそれを読み込み利用することが出来ます．この別ファイルのことをモジュールといいます．モジュールを活用することでプログラムの再利用性や可読性を向上させることが出来ます．以下は最低限の要件を満たしたモジュールの書き方です.

```
#!/usr/bin/env perl
use strict;
package My_module;
require Exporter;
our @ISA = qw(Exporter AutoLoader);

our @EXPORT = qw(
                hello_world
);
our $VERSION = '1.0';

sub hello_world{
    print " hello_world\n ";
}

1;
__END__
```

問題 57(★☆☆☆☆) 上の例文を My_module.pm というファイルに保存しましょう.

18.2 使い方

解説 use\index{use}関数を用いてモジュールをインポートすることが出来ます.

```
use My_module;
hello_world;
```

問題 58(★☆☆☆☆) 上のようにして，作ったモジュールを利用してみましょう.

問題 59(★☆☆☆☆) nstore を使って変数を保存し，さらに別のプログラムで再利用してみましょう.

問題 60(★☆☆☆☆) 任意のプログラム (Matinspector, clustalW, G 等) を動かす expect を作成し，それと組み合わせたモジュールを作成してください．そして引数を与えて実行しましょう.

19 略記法，慣用句，定石

19.1 $_ を使う

解説 基本的に関数，ループ，演算子は引数を与えなくてはいけませんが，与えなかった場合デフォルトの引数を考慮してくれます．これを逆手にとって省略した書き方が可能です．デフォルトの引数には $_, @_, @ARGV, STDIN などがあります.

```
printf “ %s\t%s\n ”, (split)[0, 7] for ‘w‘;
```

問題 61(★☆☆☆☆) 上のコードを省略のない形に戻しましょう.

19.2 カッコを減らす

解説 いままでもでてきましたが，関数のデフォルトや特殊なループ表現を用いると，余計なカッコを減らすことが出来ます．無理に減らすのは逆効果ですが，決まったパターンの省略法を身につけると見やすいプログラムが書けます．例えば下の式は皆同じ意味です.

```
#i
foreach my $i(@array){
    my @elm = split(/ /, $i);
    print “ $elm[0]\t$elm[1]\n ”;
}

#ii
for(@array){
    my @elm = split(/ /, $i);
    printf “ %s\t%s\n ”, @elm[0,1];
}

#iii
printf “ %s\t%s\n ”, (split)[0,1] for @array;
```

問題 62(★☆☆☆☆) 今まで書いてきたコードを見直し，カッコが減らせる箇所がないか探して見ましょう.

19.3 関数のデフォルト設定を生かす

解説 関数のデフォルト設定を生かすと余分なコードが減り，見やすくなります．ユーザ関数の中でデフォルト設定を行いたいときは以下のように書くと良いでしょう

```
sub hoge{
    my($foo, $bar) = @_;
    $foo ||= 10;
    $bar ||= ‘ atg ’;
}
```

問題 63(★☆☆☆☆) デフォルト設定にはどういうものがあるか調べてみましょう．特に

- chomp
- m//
- s///g
- shift
- split

はデフォルト設定について把握しておきましょう．

第3章　実践 Bioinformatics

この章では Perl のレファレンスの使い方を徹底的にマスターし，配列の重み行列の作成など配列解析に応用できる技術の取得を目指します.

1　Perl によるシグナル配列の解析

1.1　ハッシュの復習

解説 Perl でハッシュは非常に頻繁に使われます．ある要素と他の要素を簡単に関係付けることができるので，非常に便利です．例えば，あるタンパク質がどのような機能を持っているかを記録していくときに，

```
my %func;
```

で，タンパク質名と機能を対応させるハッシュの変数を定義することができます.

```
$func{ "SAM1" } = "Metabolism";
$func{ "CRM1" } = "Transport";
$func{ "TAF25" } = "Transcription";
```

で実際にタンパク質名と機能とを対応させることができます．タンパク質名は“ キー ”となり，機能はその“ 値 ”となります.

```
my @proteins = keys(%func);
```

は%func に登録されている全てのキーを抽出し，配列に入れます．キーの順序は決まっていません.

```
foreach my $protein (@proteins){
   print "$protein\t$func{$protein}\n";
}
```

こうすると，登録されているタンパク質名とその機能の一覧が出力されます．あるキーがハッシュに登録されているかを調べるには，defined を使います.

```
if(defined($func{ "TAF25" })){
   print $func{ "TAF25" }, "\n";
}
```

問題 1 タブ区切りのタンパク質間相互作用データのファイルと，タンパク質の機能を示したファイルをもとに，相互作用するタンパク質の機能を次のように表示していくプログラムを作成しましょう.

```
タンパク質 A タンパク質 B タンパク質 A の機能 タンパク質 B の機能
タンパク質 C タンパク質 D タンパク質 C の機能 タンパク質 D の機能
タンパク質 E タンパク質 F タンパク質 E の機能 タンパク質 F の機能
                              :
```

タンパク質間相互作用ファイルの例：

```
ACS2    SNP1
ACT1    ABP1
ACT1    AIP1
BRR2    SNP1
```

タンパク質の機能ファイルの例：

```
ABP1    Vesicular_transport
ACS2    Carbohydrate_metabolism
ACT1    Vesicular_transport
AIP1    Cell_structure
BRR2    RNA_splicing
SNP1    RNA_splicing
```

結果の例：

```
ACS2    SNP1    Carbohydrate_metabolism    RNA_splicing
ACT1    ABP1    Vesicular_transport    Vesicular_transport
ACT1    AIP1    Vesicular_transport    Cell_structure
BRR2    SNP1    RNA_splicing    RNA_splicing
```

問題 2 タンパク質間相互作用データのファイル中で各タンパク質が何回登場するか，ハッシュを用いてカウントして出力しましょう．

1.2 関数へのリファレンスの受け渡し

解説 関数で直接複数の配列や複数のハッシュを受け取ることはできません．例えば次は配列@a と@b の要素を先頭から掛け合わせてその総和を計算する関数 i_product を定義していますが，これでは正しく動きません．

```
#!/usr/bin/env perl

use strict;    # 全ての変数を my で宣言するように強制する
use warnings;  # プログラム動作に不合理な点があったときに警告を出す

sub i_product {
   my(@vector1, @vector2) = @_;
   my $total = 0;
   foreach my $i (0..$#vector1){
      $total += $vector1[$i] * $vector2[$i];
   }
   return $total;
}

my @a = (1,2,3);
my @b = (4,5,6);
print i_product(@a, @b);
```

なぜなら，i_product(@a, @b) としたときに，@a と@b の中味が展開され (1,2,3,4,5,6) となり，@a と@b の境界がなくなってしまうからです．このケースでは，@vector1 に (1,2,3,4,5,6) が入ってしまい，@vector2 には何も入りません． そこで配列そのものではなく，配列へのリファレンスを渡すことでこの問題を回避します．リファレンスは数値や文字列と同じ部類であるスカラーに入るの

で，問題なく複数のリファレンスの受け渡しを行うことができます．

```
#!/usr/bin/env perl

use strict;    # 全ての変数を my で宣言するように強制する
use warnings;  # プログラム動作に不合理な点があったときに警告を出す

sub i_product {
   my($vector_ref1, $vector_ref2) = @_;
   my $total = 0;
   foreach my $i (0..$#$vector_ref1){
      $total += $vector_ref1->[$i] * $vector_ref2->[$i];
   }
   return $total;
}

my @a = (1,2,3);
my @b = (4,5,6);
print i_product(\@a, \@b);
```

- @a は配列 (1,2,3) を表します．
- $vector_ref1 は配列 (1,2,3) へのリファレンスを表します．
- @$vector_ref1 は配列 (1,2,3) を表します．これは@a と等価です．
- $$vector_ref1[1] および$vector_ref1->[1] は配列@$vector_ref1 の 1 番目の要素，つまり 2 を表します（0 番目から数える）．

配列だけでなく，ハッシュもリファレンスにして関数に渡すことができます．例えば下のプログラム (一部) を考えてみましょう．

```
sub test_hash_ref {
   my($h1_ref, $h2_ref) = @_;
   print $h1_ref->{ "height" }, "\n";
   print $h2_ref->{ "weight" }, "\n";
}

my %c = ("height" =>160, "weight" => 55);
my %d = ("height" =>175, "weight" => 70);
test_hash_ref(\%c, \%d);
```

- %c はハッシュ(“ height ” =>160, “ weight ”=> 55) を表します．
- $h1_ref はハッシュ("height" => 160, "weight" => 55) へのリファレンスを表します．
- %$h1_ref はハッシュ(“ height ” =>160, “ weight ”=> 55) を表します．これは%c と等価です．
- $$h1_ref{“ height ”}および$h1_ref->{“ height ”}は 160 を表します．

問題 3 配列へのリファレンスとハッシュへのリファレンスを受け取り，配列をハッシュに従って置き換えたものを返す関数 hash_trans を作成しましょう．例えば，@a = (“ a ”, “ b ”, “ a ”, “ c ”),

%h = (“ a ” => “ apple ”, “ b ”=> “ banana ”, “ c ” => “ candy ”) として hash_trans(\@a, \%h) とすると，(“ apple ”, “ banana ”, “ apple ”, “ candy ”) が返ってくるようにします．

1.3 リファレンスを含む配列とハッシュ

解説 今まで配列で扱ってきた要素は数値か，文字列でした．

```
@array = (“ a ”, “ t ”, “ g ”, “ c);
```

しかしそれ以外に，配列へのリファレンスを配列の要素とすることができます (ref はリファレンスを表します)．

```
("a", "t", ref, "g")
          ↓
          (0, 1, 2, 3)
```

上記を実現するためには，

```
@array2 = (“ a ”, “ t ”, [0, 1, 2, 3], “ g ”);
```

とします．あるいは，

```
@array3 = (0, 1, 2, 3);
```

という代入をしておき，

```
@array2 = (“ a ”, “ t ”, \@array3, “ g ”);
```

とすることもできます．この場合は@array3 を操作すると，@array2 の中味に影響が出ます．ここで，

- $array2[1] は ”t ”を表します．
- $array2[2] は (0, 1, 2, 3) へのリファレンスを表します．
- @{$array2[2]}は (0, 1, 2, 3) を表します．
- ${$array2[2]}[3] は 3 を表します．もう少し見やすい表記で，$array[2]->[3] も 3 を表します．これをうまく使いこなすと，二次元配列の表現が可能になります．

同様にして下の例のように，ハッシュへのリファレンスを含む配列を作ることも可能です．

```
("a", "t", ref, "g")
          ↓
          ("X" => 123, "Y" => 456)
```

```
@array4 = (“ a ”, “ t ”, {“ X ”=>123, “ Y ”=>456}, “ g ”);
```

- $array4[2] は{“ X ”=>123, “ Y ”=>456}へのリファレンスを表します．
- %{$array4[2]}は{“ X ”=>123, “ Y ”=>456}を表します．
- ${$array4[2]}{“ X ”}は 123 を表します．もう少し見やすい表記で，$array4[2]->{“ X ”}も 123 を表します．

リファレンスを要素に持つハッシュを作ることも可能です．

```
("X" => 100, "Y" => ref, "Z" => 200)
                    ↓
                    ("a" => 12, "t" => 45)
```

これを実現させるためには例えば，次のようにします．

```
%hash1 = (“ X ”=>100, “ Y ”=>{“ a ”=>12, “ t ”=>45}, “ Z ”=>200);
```

- $hash{“ X ”}は 100 を表します．
- $hash{“ Y ”}は{“ a ”=>12, “ t ”=>45}へのリファレンスを表します．
- %{$hash{“ Y ”}}は{“ a ”=>12, “ t ”=>45}を表します．
- ${$hash{“ Y ”}}{“ a ”}は 12 を表します．もう少し見やすい表記で，$hash{“ Y ”}->{“ a ”}も 12 を表します．

問題 4 配列へのリファレンスからなる配列の中味を表示する関数 showarray2d を作成しましょう．

```
@a = ([1,2], [2,3], [1,1]);
print show_array_2d(@a);
```

とすると

```
1,2
2,3
1,1
```

のような出力が得られるようにします．

問題 5 行列計算をする関数 mproduct を作成しましょう．

```
@a = ([1,2],[2,3],[1,1]);
@b = ([2,1],[1,2])
@c = m_product(\@a, \@b);
```

とすると，下記の計算が行われます．

$$\begin{pmatrix} 1 & 2 \\ 2 & 3 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 1 \\ 1 & 2 \end{pmatrix} = \begin{pmatrix} 4 & 5 \\ 7 & 8 \\ 2 & 3 \end{pmatrix}$$

このとき，c は ([4,5], [7,8], [3,3]) となるようにしましょう．

1.4 重み行列の作成

解説 転写開始部位周辺や，開始コドン周辺は特定の塩基で偏っていることが知られています．どの位置でどの塩基にどれくらいの偏りがあるかを表現する 1 つの方法が重み行列を使うことです．

今，以下の開始コドン周辺の塩基配列を例にとりましょう．囲みが開始コドンです．

agt atg act
cac atg aac
tag atg aga
tac atg cga
agc atg aga

開始コドンの a の位置を 0 とし，その上流に行くに従って位置を 1 だけ引いていき，下流に行くに従って 1 だけ足していきます．すると各位置における各塩基の頻度を表した以下のような表ができます．

表 3.1: 度数表

	-3	-2	-1	0	1	2	3	4	5
A	2	3	0	5	0	0	4	1	3
C	1	0	3	0	0	0	1	1	1
G	0	2	1	0	0	5	0	3	0
T	2	0	1	0	5	0	0	0	1

これを割合に直すと，以下のようになります．

表 3.2: 重み行列

	-3	-2	-1	0	1	2	3	4	5
A	0.4	0.6	0.0	1.0	0.0	0.0	0.8	0.2	0.6
C	0.2	0.0	0.6	0.0	0.0	0.0	0.2	0.2	0.2
G	0.0	0.4	0.2	0.0	0.0	1.0	0.0	0.6	0.0
T	0.4	0.0	0.2	0.0	1.0	0.0	0.0	0.0	0.2

これを重み行列 (`weight matrix`) といいます．重み行列はハッシュおよびリファレンスを使うと簡単に表現できます．重み行列を表すハッシュを`%matrix` として，例えば`$matrix{ -2 }->{ “ c ” }` を`-2` の位置の `c` の頻度を表す数値にするのが分かりやすいでしょう．では，ここで与えられた塩基配列群から行列`%matrix` を返す関数 `return_matrix` を作ってみましょう．

```
sub return_matrix {
   my($seqs_ref, $start) = @_;
   # $seqs_ref は塩基配列群へのリファレンス
   # $start が位置 0 を塩基配列の何番目にするかという数字
   my %matrix;
   my $j = 0;

   my $counted_flag; # 0 か 1 の値を取るフラッグ.
   # 0: 長さ$j+1 以上の配列が一本もない
   # 1: 長さ$j+1 以上の配列が少なくとも一本はある

    do {
        $counted_flag = 0;
        for my $i (0..$#$seqs_ref){ # 配列の数だけ繰り返す
            if($j < length($seqs_ref->[$i])){
               # 配列一本の長さが$j+1 以上かどうか確認.
            $counted_flag = 1; # フラッグを立てる
            my $nuc = substr($seqs_ref->[$i], $j, 1);
            # 対象配列の$j 番目の塩基を取り出す.
            $matrix{ $j - $start }->{ $nuc } ++;
            # 行列中の値に反映させる
         }
      }
      $j ++;
   } while ($counted_flag);

   return %matrix;
}
```

なお

```
do {“ 処理 ”} while(“ 条件 ”);
```

で，まず 1 回“ 処理 ”を行い，次からは“ 条件 ”が成立するときに“ 処理 ”を繰り返します.
上記プログラムを作成した上で下のように関数を呼び出すと，3 番目の塩基を位置 0 としたときの行列%matrix を作成することができます.

```
my @sequences = (
   "agtatgact",
   "cacatgaac",
   "tagatgaga",
   "tacatgcga",
   "agcatgaga");
my %matrix = return_matrix(\@sequences, 3);
```

問題 6 上記プログラムで，行列中の数値は度数そのもので，割合にはなっていません．これを割合にするようにプログラムを改良しましょう．下記の dips_w_matrix 関数は，重み行列$wm_ref(リファレン

ス) と表示開始位置$pos_from，表示終了位置$pos_to を受け取ると，その区間の重み行列を表示します.

```
sub disp_w_matrix($$$){
    my($wm_ref, $pos_from, $pos_to) = @_;

    printf("%3s", " ");
    for my $position ($pos_from..$pos_to){
        printf("\t%4d", $position);
    }
    print "\n";

    printf("%3s", "--");
    for my $position ($pos_from..$pos_to){
        printf("\t%4s", "----");
    }
    print "\n";

    foreach my $nuc ("a", "c", "g", "t"){
        printf("%3s", $nuc);
        for my $position ($pos_from..$pos_to){
            my $val = $wm_ref->{$position}->{$nuc};
            if(!defined($val)){ $val = 0; }
            printf("\t%.2lf", $val);
        }
        print "\n";
    }
    print "\n";
}
```

下記の呼び出しで，重み行列%m の位置-1 から 2 までの内容が表示されます.

```
disp_w_matrix(\%m, -1,2);
```

1.5 シグナル配列の探索

解説 今度は 1 塩基ごとではなく，複数の塩基配列パターンとして特異的なシグナルを検出する方法を学びます.

```
ac agg tagtatgact
caa agg tcaatgaac
ta agg ctcgatgaga
ta agg catcatgcga
agc agg ctgatgaga
```

-7 の位置の ”agg ”の出現度数は 3，出現割合は 3/5=0.6，-6 の位置の ”agg ”の出現度数は 2，出現割合は 2/5=0.4 になります．このように与えられた配列と位置，パターンに対して，度数やその

割合を求める関数を作成しましょう.

```
sub calc_freq($$$){
   my($seq_set_ref, $pos, $pat) = @_;
   # $seq_set_ref 塩基配列群へのリファレンス
   # $pos 求める位置．但し，塩基配列群の先頭の位置を 0 とする
   # $pat 出現頻度を求めるパターン

   my $total = 0; # 解析対象となった配列の数
   my $count = 0; # 配列パターン$pat が位置$pos に検出された配列数

   foreach my $seq (@$seq_set_ref){
      if($pos + length($pat) - 1 < length($seq) &&
         substr($seq, $pos, length($pat)) =~ /^[acgt]+$/){
         # $pos が配列の長さを超えないかの確認と,
         # a,c,g,t 以外の塩基が含まれないかの確認
            $total ++;
            if(substr($seq, $pos, length($pat)) eq $pat){
               $count ++; # 配列パターンが$pat にマッチ
            }
      }
   }

   return($count, $total);

}
```

上記関数を例えば

```
my @sequences = (
   "acaggtagtatgact",
   "caaaggtcaatgaac",
   "taaggctcgatgaga",
   "taaggcatcatgcga",
   "agcaggctgatgaga");
my($count, $total) = calc_freq(\@sequences, 3, “ agg ”);
```

のように呼び出すと，3 番目の位置に ”agg ”というパターンが出現する回数が数えられ，$count に 2，$total に解析対象となった配列数 5 が入ります．$count / $total は出現割合を表します.

実際にはどのパターンがどの位置で顕著に出現するかは分かりません．そこで決められた範囲内における決められた長さの塩基配列パターンの頻度を全て調べる必要があるでしょう．次はある範囲内（下の例では先頭から 0 番目〜10 番目の位置）の特定のパターン（下の例では ”agg ”）の頻度をプロットしていくプログラムの一例です.

```
foreach my $pos (0..10){ # 調べる位置の範囲
   my($count, $total) = calc_freq(\@seq, $pos, "agg");
   print "$pos\t", $count / $total, "\n";
}
```

ハッシュを使うと，全パターンを 1 度に網羅することができます．下記の関数 calc_freq_all では，与えられた配列群$seq_set_ref の中の指定された位置$pos の決められた長さ$len の塩基配列の出現回数をハッシュ%count で数えます．

```
sub calc_freq_all($$$){
  my($seq_set_ref, $pos, $len) = @_;
  # $seq_set_ref 塩基配列群へのリファレンス
  # $pos 求める位置．但し，塩基配列群の先頭の位置を 0 とする
  # $pat 出現頻度を求めるパターン

  my $total = 0; # 解析対象となった配列の数
  my %count; # 配列パターンが位置$pos に検出された配列数

  foreach my $seq (@$seq_set_ref){
    if($pos + $len - 1 < length($seq) &&
       substr($seq, $pos, $len) =~ /^[acgt]+$/){
      # $pos が配列の長さを超えないかの確認と，
      # a,c,g,t 以外の塩基が含まれないかの確認
      $total ++;
      $count{ substr($seq, $pos, $len) } ++; # パターンの出現頻度を数える
    }
  }
  return(\%count, $total);
}
```

上記関数を例えば

```
my @sequences = (
   "acaggtagtatgact",
   "caaaggtcaatgaac",
   "taaggctcgatgaga",
   "taaggcatcatgcga",
   "agcaggctgatgaga");
my($count, $total) = calc_freq_all(\@sequences, 2, 3);
```

で，$count(この場合はリファレンスになります) に塩基配列の全パターンの出現回数を記録し，

```
print $count->{"agg"}, " \n";
```

で例えば ”agg ”の出現回数を出力することができます．

問題 7 決められた範囲 ($pos1 から$pos2) において，特定位置に最も出現する 4 塩基パターンと，その位置を求める関数を作成しましょう．

2 Perl による遺伝子発現データの解析

2.1 遺伝子発現データとは

遺伝子発現データは，各遺伝子がどのような時期または組織，条件で発現しているかを表した数値データです．そして同時期に発現している遺伝子のグループは，同一の機能を持っている可能性が高いことが知られています．同時期に発現しているかを測る指標が相関係数です．

今遺伝子 1 の発現データ $x_{11}, x_{12}, x_{13}, \ldots, x_{1n}$ と，遺伝子 2 の発現データ $x_{21}, x_{22}, x_{23}, \ldots x_{2n}$ が与えられたとき，相関係数 r_{12} の計算式は以下の通りです．

$$\begin{cases} r_{12} = \frac{1}{n}\sum_{j=1}^{n} z_{1j} z_{2j} \\ z_{ij} = \frac{x_{ij} - \overline{x}_i}{\sigma_i} \end{cases}$$

ここで，$\overline{x}_1$ は遺伝子 1 の発現量の平均，$\overline{x}_2$ は遺伝子 2 の発現量の平均を表します．また，σ_1 は遺伝子 1 の発現量の標準偏差，σ_2 は遺伝子 2 の発現量の標準偏差を表します．つまり平均と標準偏差を求めることができれば，相関係数を求めることができるのです．

2.2 平均の計算

解説 まず与えられた配列の平均値を計算する関数を実装してみましょう．

```
sub average {
   my @data = @_;
   my $ndata = $#data + 1;
   my $total = 0;

   # @data の合計を$total に入れる処理

   return $total / $ndata;
}
```

以下の文で関数の動作確認ができます．2.5 と表示されれば成功です．

```
my @x1 = (1.0, 2.0, 3.0, 4.0);
print average(@x1);
```

問題 8 平均を求める関数 average を実装しましょう．

2.3 標準偏差の計算

解説 標準偏差の計算には次のような関数を作ります．

```
sub sdev {
  my @data = @_;
  my $ndata = $#data + 1;
  my $average = average(@data);
  my $dev_total2 = 0;

  # (@data の各要素 - @data の平均)2 の合計を$total に入れる処理
   return sqrt($dev_total2 / $ndata);
}
```

以下の動作確認で 2 という答えが返ってくれば成功です.

```
my @x1 = (-1.0, -1.0, 3.0, 3.0);
print sdev(@x1);
```

問題 9 標準偏差を求める関数 sdev を実装しましょう.

2.4 相関係数の計算

二つの配列の読み込み

相関係数を計算するには，２つの配列を読み込む必要があります．配列が２つもあるので，リファレンスを使って配列を関数に渡してあげなければなりません.

リファレンスの受け取りと，配列の中味の確認は次のように行うことができます.

```
sub correlation {
   my $data1 = $_[0];
   my $data2 = $_[1];
   print join(“ , ”, @$data1), “ \n ”;
   print join(“ , ”, @$data2), “ \n ”;
}
```

またリファレンスを関数に渡すには,

```
my @x1 = (1,3,5,7,9);
my @x2 = (0,2,4,6,8);
correlation(\@x1, \@x2);
```

のようにします.

問題 10 配列のリファレンスの受け渡しがちゃんとなされているか，上記プログラムは打ち込んで確認しましょう.

欠損部分を排除

各遺伝子に関して常に発現データを取得することができるとは限りません．そこで欠損部部を取り除く必要があります.

@$data1 = (1, 2,0, ””,5); @$data2 = (0, ””,3, 6,7);	→	@data1m = (1,0,5); @data2m = (0,3,7);

この操作は次のように書くことができます．実際に打ち込んで動作を確認しましょう．

```
sub correlation {
  my $data1 = $_[0];
  my $data2 = $_[1];
  my @data1m;
  my @data2m;
  for my $i (0..$#$data1){
    if(defined($data1->[$i]) && $data1->[$i] =~ /\d/ &&
       defined($data2->[$i]) && $data2->[$i] =~ /\d/){
      push(@data1m, $data1->[$i]);
      push(@data2m, $data2->[$i]);
    }
  }
  print join(",", @data1m), "\n";
  print join(",", @data2m), "\n";
}
```

相関係数を求める

最後に相関係数を求める関数を完成させましょう．次のようになるはずです．

```
sub correlation {
  my $data1 = $_[0];
  my $data2 = $_[1];
  my @data1m;
  my @data2m;
  for my $i (0..$#$data1){
    if(defined($data1->[$i]) && $data1->[$i] =~ /\d/ &&
       defined($data2->[$i]) && $data2->[$i] =~ /\d/){
      push(@data1m, $data1->[$i]);
      push(@data2m, $data2->[$i]);
    }
  }

  my $corr = 0;
  my $ndata = $#data1m + 1;
  my $average1 = average(@data1m);
  my $average2 = average(@data2m);
  my $sdev1 = sdev(@data1m);
  my $sdev2 = sdev(@data2m);
  for my $i (0..$#data1m){
    $corr += 1.0/$ndata *
      ($data1m[$i] - $average1)*($data2m[$i] - $average2)
        / ($sdev1 * $sdev2);
  }
  return $corr;
}
```

遺伝子ID　遺伝子シンボル名

遺伝子の発現組織

Gene ID	Gene Symbol	Probe	Description	Thymus	cerebellum	WholeBrai	spinalcord	Lung	Heart	Liver	kidney
84694	CX62	gnf1h00843_at	connexin 62	127.0	487.0	126.0	84.0	385.0	82.0	99.0	274.0
7455	ZAN	gnf1h00040_at	zonadhesin	711.0	107.0	512.0	921.0	473.0	897.0	449.0	422.0
116337	PANX3	gnf1h01225_at	pannexin 3	432.0	537.0	695.0	342.0	358.0	488.0	579.0	615.0
149461	CLDN19	gnf1h06384_at	claudin 19	1127.0	1378.0	923.0	1161.0	1368.0	1340.0	1006.0	1102.0
57471	KIAA1189	gnf1h08650_at	KIAA1189	411.0	729.0	2433.0	9714.0	202.0	380.0	474.0	538.0
128854	TSPYL3	gnf1h01707_at	TSPY-like 3	222.0	29.0	218.0	164.0	122.0	47.0	246.0	154.0
10189	THOC4	gnf1h10382_at	THO complex 4	1541.0	76.0	119.0	95.0	1521.0	278.0	116.0	54.0
85439	STN2	gnf1h01078_at	stonin 2	417.0	307.0	83.0	699.0	225.0	390.0	291.0	355.0
388611	UNQ6122	gnf1h10827_at	EAPG6122	206.0	442.0	310.0	262.0	204.0	125.0	89.0	403.0
56666	PANX2	gnf1h01165_at	pannexin 2	45.0	42.0	25.0	45.0	25.0	34.0	30.0	25.0

84694		127.0	487.0	126.0	84.0	385.0	82.0	99.0	274.0
7455		711.0	107.0	512.0	921.0	473.0	897.0	449.0	422.0
116337		432.0	537.0	695.0	342.0	358.0	488.0	579.0	615.0
149461		1127.0	1378.0	923.0	1161.0	1368.0	1340.0	1006.0	1102.0
57471		411.0	729.0	2433.0	9714.0	202.0	380.0	474.0	538.0
128854	%exp	222.0	29.0	218.0	164.0	122.0	47.0	246.0	154.0
10189		1541.0	76.0	119.0	95.0	1521.0	278.0	116.0	54.0
85439		417.0	307.0	83.0	699.0	225.0	390.0	291.0	355.0
388611		206.0	442.0	310.0	262.0	204.0	125.0	89.0	403.0
56666		45.0	42.0	25.0	45.0	25.0	34.0	30.0	25.0

図 3.1: 遺伝子発現データファイルの構造例 (上) とハッシュによる管理 (下)

動作確認は以下のように行います．-1 という答えが出れば成功です．

```
my @x1 = (1,"",-3, 7,-7);
my @x2 = (0, 2, 4,"", 8);
print correlation(\@x1, \@x2);
```

問題 11 相関係数を求めるプログラムを完成させましょう．

2.5 遺伝子発現データのファイルからの読み込み

解説発現データファイルには一般的に図 3.1(上) のように、各時期・組織における発現量が遺伝子 ID・シンボル名やその遺伝子の機能情報とともに表形式で並べられています。そこで図 3.1(下) に示すように，遺伝子 ID が与えられたときに，対応する発現パターンを配列へのリファレンスで返すようなハッシュを作成します．以下のプログラムを打ち込んで，ハッシュによる遺伝子 ID と遺伝子発現パターンの対応付けができることを確認しましょう．

```
my %exp;
my @exp = (127.0, 487.0, 126.0, 84.0, 385.0, 82.0, 99.0, 274.0);
$exp{ "84694" } = [ @exp ];
print join(",", @{$exp{ "84694" }});
```

次に，与えられた発現データファイルより，遺伝子 ID とその発現パターンを上記のようにハッシュで結びつける関数を書きましょう．各行を@r に読み込んだら、shift で先頭の情報を抜き出し、残りを発現情報としてハッシュに登録します．

```
sub read_expression {
  my $filename = $_[0];
  local *FH;
  my %exp;
  open(FH, $filename);
  my $header = <FH>;
  while(<FH>){
    chomp;
    my @r = split(/\t/);
    my $gene_id     = shift @r;  # 遺伝子 ID の読み込み
    my $gene_symb   = shift @r;  # 遺伝子シンボルの読み込み
    my $probe_id    = shift @r;  # マイクロアレイのプローブ ID の読み込み
    my $description = shift @r;  # 遺伝子に関する情報
    $exp{ $gene_id } = [ @r ];
  }
  close FH;
  return %exp;

}
```

問題 12 図 3.1 のデータをタブ区切りのファイルにしてからそれをプログラムで読み込み、”84694(CX62)”と”85439(STN2)”の遺伝子発現パターンの相関係数を求めましょう.

問題 13 図 3.1 で相関係数が 0.2 以上の遺伝子のペアを列挙しましょう.

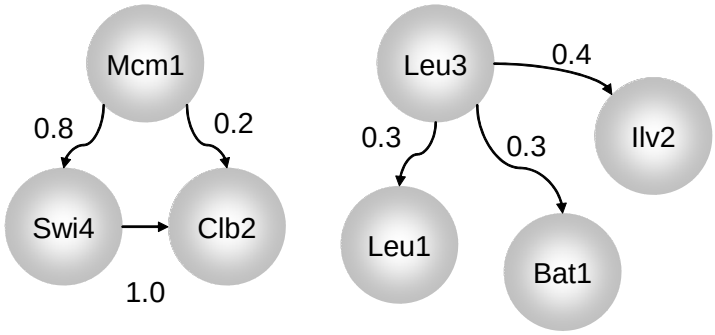

図 3.2: 転写制御ネットワークの例

3 分子間ネットワークの解析

3.1 Perl によるグラフ表現

解説 細胞内では遺伝子やタンパク質、代謝物質など様々な因子が相互作用して生命現象を起こしています。分子間のつながりの集合のことを分子間ネットワークと呼びます。分子と分子の間のつながりは数学ではグラフを用いて扱うことができます。Perl ではグラフをハッシュおよびハッシュへのリファレンスを用いるとうまく表現できます。例えば図 3.2 では、Mcm1 が Swi4 の転写制御をしていますが、この Mcm1 → Swi4 という関係はハッシュ(以下の例ではハッシュ%pdi がグラフを表す) とそのリファレンスを用いて

```
my %pdi = (
  "Mcm1" => { "Swi4" => "" }
);
```

と表すことができます。つまり%pdi のキーを"Mcm1"、値を { "Swi4" => "" }にしていることになります。直感的に説明するならば、”Mcm1”を入れるとそのターゲットがハッシュによって検索できるのでこういう表現が適切、というところでしょう。実は””にはつながりの重み (エッジの重み) を記すことができますので、Mcm1 → Swi4 の重みが 0.8 であれば、

```
my %pdi = (
  "Mcm1" => { "Swi4" => 0.8 }
);
```

と表すことができます。さらに図 3.2 の重みに従って Mcm1 → Swi4 と Mcm1 → Clb2、Swi4 → Clb2 を表現したい場合は、

```
my %pdi = (
  "Mcm1" => { "Swi4" => 0.8, "Clb2" => 0.2 },
  "Swi4" => { "Clb2" => 1.0 }
);
```

とします。そして例えば後から Leu3 → Leu1 を%pdi に追加したい場合は、

```
$pdi{ "Leu3" }->{ "Leu1" } = 0.3;
```

とします。->は省略可能です。

問題 14 図 3.2 の重みに従ってハッシュ%pdi にさらに、Leu3 → Bat1 と Leu3 → Ilv2 を追加しましょう。

問題 15 図 3.2 を表した以下のファイルの内容からネットワーク構造を%pdi に読み込むスクリプトを書きましょう。

```
Mcm1   Swi4   0.8
Mcm1   Clb2   0.2
Swi4   Clb2   1.0
Leu3   Leu1   0.3
Leu3   Bat1   0.3
Leu3   Ilv2   0.4
```

3.2 基本的なネットワーク解析

解説 3.1 節でハッシュ%pdi にグラフを実装できました。ここではそれを用いた簡単な解析を行いましょう。転写制御ネットワーク解析でまず気になるのは、各遺伝子がどの遺伝子を制御しているかということでしょう。これは keys を使うと簡単に知ることができます。例えば、Leu3 が制御している遺伝子は、

```
print join(",", keys(%{$pdi{"Leu3"}})), "\n";
```

で得ることができるのです。これは次のように分解して考えることができます。まず、`$pdi{"Leu3"}` は`{ "Leu1" => 0.3, "Bat1" => 0.3, "Ilv2" => 0.3 }`へのリファレンスです。従ってそのキーを表す `keys %$pdi{"Leu3"}` は、制御対象である `("Leu1", "Bat1", "Ilv2")` になるのです。ネットワーク中の全ての制御関係 (重みを含む) を列挙したい場合は、以下のようにすることができます。

```
for my $m1 (keys %pdi){
  for my $m2 (keys %{$pdi{$m1}}){
    print "$m1\t$m2\t$pdi{$m1}->{$m2}\n";
  }
}
```

問題 16 各遺伝子が制御している他の遺伝子の数を表示するスクリプトを書きましょう。図 3.2 を入力とした場合、以下のような出力が期待されます。

```
Mcm1   2
Swi4   1
Leu3   3
```

4 Perl による GenBank ファイルの解析

GenBank は米国の NCBI(国立バイオテクノロジー情報センター) が管理している膨大な量の塩基配列・アミノ酸配列情報を搭載した遺伝子データベースです。GenBank ファイル[1]には塩基配列はもちろんのこと、どの部分がタンパク質をコードしているかなど、それらの配列に付随する有益な情報が載せられており、プログラムを使った網羅的な解析をする上で大変有用です。そこでこの節では GenBank ファイルの処理の基本を学びます。

4.1 GenBank ファイルの構造

解説 GenBank のファイルは基本的に次のような構造になっています.

```
LOCUS エントリー名
そのエントリーの塩基配列に関する情報
                    :
                    :
ORIGIN
そのエントリーの塩基配列
                    :
                    :
//
LOCUS エントリー名
そのエントリーの塩基配列に関する情報
                    :
                    :
ORIGIN
そのエントリーの塩基配列
                    :
                    :
//
LOCUS エントリー名
                    :
                    :
そのエントリーの塩基配列
                    :
                    :
//
```

各塩基配列はエントリーという単位で格納されています．そして各エントリーは塩基配列の情報を記述するヘッダ部分 (LOCUS から ORIGIN の間) と実際の塩基配列の情報が書かれている部分 (ORIGIN から//の間) に分かれています．またエントリー中の記述の意味の例を次のページに示しました。霊長類，脊椎動物などの GenBank ファイルには複数のエントリーが含まれています．しかし，大腸菌の全ゲノム配列などの GenBank ファイルには 1 つのエントリーしか含まれていません．通常配列解析を行う場合は，GenBank 中の各エントリー中の塩基配列と塩基配列に関する情報を照らし合わせながら，解析を行います.

4.2 初歩的な塩基配列処理

問題 17 では解析のごく初歩的な例として，各エントリーの配列の先頭と末端 5 塩基を表示するプログラムを作成しましょう．次のエントリーをそのプログラムの入力とすると，gaatt, ctatg が出力される

[1] ftp://ftp.ncbi.nih.gov/genbank/よりダウンロード可能。

はずです.

```
LOCUS       HSAPC3B       558 bp    RNA             PRI       10-NOV-1994
DEFINITION  Human mRNA for pre-apolipoprotein CIII. ←この配列はどういうものか、
ACCESSION   X01388                                    その定義が記述されている
NID         g28727
KEYWORDS    apolipoprotein; lipoprotein; signal peptide.
SOURCE      human.
  ORGANISM  Homo sapiens ← 生物名
            Eukaryotae; mitochondrial eukaryotes; Metazoa/Eumycota group;
            Metazoa; Eumetazoa; Bilateria; Coelomata; Deuterostomia; Chordata;
            Vertebrata; Gnathostomata; Osteichthyes; Sarcopterygii; Choanata;
            Tetrapoda; Amniota; Mammalia; Theria; Eutheria; Archonta;
            Primates; Catarrhini; Hominidae; Homo.
REFERENCE   1  (bases 1 to 558)
  AUTHORS   Levy-Wilson,B., Appleby,V., Protter,A., Auperin,D. and
            Seilhamer,J.J.
  TITLE     Isolation and DNA sequence of full-length cDNA for human
            preapolipoprotein CIII
  JOURNAL   DNA 3 (5), 359-364 (1984)
  MEDLINE   85076166
FEATURES             Location/Qualifiers
     source          1..558
                     /organism="Homo sapiens"
     CDS             72..371  ←コード領域
                     /codon_start=1              ↓コードされているタンパク質名
                     /product="pre-apolipoprotein CIII"
                     /db_xref="PID:g28728"       ↓コードされているアミノ酸配列
                     /translation="MQPRVLLVVALLALLASARASEAEDASLLSFMQGYMKHATKTA
                     KDALSSVQESQVAQQARGWVTDGFSSLKDYWSTVKDKFSEFWDLDPEVRPTSAVAA"
     sig_peptide     72..131
     mat_peptide     132..368
                     /product="apolipoprotein CIII"
     misc_feature    532..537
                     /note="polyA signal"

BASE COUNT      107 a    175 c    143 g    133 t
ORIGIN  ↓ここから下が実際の塩基配列 (下線部分が CDS)
        1 gaattctttt ttttttttt tttgttgctc agttcatccc tagaggcagc tgctccagga
       61 acagaggtgc catgcagccc cgggtactcc ttgttgttgc cctcctggcg ctcctggcct
      121 ctgcccgagc ttcagaggcc gaggatgcct cccttctcag cttcatgcag ggttacatga
      181 agcacgccac caagaccgcc aaggatgcac tgagcagcgt gcaggagtcc caggtggccc
      241 agcaggccag gggctgggtg accgatggct tcagttccct gaaagactac tggagcaccg
      301 ttaaggacaa gttctctgag ttctgggatt tggaccctga ggtcagacca acttcagccg
      361 tggctgcctg agacctcaat accccaagtc cacctgccta tccatcctgc gagctccttg
      421 ggtcctgcaa tctccagggc tgcccctgta ggttgcttaa aagggacagt attctcagtg
      481 ctctcctacc ccacctcatg cctggccccc ctccaggcat gctggcctcc caataaagct
      541 ggacaagaag ctgctatg
//
```

まず $seq という変数に塩基配列を格納していき，格納し終わった段階で $seq の先頭と末尾の 5 塩基を表示するという方法を考えましょう.
塩基配列が格納されている部分は ORIGIN から//の間です. 実は ORIGIN から//の間の文字を取り出すという方法を使った方が正確ですが，これについては後ほど触れることにして，ここでは塩基配列が書かれている行が,

空白 数字 空白 塩基配列

という順番で並んでいることを利用しましょう．これは正規表現で表すと，

```
/^ *[0-9]+ [a-z]/
```

になります．次にこの条件に合致する行から塩基配列を取り出して格納しておく必要があります．まず，塩基配列以外の文字（数値，空白など）を，

```
$_ =~ s/[^a-z]//g;
```

によって取り除き，次に塩基配列格納変数$seq に以下のような文を使って塩基配列を追加していきます[2]．

```
$seq .= $_;
```

そして//という行を見つけたら，$seq の先頭と末尾の 5 文字を substr や length を使って表示します．substr 関数は，

```
$substring = substr($string, 0, 3);
```

のように（文字列, 切り出す場所, 切り出す文字数）という引数を与えて使います． なお，Perl では先頭の文字は 1 番目ではなく 0 番目から始まることに注意して下さい．また先頭に LOCUS という文字列が現れたら，新しいエントリーの始まりなので，

```
$seq = “ ”;
```

として中味を空にします.

```
#!/usr/bin/env perl

use strict;    # 全ての変数を my で宣言するように強制する
use warnings;  # プログラム動作に不合理な点があったときに警告を出す

local(*FILE);
my $seq = "";

open(FILE, $ARGV[0]) || die "Cannot open \"$ARGV[0]\": $!";
while(<FILE>){
    if($_ =~ /^LOCUS/){ $seq = ""; }
    elsif($_ =~ /^ *[0-9]+ [a-z]/){
            $_ =~ s/[^a-z]//g;
            $seq .= $_;
    }
    elsif($_ =~ /^\/\//){
            #ここは考えましょう
    }
}
close FILE;
```

このプログラムのファイル名を head_tail_seq.pl、処理の対象となる GenBank ファイルを test.seq[3] とすると、chmod +x ./head_tail_seq.pl で実行権を与えた後、

```
./head_tail_seq.pl test.seq
```

と打ち込みます。

[2] これは$seq = $seq . $_と意味は同じですが、多くの処理系では本文中の記述の方が高速な処理になります。

[3] GenBank のサンプルファイルは http://www.bioinfo.sfc.keio.ac.jp/class/genpro/Seqs/test.seq よりダウンロード可能。このファイルには複数のエントリーが含まれています。

4.3 ゲノムの GC 含有量の解析

解説 GC 含有量[4]は核酸配列の性質を調べる上で基本的な指標で、以下のように定義されます。

$$\text{GC 含有量} = \frac{\text{対象とする核酸配列に含まれる C と G の数}}{\text{対象とする核酸配列の長さ}}$$

例えば、ACTCAATGAG という核酸配列の GC 含有量は 40%になります。核酸配列の GC 含有量には以下のような生物学的意義や応用があります。

- 核酸の立体構造上の安定性、二次構造のできやすさ
- ゲノム全体の塩基組成
- ゲノム上の遺伝子領域の予測
- ゲノム上の外来性領域の予測

問題 18 大腸菌のゲノム配列[5]上の GC 含有量の推移を計算しましょう。また計算結果を Microsoft Excel などを用いて、図 3.4 のようなグラフにしましょう。

まず与えられた核酸配列の GC 含有量を計算する関数 calc_gc を以下のように作成します。

```
sub calc_gc {
  my $seq_frag = $_[0];
  my $gc_count = $seq_frag =~ tr/cg/cg/;
  return 1.0 * $gc_count / length($seq_frag);
}
```

ここで、$gc_count = $seq_frag =~ tr/cg/cg/; は$seq_frag 中に含まれる c や g の数を数え[6]、それを$gc_count に代入します。これを配列の長さ length($seq_frag) で割れば GC 含有量が計算できます。このとき、1.0 を掛けて小数演算であることを明示します。この関数を

```
print calc_gc("acgta");
```

のように呼び出すと、0.4 が表示されるでしょう。さて次に塩基配列の読み込みですが、問題 17 では塩基配列が書かれている行のパターンマッチを行ってした。しかしここではより厳密に、次のように"ORIGIN"以下の行から"//"の部分までの塩基配列を読み込み、それを返す関数 get_sequence を作成します。

[4]GC 含量、GC content とも呼ばれます。
[5]大腸菌のゲノム配列は http://www.bioinfo.sfc.keio.ac.jp/class/genpro/Seqs/ecoli.gbk よりダウンロード可能。
[6]厳密にいうと、c を c に g を g に変換してその変換数を返します。これについては後で詳しく説明します。

```
sub get_sequence {

  my $fh = $_[0];          # ファイルハンドルを$fh へ
  my $seq = "";
  my $seq_frag;

  while(<$fh>){            # 塩基配列の行を一行読み込む
    if($_ =~ /^\/\//){     # //を見つけたらそこで終了
      last;
    }
    else {
      $_ =~ s/[^a-z]//g;  # 塩基配列の行に含まれる数字、空白などを削除する
      $seq .= $_;          # $seq に配列全体を格納
    }
  }
  return $seq;
}
```

この関数は ORIGIN を見つけた後、

```
$seq = &get_sequence(*FILE); #ファイルハンドルを渡す
```

とファイルハンドルに*を付けて引数にして呼び出すことによって、塩基配列が$seq に代入されます。一般にファイルハンドルを関数の引数として渡すときは,

```
関数名（*ファイルハンドル名）;
```

で関数を呼び出し，受け取る関数では

```
my $ファイルハンドルを受け取る変数名 = $_[0];
```

で受け取ります.

さて次に GenBank ファイルを読み込んで、"ORIGIN"を見つけた後、get_sequence を呼び出す部分を作成します。

```
local *FH;
open(FH, $ARGV[0]) || die "Cannot open \"$ARGV[0]\": $!";
my $seq = "";
while(<FH>){
  if(/^ORIGIN/){
    $seq = get_sequence(*FH);
  }
}
close FH;
```

そして図 3.3 に示すように、ゲノム配列上にウィンドウを置き、ウィンドウ中の GC 含有量を計算し、ウィンドウを移動させるというステップを繰り返します。次のプログラムではウィンドウの大きさを 1000 塩基、移動幅を 500 塩基に設定しています。

大容量 GenBank 塩基配列の処理

関数 get_sequence は塩基配列がそれほど長くない場合に有効です．しかし，ヒトゲノムの GenBank ファイルなど塩基配列が数十億の長さになる場合は，マシンによっては処理が困難になってしまいます．そこでこのような場合は，塩基配列だけが書かれているファイルを次の関数を用いて作成します．

```
sub save_sequence {

    my($filename, $fh) = @_;
    my($seq_frag);
    local(*SEQFILE);

    open(SEQFILE, "> $filename");
    # $filename という名前のファイルを書きこみ用にオープンする

    while(<$fh>){
        if($_ =~ /^\/\//){
            last;
        }
        else {
            $_ =~ s/[^a-z]//g;
            print SEQFILE $_;
        }
    }

    close SEQFILE;

}
```

この関数を

```
&save_sequence(“ seq.tmp ”,*FILE);
```

のように呼び出します．すると塩基配列だけが書かれた”seq.tmp”というファイルが作成されます。そして塩基配列を読み込むときは、次のように SEQFILE を open し，seek 関数と read 関数を使って塩基配列の任意の場所を読み込むといいでしょう。

```
open(SEQFILE, "seq.tmp"); # 塩基配列だけが書かれたファイル"seq.tmp"をオープン
seek(SEQFILE, 100, 0);    # 塩基配列の 100 番目から読み込みを開始する
read(SEQFILE, $seq, 3);   # 3 文字読みとって、それを$seq に格納する
print "$seq\n";
close SEQFILE;
```

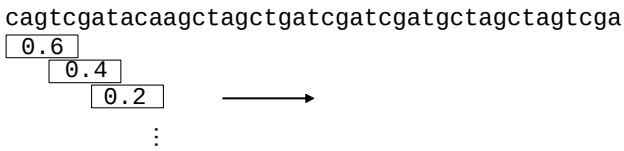

図 3.3: ウィンドウを使用したゲノム上の GC 含有量推移の計算

この例ではウィンドウの大きさを 5 塩基、移動幅を 3 塩基に設定した。ウィンドウの中の数値は GC 含有量を表す。

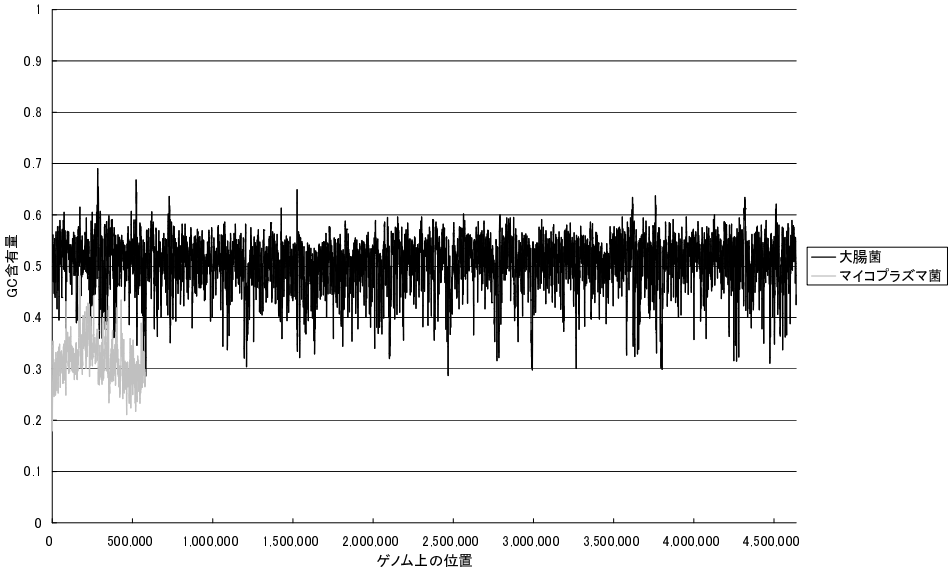

図 3.4: 大腸菌とマイコプラズマ菌のゲノム中の GC 含有量

```
my $win_size = 1000;
my $step = 500;
for(my $pos = 0;$pos + $win_size <= length($seq); $pos += $step){
  my $seq_frag = substr($seq, $pos, $win_size);
  my $gc_content = calc_gc($seq_frag);
  print "$pos\t$gc_content\n";
}
```

問題 19 GC skew はゲノム配列の片方の鎖の C の偏りを表す指標で以下の式で定義されます。

$$\text{GC skew} = \frac{\text{対象とする核酸配列に含まれる C の数} - \text{対象とする核酸配列に含まれる G の数}}{\text{対象とする核酸配列に含まれる C と G の数}}$$

ゲノム中の GC skew の推移を表すプログラムを作成し、実行結果を図にしてみましょう。大腸菌など生物によっては複製開始点を基点として GC skew に興味深い大きな変動が見られるでしょう。

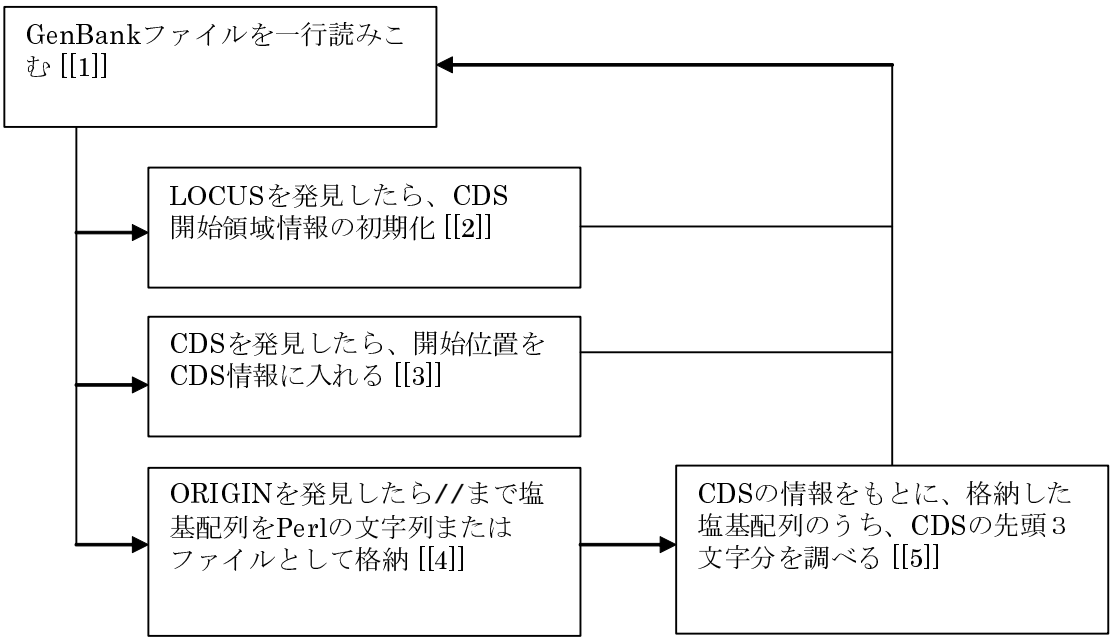

図 3.5: CDS 配列の処理の骨格

4.4 塩基配列に関する情報の格納とコード領域の解析

解説 次に CDS（protein-CoDing Sequence; タンパク質コード配列）が ATG で始まる割合を求める Perl プログラムを作成しましょう．基本的な処理を図 3.5 に示しますが、基本的な処理はファイルを一行ずつ読みながら，LOCUS, CDS, ORIGIN のキーワードに注目することです．すなわち、

1. LOCUS を見つけたら，エントリーが始まるので，様々な準備（初期化）をします.
2. CDS を見つけたら，CDS 開始領域の情報を格納します.
3. ORIGIN を見つけたら，塩基配列を変数（文字列）またはファイルに格納し，その後で CDS 開始領域の情報をもとに開始領域が ATG で始まっているか，判断します.

プログラムは次のような形になるでしょう.

```
#!/usr/bin/env perl

use strict;    # 全ての変数を my で宣言するように強制する
use warnings;  # プログラム動作に不合理な点があったときに警告を出す

# ここに後ほど説明する get_sequence, save_sequence などの関数を置く

my(@cds_start_set, $cds_start); # @cds_start_set に開始位置を記録していく
my($cds_count, $atg_count);     # CDS の数, atg で始まる CDS の数をカウント
open(FILE, $ARGV[0]) || die "Cannot open \"$ARGV[0]\": $!\n";

$cds_count = 0;
$atg_count = 0;

while(<FILE>){ # [[1]]
   chomp; # 行末端の改行記号を消す
   if($_ =~ /^LOCUS/){
      # [[2]]
   }
   elsif($_ =~ /^     CDS             ([0-9]+)\.\.([0-9]+)/){
      # [[3]]
   }
   elsif($_ =~ /^ORIGIN/){
      # [[4,5]]
   }
}

close FILE;

# 上記部分が完成したら、
# print "$atg_count / $cds_count = ", 1.0*$atg_count / $cds_count, "\n";
```

[[1]] はもうこれで完成していますから, [[2〜5]] を完成させましょう. まず上記プログラムをそのまま打ち込み（ファイル名は例えば cds_start.pl にする）,

```
./cds_start.pl test.seq
```

と打ちこんで実行してみましょう (ここで test.seq は GenBank ファイル名). 何も出力されないはずです. エラーが出るようならプログラムを修正しましょう.

問題 20 CDS の開始領域の位置の情報を@cds_start_set に格納しましょう（[[3]] の部分）. まず, grep コマンドを利用し, ” CDS ” の文字列を含んでいる行を検索しましょう. (ここでは%は Shell のプロンプトを意味します)

```
% grep " CDS" test.seq | less
```

次のように表示されるはずです.

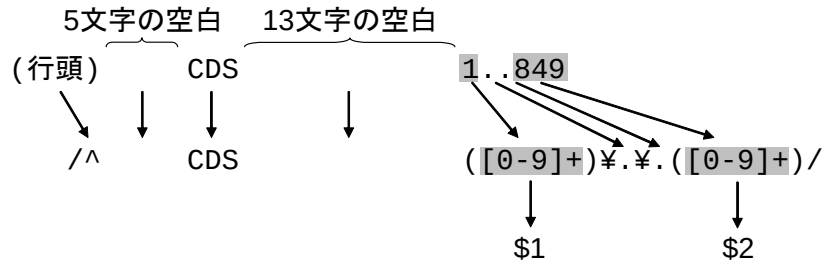

図 3.6: 正規表現による CDS 情報の処理

```
     CDS             join(3066..3157,3281..3503,4393..4521)
     CDS             466..>496
     CDS             1..849
     CDS             135..521
     CDS             9..695
     CDS             200..1444
     CDS             complement(1..297)
     CDS             64..444
     CDS             complement(116..403)
     CDS             1..90
     CDS             <1..>185
      :
```

今回は，"<"，">"，"join"，"complement"といった文字列を含む CDS は除外します[7]．つまり，格納すべき CDS の開始位置の情報は，上から順に，1，135，9，200，64，1，になります．
さて

```
     CDS             1..849
```

のような CDS の記述がある行を見つけたら，コード領域開始位置の 1 を取り出す処理が必要になります．問題は CDS　1..849 の 1 の部分をどうやって取り出すかですが，例えば 1 の部分をとりあえず$cds_start という変数に格納したい場合は，

```
   elsif($_ =~ /^     CDS             ([0-9]+)\.\.([0-9]+)/){
       $cds_start = $1;
                    :
                    :
   }
```

のようにするといいでしょう．GenBank ファイルの形式では CDS という文字列の前には 5 文字の空白、後には 13 文字の空白が入りますので，上記プログラムでもそれに従って正規表現に空白を入れます。図 3.6 に示すように，$1 は一番目の括弧内にマッチした正規表現，$2 は二番目の括弧内にマッチした正規表現という意味になります．したがって，$1 にはコード領域の開始の位置の情報，$2 にはコード領域の終わりの位置の情報（1..849 なら 849）が入ります．

そして，CDS の開始領域をどのように記録していくかを考えなければなりません．CDS は 1 つのエン

[7]"<"は位置がより前方にあることを表し、">"は後方を表します。また join は CDS が exon に分断されていることを表します。complement については後述します。

トリーに複数含まれている場合があるので，格納するのは配列変数にするのがいいでしょう．例えば格納する配列変数が@cds_start_set という名前だとすれば，まず 1 を@cds_start_set に格納します．次に，

```
     CDS             135..521
```

という行を見つけたら，135 をまた@cds_start_set に格納します．この時点で$cds_start_set[0] は 1，$cds_start_set[1] は 135 になるはずです．

さて，実際の格納の手順ですが，push 関数が有用でしょう．push 関数は配列の末尾に新しい要素を加える関数です．例えば，

```
@cds_start_set = (100, 200, 300);
```

のときに，

```
push(@cds_start_set, 1000);
```

とすると，@cds_start_set の要素は，(100，200，300，1000) になります．ちゃんと CDS 開始位置の情報が入っているか print 文などを使って確認しながらプログラムを作成しましょう．

問題 21 ORIGIN を見つけたら，まず塩基配列を文字列またはファイルとして格納しましょう（[[4]] の部分）．

塩基配列を文字列として格納するには，4.3 節で作成した関数 get_sequence を use strict の直後などに配置すればいいでしょう。そして [[4]] の適切な箇所で，

```
     $seq = &get_sequence(*FILE); #ファイルハンドルを渡す
```

のように呼び出すと，$seq には塩基配列が格納されます．

問題 22 塩基配列が入った文字列から@cds_start_set に基づいて CDS 開始領域の塩基配列をみていきましょう（[[5]] の部分）．

塩基配列を文字列変数としている場合は，例えば CDS の最初の 3 文字を表示したい場合は以下のようにするといいでしょう．

```
foreach $cds_start (@cds_start_set){
  print substr($seq, $cds_start - 1, 3), "\n";
}
```

あとは，$cds_count や$atg_count を使って CDS の数や atg の数を数えていくだけです．

問題 23 CDS 開始領域情報の初期化を行いましょう（[[2]] の部分）．

CDS 開始領域情報は@cds_start_set なので，

```
@cds_start_set = ();
```

でいいでしょう．

問題 24 問題 20〜23 を完成させ，コード領域（CDS）が ATG で始まる割合を求める Perl プログラムを作成しましょう[8]．

4.5 相補鎖の処理

解説 DNA は二重らせん構造をしています．そして A の反対側には T が，C の反対側には G が結合しています．反対側に結合している DNA 鎖を相補鎖と呼びます．

[8]真核生物の場合，CDS の開始はほとんど ATG です．ATG で始まらないエントリーは，CDS の一部の領域しか登録されていないか，CDS が偽遺伝子，データベースのエラーというものがほとんどです．原核生物では CDS が ATG から始まらない例が数多く知られています．

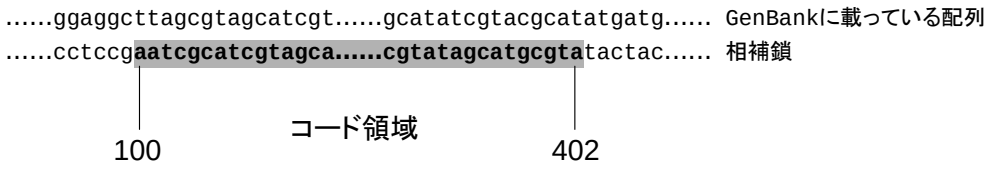

図 3.7: 相補鎖に存在するコード領域 (CDS)

```
……ctagctagctagtgcatgtga……
……gatcgatcgatcacgtacact……  ←相補鎖
```

遺伝子は相補鎖の方にコードされていることもあります．例えば図 3.7 の例では，灰色の部分に遺伝子がコードされています．この場合，GenBank の記述は以下のようになります．

```
     CDS             complement(100..402)
```

相補鎖を扱うときには，まず塩基配列を逆向きにして，次に A → T，T → A，C → G，G → C の変換を行わなければなりません．この相補鎖の扱いを課題を通して学びましょう．ここからはエントリー (LOCUS) が 1 つしかない、大腸菌の全ゲノム配列を扱います。

問題 25 大腸菌のゲノムの各コード領域（CDS）の塩基配列を切り出し，配列変数に格納してゆき，最後に通し番号とともに CDS 配列を出力しましょう．4.4 節で作ったプログラムのかなりの部分を流用することが可能でしょう。ただ前述の通り、この GenBank ファイルにはエントリー (LOCUS) は 1 つしかありませんので、全ゲノム配列を読み終わった後の処理は 1 回で済ますことができます。従って、[[5]] の部分を while 文の外に出してだいたい次のようにします。

```
sub get_sequence{

   # 関数 get_sequence の中味を書く

}

my @cds_start_set;   # @cds_start_set に CDS 開始位置を記録していく
my @cds_end_set;     # @cds_end_set に CDS 終了位置を記録していく
my $seq; # 塩基配列

open(FILE, $ARGV[0]) || die "Cannot open \"$ARGV[0]\": $!\n";

while(<FILE>){
  chomp; # 行末端の改行記号を消す
  if($_ =~ /^     CDS             ([0-9]+)\.\.([0-9]+)/){
    # CDS 開始および終了位置の記録。ここは考えましょう。
  }
  elsif($_ =~ /^ORIGIN/){
    $seq = &get_sequence(*FILE);
    last; # 塩基配列を全部読み終わったら、ファイルの読み取りを抜ける
  }
}

close FILE;
```

そして、続く [[5]] の部分では読み込んだ CDS の数 (`$#cds_start_set` がこれに相当します) だけ、CDS の配列を出力します。

```
for my $ncds (0..$#cds_start_set){  # CDS の通し番号と CDS 配列を出力していくループ
  my $cds_seq = substr($seq,
                       $cds_start_set[ $ncds ] - 1,
                       $cds_end_set[ $ncds ] - $cds_start_set[ $ncds ] + 1);
  print "$ncds\t$cds_seq\n";
}
```

問題 26 相補的な塩基配列を求める機能を先ほどのプログラムに組み込みましょう．先ほど説明した通り，'complement' というキーワードは，タンパク質をコードする DNA 塩基配列情報が相補鎖の方にあることを意味します．そこで，@complement という配列変数を用意し，'complement' があったら 1（真）を，なかったら 0（偽）を記憶しておきます．こうしておくと，後の処理で@complement を参照すれば，相補鎖に変換すべき CDS とそうでない CDS を識別できます．

```
while(<FILE>){
  chomp; # 行末端の改行記号を消す
  if($_ =~ /^     CDS             ([0-9]+)\.\.([0-9]+)/){
    # ここは考えましょう
    # @complement に 0 を追加
  }
  elsif($_ =~ /^     CDS             complement\(([0-9]+)\.\.([0-9]+)\)/){
    # ここは考えましょう
    # @complement に 1 を追加
  }
  elsif($_ =~ /^ORIGIN/){
    $seq = &get_sequence(*FILE);
    last;
  }
}
```

さらに，各 CDS の塩基配列を引数で渡すと，その相補鎖を返してくれる関数を作ると便利です．

```
sub complemental{
    my $seq = $_[0];
    my $complement;
    #ここは考えましょう; # 配列を逆順にする処理
    #ここは考えましょう; # A → T, G → C といったように相補塩基に変換する処理
    return $complement;
}
```

reverse は，文字列を逆順にしたものを返す関数です．例えば，$a = reverse("top"); としてやると，$a に"pot"が入ります．また，相補塩基に変換するには，tr/// を使うと便利です．tr/// は一文字置換の関数ですが，特定の文字がきたら特定の文字に，という複数の条件を一文で書き表すことができます．つまり，

```
$seq =~ tr/acgt/tgca/;
```

と書くだけで，a を t に，c を g に，g を c に，t を a に置換してくれます．あとは問題 25 を改良し，@cds_seq の中身を出力します．このとき@complement の値が 0 のときはそのままで，1 のときは complemental 関数にかけてから，出力します．

4.6 塩基の計数

解説 ここでは，大腸菌のゲノムにおける塩基組成を調べましょう．

問題 27 各 CDS における各塩基の度数を求めましょう．

問題 28 全 CDS における各塩基の度数（全 CDS の集計値）を求めましょう．

問題 29 全 CDS における各塩基の相対度数（総塩基数に対する各塩基の度数の比率）を求めましょう．

問題 27 では，a,c,g,t のそれぞれについて各 CDS の数をカウントします．
文字列操作関数である tr///は，s///とは違い，置き換えた文字数を返します．例えば，

```
$i = $str =~ tr/a-z/A-Z/;
```

とすると，$str 中の文字列を検索し，a-z 中に含まれる文字を見つけると，A-Z 中の対応する位置の文字に置換し（つまりアルファベットの小文字を大文字に置換し），置換した文字数を$i に代入します．これを利用して

```
$a = $seq =~ tr/a/a/;
```

と同じ文字を代入することにより，その文字の数をカウントすることができます．
生物種によっては，CDS 内に a,c,g,t 以外の文字が含まれるケースがあります．このようなケースでは，CDS の総文字数と a,c,g,t の総数とは一致しません．これは，

```
$others = length($seq) - ($a + $c + $g + $t);
```

とすることにより確認できます．length($seq) は，$seq の中の文字数を返します．

問題 28 では，a,c,g,t のそれぞれについて，全 CDS の数を集計します．
例えば，a の数を集計するには，for 文の中で以下の処理を行います．

```
$a += $seq =~ tr/a/a/;
```

問題 29 では，a,c,g,t のそれぞれについて，度数を比率に変換します．
小数の出力は，print の場合，小数点以下 8 桁くらいまで表示されて見にくいので，printf を使って小数第 3 位までの出力に整形しましょう．

```
printf("%.3f %.3f %.3f %.3f", $pr_a, $pr_c, $pr_g, $pr_t);
```

4.7 コドンの計数

解説 コード領域（CDS）の塩基配列は遺伝子発現時にコドンと呼ばれる 3 塩基の単位でアミノ酸配列に翻訳されます．ここでは，大腸菌ゲノムにおけるコドン組成を調べましょう．

問題 30 各 CDS における各コドンの度数を求めましょう．

問題 31 全 CDS における各コドンの度数（全 CDS の集計値）を求めましょう．

問題 32 全 CDS における各コドンの相対度数（総コドン数に対する各コドンの度数の比率）を求めましょう．

問題 30 では，各 CDS で使用されるコドンをそれぞれ計数します．
コドンは 3 連続塩基ですので 1 文字置換をする tr///を使って数えることはできません．複数置換の s///では置換された数を返さないのでこれを使うこともできません．仕方がないので，ここでは for 文を使って 3 文字づつ読んでいきましょう．ここでは塩基配列の最初から 3 文字の単位で読むのですから，例えば”atgcggctg”という塩基配列なら 1 つ目のコドンが”atg”，2 つ目のコドンが”cgg”，3 つ目のコドンが”ctg”というようになります．つまり，これは [1 文字目] から [配列の長さ-2 文字目] まで for 文で繰り返すことになります．ここで注意しなければいけないことは，Perl では先頭の番号は 1 ではなく 0 から始まる点です．つまり，for 文で繰り返す場合は [0 文字目] から [配列の長さ-3 文字目] までとなります．

```
for($p = 0; $p <= length($seq) - 3; $p += 3){
   # 処理
}
```

コドンを切り出すには，substr 関数を使います．

```
   $codon = substr($seq, $p, 3);
```

各コドンの計数にはハッシュをうまく利用しましょう．ハッシュの引数は文字列（この場合，1 コドン）でいいので，

```
   $count{$codon} ++;
```

としてやることで，$codon の中身が何であるかを気にせずにカウントすることができます．

問題 31 では，全 CDS 中の各コドンの数を集計します．
全 CDS の数を集計するには，全 CDS に対する処理を行う for 文の中で問題 30 の処理を行えばいいでしょう．

```
for my $ncds (0..$#cds_start_set){
   # 問題 30 の処理
}
```

問題 32 では，各コドンについて，度数を比率に変換します．
$total に全コドンの総度数を記憶しておけば，以下の処理で求めることができます．

```
foreach (keys %count){
    $proportion{$_} = $count{$_} / $total;
}
```

4.8 アミノ酸配列の処理

解説 CDS 領域の塩基配列が実際にどのようなアミノ酸配列になるかという情報は、CDS の注釈の中で translation というところに書かれています．ここでは，大腸菌のゲノムにおけるアミノ酸組成を調べましょう．

```
     CDS             complement(20815..21078)
                     /gene="rpsT"
                     /codon_start=1
                     /transl_table=11
                     /product="30S ribosomal subunit protein S20"
                     /translation="MANIKSAKKRAIQSEKARKHNASRRSMMRTFIKKVYAAIEAGDK
                     AAAQKAFNEMQPIVDRQAAKGLIHKNKAARHKANLTAQINKLA"
```

問題 33 各 CDS のアミノ酸配列を切り出し，その長さを出力しましょう．

まず/translation=””の中味を切り出すことを考えましょう．プログラムの骨組みは次のようになります．

```
while(<FILE>){
  chomp; # 行末端の改行記号を消す
  if($_ =~ /^     CDS             ([0-9]+)\.\.([0-9]+)/){
     # @cds_start, @cds_end および@complement に対する処理
  }
  elsif($_ =~ /^     CDS             complement\(([0-9]+)\.\.([0-9]+)\)/){
     # @cds_start, @cds_end および@complement に対する処理
  }
  elsif($_ =~/^                     \/translation=\"([^\"]+)/){
    $translation[ $#cds_start_set ] = $1;
    # 終わりまで translation のアミノ酸配列を読む
  }
  elsif($_ =~ /^ORIGIN/){
    $seq = &get_sequence(*FILE);
    last;
  }
}
```

まず/translation=”の行を検索し、アミノ酸部分を$1 に抽出するために以下の正規表現を用います。

```
  elsif($_ =~/^                     \/translation=\"([^\"]+)/){
```

次に、アミノ酸配列の 1 行目が$1 に格納されていますので、これを配列@translation に格納します。@translation の何番目に入れるかは、対応する CDS を@cds_start の何番目に入れたかということと一致させなければならないので、$#cds_start_set 番目と指定します。

```
    $translation[ $#cds_start_set ] = $1;
```

そして/translation の終わりまでアミノ酸配列を読み、@translation に追加していきます。このとき、行末のダブルクオテーション (”) が終わりの判定になります。

```
    while($_ !~ /\"$/){   # translation の終わりまで読む
      $_ = <FILE>;        # ファイルを一行読み込んで$_に入れる
      my $tmp = $_;       # 空白などを含むアミノ酸配列を一時的に$tmp へ代入
      $tmp =~ s/\"|\s//g; # ダブルクオテーションや空白などを消去する
      $translation[ $#cds_start_set ] .= $tmp; # アミノ酸配列を格納
    }
```

問題 34 CDS 毎に各アミノ酸の度数を求めましょう.

まず，@translation の各要素に文字列として格納されているアミノ酸配列を，1 文字づつに分割し，配列変数に代入します．文字列を分割するには，split 関数を使います.

```
split(/正規表現/, $variable);
```

split 関数は，指定した正規表現にマッチした文字列をセパレータとして，分割した文字列のリストを返します．以下のように，セパレータをヌル文字列（空文字列）にすれば，1 文字ずつに分割されます．例えば以下の例で$translation[$i] が ”SLQ ”の場合，@amino_seq は (“ S ”, “ L ”, “ Q ”) になります.

```
@amino_seq = split(//,$translation[$i]);
```

ここで，$translation[$i] は，i 番目の CDS のアミノ酸配列を意味します.

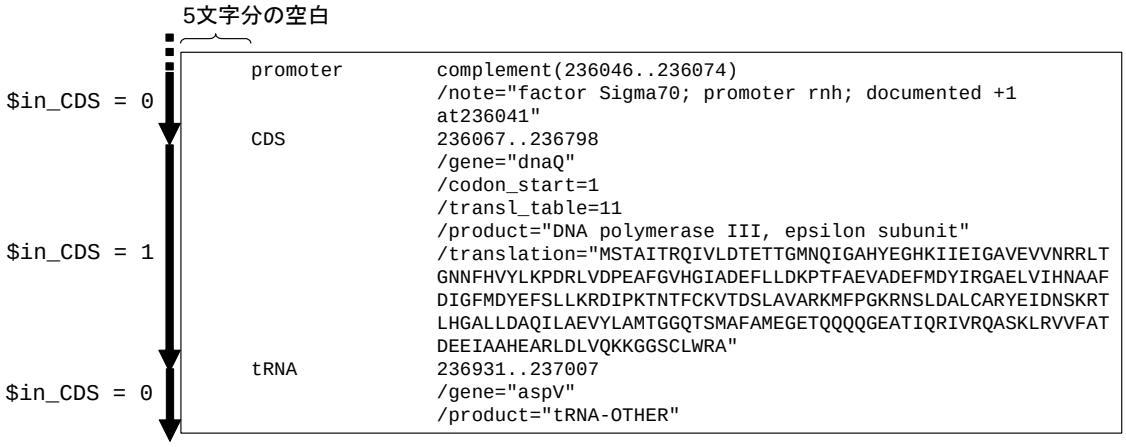

図 3.8: 変数$in_CDS による CDS の注釈領域の監視

次に，foreach 文において各アミノ酸の使用回数をカウントします.

```
foreach $amino (@amino_seq){
   $count{$amino} ++;
}
```

ここではハッシュをうまく利用しています．ハッシュの引数は文字列（この場合，1 アミノ酸）でいいので,

```
   $count{$amino} ++;
```

とすることで，アミノ酸$amino の数をカウントすることができます。

4.9 機能注釈情報の利用

解説 CDS の/gene には遺伝子名，/product には遺伝子産物（コードされているタンパク質）に関する情報が書かれています．例えば下の例では，gene 名は rpmF，product 名は 50S ribosomal protein L32 です.

```
     CDS             1146590..1146763
                     /gene="rpmF"
                    (中略)
                     /product="50S ribosomal protein L32"
```

これを利用することによって例えば特定のタンパク質をコードする配列のみを抽出することなどが可能です.

問題 35 大腸菌のゲノムにおける各コード領域（CDS）の gene 名と product 名を表示しましょう.

/gene=””および/product=””の中味を切り出します．問題 33 が参考になるでしょう．注意すべき点は、/gene や/product は CDS だけでなく、tRNA など他の遺伝子にも付随する注釈であるということです。従って CDS に付随する/gene や/product だけを読み取るように注意しなければなりません。そこで図 3.8 に示すように、$in_CDS という変数を定義し、CDS に付随する注釈の行を読み取っているときは 1、そうでないときは 0 にしておきます。0 → 1 の切換えのタイミングは、CDS という文

字列がファイルの 6 列目から出現したとき、1 → 0 に切換えるときは、CDS の注釈行が終わったときです。これはファイルの 6 列目に CDS という文字列以外の文字列が出現したときに相当します。プログラムの骨組みを次に示しましょう.

```
while(<FILE>){
  chomp; # 行末端の改行記号を消す
  if($_ =~ /^     CDS             ([0-9]+)\.\.([0-9]+)/){
    # @cds_start, @cds_end および@complement に対する処理
    $in_CDS = 1;   # CDS の注釈行の開始
  }
  elsif($_ =~ /^     CDS             complement\(([0-9]+)\.\.([0-9]+)\)/){
    # @cds_start, @cds_end および@complement に対する処理
    $in_CDS = 1;   # CDS の注釈行の開始
  }
  elsif($_ =~ /^     \S/){
    $in_CDS = 0;   # CDS の注釈行の終了
  }
  elsif($_ =~/^                     \/gene=\"([^\"]+)\"/ && $in_CDS == 1){
    $gene[ $#cds_start_set ] = $1;
  }
  elsif($_ =~/^                     \/product=\"([^\"]+)/ && $in_CDS == 1){
     # 処理を書きましょう
  }
  elsif($_ =~/^                     \/translation=\"([^\"]+)/){
     # 処理を書きましょう
  }
  elsif($_ =~ /^ORIGIN/){
    $seq = &get_sequence(*FILE);
    last;
  }
}
```

問題 36 リボソームタンパクをコードする遺伝子のみ表示されるように問題 35 のプログラムを改良しましょう.

product 名がリボソームタンパクを意味するものを正規表現で検索します.
まず，大腸菌の GenBank ファイルにおいて，リボソームタンパクがどのように記載されているのかを調べましょう．grep コマンドを利用し，ribosomal の文字列を含んでいる行を検索します．ワイルドカードを用いれば，複数の GenBank ファイル[9]を指定できます.

```
% grep ribosomal *.gbk | less
```

/product=”の行を見てください.
”ribosomal protein”，”ribosomal subunit protein”，”RIBOSOMAL PROTEIN”
など生物種によってリボソームタンパクの記述の仕方が異なります．これらにマッチするようにするためには例えば，以下のような正規表現を使うといいでしょう.

[9]全ゲノム配列を格納した GenBank ファイルの拡張子は通常.gbk である。

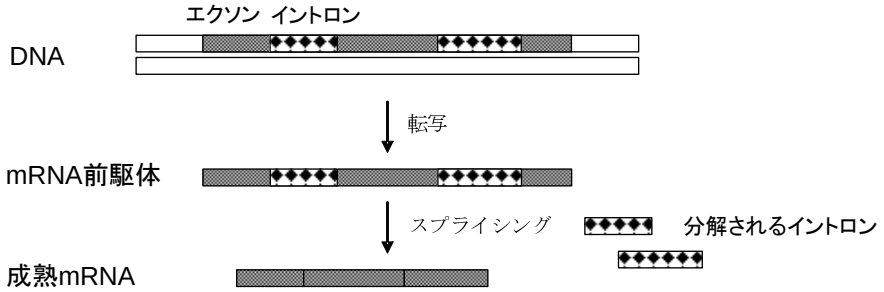

図 3.9: エクソン/イントロン構造

```
/ribosomal.*protein/i
```

.*は，どのような文字列にもマッチします．オプション i を付けると，パターンマッチのときにアルファベットの大文字と小文字を区別しません．

問題 37 リボソームタンパク遺伝子群における各アミノ酸の度数（集計値）を求めましょう．

@rp という配列変数を用意し，/product=""の中身がリボソームタンパクであれば 1（真）を，そうでなければ 0（偽）を記憶しておけば，リボソームタンパクをコードする CDS とそうでない CDS を識別できるので，後ほど@rp を参照することにより前者だけを対象として処理を行うことができます．

4.10 応用課題

解説 ＤＮＡの転写される部分で捨てられる（スプライスアウト）部分をイントロン (intron) といい，残る部分をエクソン (exon) といいます (図 3.9)．GenBank ではイントロンの場所の記述は

```
     exon            3066..3157
                     /gene="gamma-globin"
     intron          3158..3280
     exon            3281..3503
                     /gene="gamma-globin"
     intron          3504..4392
     exon            4393..4521
                     /gene="gamma-globin"
```

となっています．

問題 38 エクソン/イントロンの境界領域の配列を出力するプログラムを書きましょう．イントロン開始部位にはどのような配列が多く見られますか？霊長類の GenBank ファイルなどを調べてみましょう．

問題 39 イントロン/エクソンの境界領域の配列を出力するプログラムを書きましょう．イントロン終了部位にはどのような配列が多く見られますか？霊長類の GenBank ファイルなどを調べてみましょう．

第4章　Appendix

1　解答

第 1 章 Perl プログラミングを始めるまえに

第 2 章 Perl 入門

問題 1 print は文字列だけでなく，式の結果も出力することができます．なので解答としては以下で十分です.

```
print "123x456=";
print 123*456;
print "\n";
```

カンマ (,) を利用してまとめても良いでしょう.

```
print "123x456=", 123*456, "\n";
```

もちろん変数を使っても OK です.

```
$hoge = 123*456;
print "123x456=$hoge\n";
```

問題 2 イコール (=) が Perl では代入を意味することに気をつけましょう．後で出てきますが，等しいことを意味する記号はイコール x2(==) です

```
$x = 86400;
$y = 365;
$z = $x * $y;

print "$z\n";
```

問題 3 三行目で$x に$x+$y の結果を"改めて"代入しています.

```
$x = 10;
$y = 5;
$x = $x + $y;

print "$x\n"
```

この場合以下でも良いでしょう.

```
$x  = 10;
$y  = 5;
$x += $y;
print "$x\n"
```

問題 4 3 番目の要素は$array[2] でとってくることに注意．最後の要素にはカンマはつけません．

```
@array = (
 "Sunday",
 "Monday",
 "Tuesday",
 "Wednesday",
 "Thursday",
 "Friday",
 "Saturday"
 );

print "$array[2]\n";
```

問題 5 区切り文字が"-"であることに注意すれば OK です．

ちなみに split("pattern", sequence) のところでパターン指定の部分 (つまり第一引数) の囲いは""(ダブルクォーテーション) と//(スラッシュ) の二通りがあります．前者がパターンマッチで後者が正規表現です．

```
$seq = "Sun-Mon-Tue-Wed-Thr-Fri-Sat";
@week_array = split "-", $seq;

print "$week_array[2]\n"
```

問題 6 $i = $i+2 のところがミソです．

($i <= 11) のところは ($i < 12) 等でもかまいません．

```
my $i = 1;
while($i <= 11){
   print "There are $i sheep\n"
   $i = $i+2;
}
```

文法が気になる人はこうやっても良いでしょう．

```
print "There is a sheep\n"
my $i = 3;
while($i <= 11){
   print "There are $i sheep\n"
   $i = $i+2;
}
```

問題 7 "超えるまで出力"がポイントです．

```
my $i = 1/2;
while($i < 100000){
    $i = $i * 2;
    print "$i\n";
}
```

あるいは

```
my $i = 1;
while($i < 100000){
    print "$i\n";
    $i = $i * 2;
}
print "$i\n";
```

問題 8 変数を二つ使います.

```
my $i   = 1;
my $ans = 1;
while($i <= 15){
    $ans = $ans * $i;
    $i = $i + 2;
}
print "$ans\n";
```

問題 9 色々な方法がありますが--を使うとシンプルな解答になります.

```
for(my $i = 100; $i > 0; $i --){
    print "$i\n";
}
```

もちろんこれでも問題ありません.

```
for(my $i = 0; $i < 100; $i ++){
    print 100-$i, "\n";
}
```

foreach を使ってもできます.

```
print "$_\n" for reverse 1..100;
```

問題 10 [ ] の中に式が書けることを知っておくと後々役立ちます.

```
my @array = (
 "Sunday",
 "Monday",
 "Tuesday",
 "Wednesday",
  "Thursday",
 "Friday",
 "Saturday"
 );

foreach(1..30){
    print "$_\t$array[$_%7]\n";
}
```

あるいは

```
my @array = (
 "Sunday",
 "Monday",
 "Tuesday",
  "Wednesday",
 "Thursday",
 "Friday",
 "Saturday"
 );

for(@array){
    print "$_\t";
}
print "\n";

for(1..30){
    print "$_\t";
    print "\n" if $_%7 == 0;
}
```

以下は Perl らしいコードの例．解読してみてください．

```
my @array = (
 "Monday",
 "Tuesday",
  "Wednesday",
 "Thursday",
 "Friday",
 "Saturday",
 "Sunday"
 );

print "$_\t" for @array;
print "\n";
print  $_    for map{$_%7?"$_\t":"$_\n"}1..30;
```

問題 11 いわゆる静的なハッシュ作成．

```
%login = (
 matsui => "t03908mm",
 okada  => "t02176yo"
 );

print "matsui\t$login{matsui}\n";
print "okada\t$login{okada}\n";
```

問題 12 動的なハッシュ作成．エントリーをつけ足していくことができることから"動的".

```
$login{ando} = "t03053fa";
```

問題 13 基本的なハッシュエントリーの出力方法その 1.

```
foreach my $key (keys %login){
    print "$key\t$login{$key}\n";
}
```

$key の代わりに$_を使うと略すことができます．つまり

```
foreach $_ (keys %login){
    print "$_\t$login{$_}\n";
}
```

と

```
foreach(keys %login){
    print "$_\t$login{$_}\n";
}
```

は等価です.

問題 14 基本的なハッシュエントリー出力方法その 2.

```
while(my($key, $value) = each %login){
    print "$key\t$value\n";
}
```

keys を使う方法はまず全ての key を含む配列を作っていました．しかし key の数が膨大なときは効率が悪くなります．そのような時はエントリーを一つずつ取り出していくこちらの方法をとるべきでしょう.

問題 15 例文は以下のスクリプトと同じです.

```
#(1) 塩基配列を一文字ずつ分解し，@array という配列に入れる.
@array = split '', $seq

#(2) カウント
foreach my $elm(@array){
    $count{$elm} = $count{$elm} + 1;
}

#(3) 出力
foreach my $nuc(keys %count){
    print "$nuc\t$count{$nuc}\n";
}
```

(1) ではまず文字列をより処理しやすい配列に変換しています.

```
split '', $seq
```

とすることで文字列をヌルで切る，つまり一文字ずつ切り分けています.

次に (2) でカウントをしています．$count{}は%count の要素であることに注意．%count のエントリーにない key が来たときは value と共に新規に登録します．%count のエントリーの内容を 1 ループごとにたどってみるとよりはっきりするかもしれません.

(3) は出力です．key と value を順に出力しています．

問題 16 $line は毎回更新されていることに注意．ループを抜けたとき$line には最終行の文字列が入っています．

```
open FILE, "testfile";

while(<FILE>){
    $line = $_;
}

print $line;
```

問題 17 改行を除いた後，単にカンマを添えてプリントすれば OK です．

```
open FILE, "testfile";

while(<FILE>){
    chomp;
    print "$_,";
}
```

最後のカンマが気になる場合，.(ドット演算子) をうまく使うとよいでしょう．

```
open FILE, "testfile";

while(<FILE>){
    chomp;
    $seq .= "$_,";
}

chop $seq;
print "$seq\n";
```

chomp は文字列の最後の文字が改行 (\n) か否かを判定して改行なら切り取る演算子でした．それに対して chop は最後の文字を無条件に切り取る演算子です．print の前にこの関数を置いておくことで余計なカンマ (,) を除外しています．以下に示すのは chomp のイメージです．

```
sub chomp{
    $seq = shift;
    chop   $seq if substr($seq, -1, 1) eq "\n";
    return $seq;
}
```

問題 18 split を使って一行ずつ配列に分割し，配列の 1 番目→ 0 番目と表示すれば OK です.

```
open FILE, "fruit.txt";

while(<FILE>){
    chomp;
    @line = split;
    print "$line[1]\t$line[0]\n";
}
```

printf を使うと簡単になります.

```
open FILE, "fruit.txt";
printf "%s\t%s\n", (split)[0,1] while <FILE>;
```

問題 19 "ディレクトリの中にあるファイルを全て処理対象にする"，といった作業はよくでてきますのでマスターしておいてください．open の代わりに opendir と readdir を使っていることに注意．opendir と readdir はセットで使います.

```
$dir = "./";
opendir DIR, $dir;
@dir = readdir DIR;

foreach(@dir){
    $file = $dir.$_;
    open FILE, $file or next;
    while(<FILE>){
        print;
    }
}
```

問題 20 ||を使うと問題の意図を達成できます.

```
for(1..100){
    $rand   = int(rand(10)) || "ZERO";
    print "$rand\n";
}
```

ただし以下の式自体は 0〜9 の整数を返すことに注意してください (10 は返さない).

```
$rand = int(rand(10));
```

また

```
$hoge ||= "hoge";
```

で 0 や未定義値のチェックを行うことができます．割り算を含むプログラムなどではこのようなケアは必須です.

問題 21 a..z で a から z まで 26 文字のリストを表します.

```
foreach my $chr(a..z){
    $alphabet = $alphabet . $chr;
}
$alphabet = $alphabet x 10;
print "$alphabet\n";
```

短く書くとこんな感じ.

```
$alphabet .= $_ for a..z;
$alphabet x= 10;
print "$alphabet\n";
```

問題 22 大きさを比較すればよいので何らかの比較演算子を使うことになります. 普通の演算子を使うと以下のようになります.

```
$first  = int(rand 10);
$second = int(rand 10);

if( $first < $second ){
    print "$first\t<\t$second\n";
}
elsif($first > $second){
    print "$first\t>\t$second\n";
}
else{
    print "$first\t=\t$second\n";
}
```

<=>は左辺が大きければ 1 を右辺が大きければ-1 を等しければ 0 を返します. これを利用しても良いでしょう. 表示を上を揃えたければあらかじめハッシュを用意しておけば良いでしょう.

```
%kigou = (
    -1 => '<',
     0 => '=',
     1 => '>'
);
$first  = int(rand 10);
$second = int(rand 10);

$judge  = ($first <=> $second);
print "$first\t$kigou{$judge}\t$second\n";
```

問題 23 閏年のルール (グレゴリオ暦)

1. 4 で割り切れる年は閏年
2. しかし 100 で割り切れた場合閏年ではない
3. ただし 400 で割り切れたら閏年

例えば 2000 年は閏年でした.

素直にコーディングすればこうなるでしょう.

```
$year = 2004;

if($year % 4 == 0){
    if($year%400 == 0){
        print "leapyear\n";
    }
    elsif($year%100){
        print "not leapyear\n";
    }
    else{
        print "leapyear\n";
    }
}
else{
    print "not leapyear\n";
}
```

フラッグを使うという手もあります.

```
$year = 2004;

$flag = 0;
if($year % 4 == 0){
    $flag = 1;
    if($year%400 != 0 && $year%100 == 0){
        $flag = 0;
    }
}

if($flag){
    print "leapyear\n";
}
else{
    print "not leapyear\n";
}
```

条件をまとめるとこんな感じです.

```
if($year%400 == 0 || $year%4 == 0 && $year%100 != 0){
        print "leapyear\n";
}
else{
    print "not leapyear\n";
}
```

問題 24 まず

```
while(<FILE>){
    処理；
}
```

の原型は

```
while($_ = <FILE>){
    処理；
}
```

であることを確認しましょう.

<>は次の行を読み込む演算子です．例えば

```
hoge\
piyo
foo\
bar
```

という文があったときループごとに仕事を追ってくと

```
1.if に入る．次を読み込み今の行と結合.
  hogepiyo
  foo\
  bar
  となって redo によって改めて一行目の処理に向かう
2.if に入らず print
3. 二行目を読み込み if に入る．次を読み込み今の行と結合.
  hogepiyo
  foobar
  となる
4.if に入らず print
5. 次の行が無いのでループを抜ける
```

これは next 等では実装できません.

redo は以下のようなパターンのデータをパースするときに良く使います.

```
A-ab
B-c
  d
  e
C-f
↓
%data = (
 A => "ab",
 B => "cde",
 C => "f"
 );
```

そのパーサ例

```
#!/usr/bin/perl

open FILE, $ARGV[0];
while(<FILE>){
    if(/-/){
        $next = <FILE>;
        if($next =~ /-/ || !$next ){
            @line    = split '- ';
            $line[1] =~ s/[^a-zA-Z]//g;
            $data{$line[0]} = $line[1];
            $_   = $next;
            redo;
        }
        else{
            $_  .= $next;
            redo;
        }
    }
}

printf "%s\t%s\n", $_, $data{$_} for keys %data;
```

問題 25 先ほどの解答を単に構文に落とせば完成です.

```
($year%400 == 0 || $year%4 == 0 && $year%100 != 0)?
print "leapyear\n":
print "not leapyear\n";
```

三項演算子を重ねても良いでしょう.

```
($year%400 == 0)?print "leapyear\n":
($year%100 != 0)?print "not leapyear\n":
($year%4   == 0)?print "leapyear\n":
                 print "not leapyear\n";
```

if 文を使えばあらゆる条件分岐は表現できるので三項演算子は必須のものではないのですが, コードを見やすく短く済ませたいときに有効です.

問題 26 正規表現の典型的な問題です. まず atg があるかどうかを調べるときは

```
$seq = "agtgctagtcgtgtagctactacgtacgt";
if($seq =~ /atg/){
    print "ATG codon exists in \$seq";
}
else{
    print "ATG codon don't exists in \$seq";
}
```

です. 同様に終止コドン (tag/tga/taa) があるかどうかを調べるときは

```
$seq = "agtgctagtcgtgtagctactacgtacgt";
if($seq =~ /tag/ || $seq =~ /tga/ || $seq =~ /taa/){
    print "ATG codon exists in \$seq";
}
else{
    print "ATG codon don't exists in \$seq";
}
```

もう少し効率よくやるならば

```
$seq = "agtgctagtcgtgtagctactacgtacgt";
if($seq =~ /t(ga|a[ga])/){
    print "It matches.";
}
else{
    print "It doesn't match";
}
```

これらをあわせれば解答は完成です.

```
$seq = "agtgctagtcgtgtagctactacgtacgt";
if($seq =~ /atg/ && $seq =~ /t(ga|a[ga])/){
    print "It matches.";
}
else{
    print "It doesn't match";
}
```

ちなみに拡張正規表現を用いればこの解答は

```
$seq = "agtgctagtcgtgtagctactacgtacgt";
if($seq =~ /^(?=.*atg)(?=.*t(ga|a[ga]))/){
    print "It matches.";
}
else{
    print "It doesn't match";
}
```

となります. `(?=...)` は先読み条件です.

```
/foo(?=bar)/
```

で後ろに bar を伴う foo にだけマッチします. 先の解答はつまり行頭の後ろに何文字かおいて atg があるし, 同様に終止コドンもある, という条件を満たすことになります.

問題 27 後方参照の練習です.

```
@nuc = ('a', 't', 'c', 'g');
for(1..300){
    $seq .= $nuc[int(rand 4)];
}

while($seq =~ /([atgc]{6})atg([atgc]{6})/g){
    print "$1\t$2\n";
}
```

問題 28 略. リファレンス等を見てください.

問題 29 コード例は色々とあります.

```
sub my_reverse{
    my $seq = shift;
    my @seq = split //, $seq;
    my $reverse_seq;
    for(@seq){
        $reverse_seq = $_ . $seq;
    }
    return $reverse_seq;
}
```

とか

```
sub my_reverse{
    my $seq = shift;
    my @seq = split //, $seq;
    my $i = length $seq;
    my $reverse_seq;
    while(--$i >= 0){
        $reverse_seq .= $seq[$i];
    }
    return $reverse_seq;
}
```

あるいは

```
sub my_reverse{
    my $seq    = shift;
    my $length = length $seq;
    my $reverse_seq;
    for(1..$length){
        $reverse_seq .= substr $seq, $length-$_, 1;
    }
    return $reverse_seq;
}
```

問題 30 定義式をそのまま実装すれば OK です.

一本の配列のエントロピーを算出する関数

```
sub entropy{
    my $seq    = lc shift;
    my $length = length($seq)||1;
    my(%count, $H);
    $count{$_}++ for split //, $seq;
    for('a', 't', 'c', 'g'){
        $count{$_}  /= $length;
        $count{$_} ||= 1;
        $H -= $count{$_}*log($count{$_})/log(2);
    }
    return $H;
}

my $seq = 'aaaaggactt';
my $entropy = &entropy($seq);
print "entropy = $entropy\n";
```

複数の配列のエントロピーを算出する関数

```
sub entropy{
    my $seq_ref = shift;
    my $number  = scalar @{$seq_ref}||1;
    my(%count, @H, $i);
    for(@{$seq_ref}){
        $i = 0;
        ++$count[$i++]{$_} for split //;
    }
    for my $pos(0..$#count){
        for my $nuc('a', 't', 'c', 'g'){
            $count[$pos]{$nuc}  /= $number;
            $count[$pos]{$nuc} ||= 1;
            $H[$pos] -= $count[$pos]{$nuc} *log($count[$pos]{$nuc}) /log(2);
        }
    }
    return @H;
}
my @seq = (
              'ataatgaatt',
              'ccacgcacta',
              'gcaacgacat',
              'ctattcacct'
          );
my @entropy = &entropy(\@seq);
print "$_\t$entropy[$_]\n" for 0..$#entropy;
```

問題 31 リファレンスにして関数に引渡し，関数の中でデリファレンスできれば正解です.

```
sub my_reverse{
    my $seq = shift;
    my @seq = split //, $$seq;
    my $reverse_seq;
    for(@seq){
        $reverse_seq = $_ . $reverse_seq;
    }
    return $reverse_seq;
}

my $reverse_seq = &my_reverse(\$seq);
```

問題 32 リファレンスの動作を確かめる問題です.

```
@array      = (1, 2, 3, 4, 5);
$array_ref1 = \@array;
$array_ref2 = [@array];

print "$array_ref1->[2]\n";
print "$array_ref2->[2]\n";

$array[2] = 'x';

print "$array_ref1->[2]\n";
print "$array_ref2->[2]\n";
```

問題 33 ハッシュに格納してしまうのが手っ取りばやく，また効率の良い方法です.

```
for(@row){
    $uniq{$_} = 1;
}
for(keys %uniq){
    push @uniq, $_;
}
```

あるいは重複する要素を記録して言ってもよいでしょう.

```
for(@row){
    next if $uniq{$_}++;
    push @uniq, $_;
}
```

これらを短くすると

```
@uniq = keys %{{map{$_, 1}@row}};
```

または

```
@uniq = grep{!$exist{$_}}@row;
```

となります

問題 34 C の構造体 like なデータ構造を構築できます.

```
$seq   = 'atgacgtggtac';
@array = ('a', 'r', 'r', 'a', 'y');
%hash  = ('h' => 'a', 's' => 'h');
sub hello{
    print "Hello! $_[0]!\n";
}

$struct = {
    scalar => \$seq,
    array  => \@array,
    hash   => \%hash,
    sub    => \&hello
};
```

もちろん直接作ってもかまいせん.

```
$struct = {
    scalar => \'atgacgtggtac',
    array  => ['a', 'r', 'r', 'a', 'y'],
    hash   => {'h' => 'a', 's' => 'h'},
    sub    => sub{"Hello! $_[0]!\n"}
};
```

問題 35 ランダム配列の作り方としては以下の二つを覚えればよいでしょう.

```
i)
@nuc = ('a', 't', 'c', 'g');
$seq .= $nuc[int(rand 4)] for 1..300;

ii)
$seq .= int(rand 4) for 1..300;
$seq =~ tr/0123/atgc/;
```

あとはループと substr を使って順にコドンを抜き出せばよいでしょう.

```
for(0..299){
    $codon = substr $seq, $_, 3;
    $count[$_%3]{$codon}++;
}

for my $frame(0..2){
    print "### $frame ###\n";
    for(sort{$count[$frame]{$b} <=> $count[$frame]{$a}}
        keys %{$count[$frame]}){
        print "$_\t$count[$frame]{$_}\n"
    }
}
```

問題 36 index 関数の特徴をうまく利用してやると以下のように windex が実装できます.

```
sub windex{
    my($seq, $pat) = @_;
    my $pos = -1;
    my @all_position;
    while( ($pos=index($seq, $pat, $pos)) >-1){
        push @all_position, $pos;
        $pos++;
    }
    return @all_position;
}
```

あるいは正規表現//g を使ってもできます.

```
sub windex{
    my($seq, $pat) = @_;
    my @all_position;
    while($seq =~ /$pat/g){
        push @all_position, length $`;
    }
    return @all_position;
}
```

問題 37 tr をカウントに使います.

```
$seq   = 'atgtgctgtagctgatgctgatcggggcgcgatc';
$count = $seq =~ tr/gcGC/gcGC/;
```

あるいは

```
$seq   = 'atgtgctgtagctgatgctgatcggggcgcgatc';
$count = tr/gcGC/gcGC/ for $seq;
```

問題 38 各関数とも重要ですので，書式，動作を確かめておきましょう.

```
@array = ('a', 'b', 'c', 'd', 'e');
$first = shift @array;
$last  = pop   @array;
unshift @array, $last;
push    @array, $first;
```

問題 39 これらの関数のうち，pop と shift は与えられた配列を変更する機能の他，削られた要素を戻り値として返します．それらを実装するにはリファレンス演算子を用いたリファレンス渡し，及び return を用いた戻り値の設定が必要です.

```
sub my_unshift{
    my($array, $elm) = @_;
    splice @{$array}, 0, 0, $elm;
}

sub my_shift{
    my $array = shift;
    my $elm   = splice @{$array}, 0, 1;
    return $elm;
}

sub my_pop{
    my $array = shift;
    my $elm   = splice @{$array}, $#array, 1;
    return $elm;
}

sub my_push{
    my($array, $elm) = @_;
    splice @{$array}, $#{$array}+1, 0, $elm;
}
```

問題 40 (A, B) = (B, A) とすることで一時変数を使うことなく安全に値の交換ができます.

```
@foo = (1, 2, 3, 4, 5);
@foo[2, 4] = @foo[4, 2];
```

問題 41 上の問題で交換するペアを 0, -1 にかえれば完成です.

```
@foo = (1, 2, 3, 4, 5);
@foo[0, -1] = @foo[-1, 0];
```

問題 45 引数は@ARGV (ARGument Vector) に入っています. 個別に値をとりたいときは$ARGV[0] などとすればよいでしょう. また@ARGV 自体は変更しないようにしましょう

```
my @arg = @ARGV;
my $mean;
$mean += $_ for @arg;
$mean /=($#arg+1)||1;
print "$mean\n";
```

問題 46 last を使って最近のログインランキングを調べるプログラムは以下の通りになるでしょう.

```
my @last = `last`;
my %count;
chomp @last;
for(@last){
    my($name) = (split)[0];
    $count{$name}++;
}
for(sort{$count{$b} <=> $count{$a}}keys %count){
    printf "%s\t%d\n", $_, $count{$_};
}
```

第 3 章 実践 Bioinformatics

問題 47 二元配列がどのように行列を表しているか理解しておきましょう.

```
sub addition{ #足し算
    my($A, $B) = @_;
    my @A = @$A;
    my @B = @$B;
    my @C;
    for my $i(0..$#A){
        for my $j(0..$#{$A[0]}){
            $C[$i][$j] = $A[$i][$j]+$B[$i][$j];
        }
    }
    return @C;
 }
sub multiplication{ #かけ算
    my($A, $B) = @_;
    my @A = @$A;
    my @B = @$B;
    my @C;
    for my $i(0..$#A){
        for my $j(0..$#{$B[0]}){
            for my $elm(0..$#B){
                $C[$i][$j] += $A[$i][$elm]*$B[$elm][$j];
            }
        }
    }
    return @C;
}
my @A = ([1, 2],
         [3, 4]);
my @B = ([5, 6],
         [7, 8]);
my @ADD  = &addition(\@A, \@B);
my @MULTI = &multiplication(\@A, \@B);
for my $i(0..$#ADD){
    for my $j(0..$#{$ADD[$i]}){
        print "$ADD[$i][$j]\t";
    }
    print "\n";
}
for my $i(0..$#MULTI){
    for my $j(0..$#{$MULTI[$i]}){
        print "$MULTI[$i][$j]\t";
    }
    print "\n";
}
```

2 リファレンス

Perl 編

スカラー変数操作法

表 4.1: スカラー変数操作法

関数	説明	使用例
length	文字列の長さを返す	length 文字列
reverse	文字列を逆にする	reverse 文字列
index	文字列中のパターンの位置を返す	index 文字列, パターン, 捜査開始位置
substr	文字列から指定した位置の部分文字列を返す	substr 文字列, 開始位置, 長さ
s	置換	s/パターン/変換後の文字列/g
tr	変換	tr/abc/pqr/
lc	全て小文字に変換	lc 文字列
sprintf	文字列の整形	`sprintf` フラッグ，変数，…
chop	最後の文字を削る	chop 文字列
chomp	改行を削る	chomp 文字列
join	リストを結合して文字列を返す	`join` /区切り文字/，`@array`
split	文字列を区切り文字を目印に分割	split /区切り文字/, 文字列
abs	絶対値を返す	abs 数値
int	小数点以下を切り捨て	int 数値
rand	乱数を返す	rand 数値

配列変数操作法

表 4.2: 配列変数操作法

関数	説明	書式
shift	先頭の要素を削除	`shift @array`
unshift	先頭に要素を追加	`unshift @array, $scalar`
pop	末尾の要素を削除	`pop @array`
push	末尾に追加	`push @array, $scalar`
sort	リストの要素をソート	`@sorted = sort @mess`
reverse	リストを逆順にならべる	`reverse @array`
map	配列を新たに作成	`map{式}@array`
grep	配列の要素を選別	`grep{式}@ array`
splice	要素を置換	`splice @array`，位置，要素数，置換後のスカラー or 配列

ハッシュ変数操作法

表 4.3: ハッシュ変数操作法

関数	説明	使用例
keys	キーをリストにして返す	`@keys = keys %hash`
values	値をリストにして返す	`@values = values %hash`
each	キーと値のペアを一つずつ返す	`while(($key, $value) = each %hash){式}`
exists	キーが存在するか判定	`exists $hash{$key}`
delete	キーを削除	`delete $hash{$key}`
defined	値が真か判定	`defined $hash{$key}`
undef	値を初期化	`undef $hash{$key}`
reverse	キーと値を入れかえる	`reverse %hash`

正規表現

表 4.4: 正規表現

正規表現	書式
.(ドット)	ワイルドカード，全てにマッチ (スペース，ヌルを含む)
\d, \D	\d は整数にマッチ
\w, \W	\w は [a-zA-Z0-9_] にマッチ
\s, \S	\s はスペース or タブ or 改行にマッチ
a|b	a または b にマッチ
[abc]	a，b，c のどれかにマッチ
[^a]	a 以外にマッチ
[a-z]	a,b,…,z のどれかにマッチ
( )	正規表現のグループ化，同時に$1 などに格納
$1, \1	( ) にマッチした文字列を格納
^	行頭を意味する
$	行末を意味する
a?	a もしくはヌルにマッチ
a*	a の 0 回以上の繰り返し
a+	a の 1 回以上の繰り返し
a{m, n}	a の m 回以上，n 回以下の繰り返し
??, *?, +?	最短マッチ
\b	単語の境にマッチ
\記号	特別な意味のあるワードをエスケープする

printf の使い方

構文

```
printf "フラッグ", 第一項, 第二項,...;
```

主なフラッグ

`%s` 文字列

`%d` 整数

`%f` 浮動小数点表示 (`%.3f` で小数第三位まで表示)

`%e` 科学表記 (`%.3e` で仮数を三桁まで表示)

`\n` 改行

`\t` タブ

例文

```
printf "%\ts%.2f\n", 'pi ~ ', '3.14159';
```

→ pi ~ 3.14

関連図書

[1] Larry Wall *et al.* (2002) Programing Perl 第三版，O ’REILLY

[2] Hall, JN.，Schwarrtz, RL(1999) Effective Perl，アスキー出版局

[3] 藤田郁，三島俊司 (2004) Perl/CGI プチリファレンス，技術評論社

[4] Christiansen, T. *et al.* (2004) Perl クックブック第二版，O ’REILLY

索 引