BIOINFOMATICSPROGRAMMING

May 12 - June 9, 2006

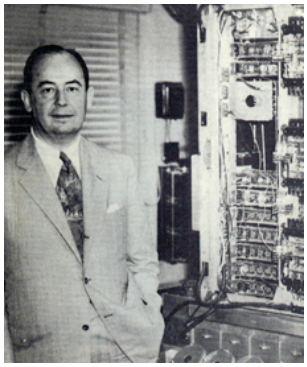
バイオインフォマティクスの研究に必須となるプログラミングの概要を学び、簡単なプログラムを作成できるようにする。また、生物学・化学のデータベースリソースを活用するための基礎的な実習を行う。

科学技術振興調整費人材養成プログラム「ゲノム情報科学研究教育機構」

プログラミング実習

By Toshiaki Katayama

ノイマン型計算機で知られる J. von Neuman は、計算機製造現場を訪れてこういったそうだ「君たち、いくら大問題を瞬時に計算できたところで、プログラムの設定に何日もかかっていては意味がない」と。



バイオインフォマティクスについても同じことが言えそうだ。生物学的な結果を得るために解くべき問題があり、それを解くためのアルゴリズムを考え、プログラムとして実装し、入力データを与えてプログラムを実行し結果を得る。この過程で時間のかかるポイントは３カ所あるが、プログラムを書く時間がボトルネックとなるのは最も惜しい。そのため、本実習では実装が容易で開発時間を短縮できる Ruby 言語による効率的なプログラミングを学ぶ。

残りの２点は、スパコンなどを利用してもプログラムの実行時間がボトルネックになる場合と、問題の解き方が不明でアルゴリズムの開発に時間がかかる場合である。前者は、その段階になってはじめてアルゴリズムの見直しや C 言語への移植などによる高速化を考えれば良い。後者は最も本質的に時間をかけるべき部分で、問題設定や解き方の開発自体に研究として取り組む必要があるだろう。

Rubyの概要

リソースの紹介

By Toshiaki Katayama

Rubyとは

Rubyのウェブページには、以下のように書かれています：

Rubyは、手軽なオブジェクト指向プログラミングを実現するための種々の機能を持つオブジェクト指向スクリプト言語です。本格的なオブジェクト指向言語である Smalltalk、EiffelやC++などでは大げさに思われるような領域でのオブジェクト指向プログラミングを支援することを目的としています。もちろん通常の手続き型のプログラミングも可能です。

Rubyはテキスト処理関係の能力などに優れ、Perlと同じくらい強力です。さらにシンプルな文法と、例外処理やイテレータなどの機構によって、より分かりやすいプログラミングが出来ます。

まあ、簡単にいえばPerlのような手軽さで「楽しく」オブジェクト指向しようという言語です。どうぞ使ってみてください。

Rubyはまつもと ゆきひろ(matz@netlab.jp)が個人で開発しているフリーソフトウェアです。

Rubyの特長

- シンプルな文法
- 普通のオブジェクト指向機能(クラス、メソッドコールなど)
- 特殊なオブジェクト指向機能 (Mix-in 、特異メソッドなど)
- 演算子オーバーロード
- 例外処理機能
- イテレータとクロージャ
- ガーベージコレクタ
- ダイナミックローディング(アーキテクチャによる)
- 移植性が高い。多くのUNIX上で動くだけでなく、DOSやWindows、Mac、BeOSなどの上でも動く

Ruby関連のウェブページ

Ruby言語のホームページは

http://www.ruby-lang.org/

になります。リリース情報やダウンロードの案内など、公式情報が掲載されています。

Rubyリファレンスマニュアル

Rubyのリファレンスマニュアルは以下のページで参照できます。

http://www.ruby-lang.org/ja/man/

マニュアルのコンテンツは Wikiで管理されており複数のボランティアにより維持されています。

るびま

日本Rubyの会が発行している、Rubyに関するフリーのウェブ雑誌「Rubyist Magazine (通称：るびま)」

http://jp.rubyist.net/magazine/

には、 入門用からマニアックなものまでクオリティの高い記事が多数掲載されています。

ReFe

ReFe は Ruby の日本語マニュアルをコマンドラインで引くためのツールです。配布サイト

http://i.loveruby.net/ja/prog/refe.html

から最新版をダウンロードし、インストールして使います。

ri

ri は ReFe の元となった英語版のマニュアル検索ツールです。Ruby 1.8 以降では標準添付されていますが、検索できるようにするためには rdoc コマンドでドキュメントを生成する必要があります。

RAA

Ruby の様々なライブラリを集めたサイトが Ruby Application Archive (RAA) です。

http://raa.ruby-lang.org/

このページでは、キーワードや分類によって多数の Ruby ライブラリを検索できます。

RubyForge

RubyForge は多くの Ruby 関連プロジェクトの開発・配布サイトです。

http://www.rubyforge.net/

RubyForge では Ruby on Rails と共に広く使われるようになった Ruby Gems というパッケージ形式での配布も行われています。

書籍

Ruby関連の書籍も増えてきましたが、以下の本などが参考になると思います。

Rubyのインストール方法

「るびま」にて詳しく解説されています http://jp.rubyist.net/magazine/?FirstStepRuby

ReFeのインストール方法

```
% wget http://i.loveruby.net/archive/refe/refe-0.8.0-withdocsrc.tar.gz
% tar zxvf refe-0.8.0-withdocsrc.tar.gz
% cd refe-0.8.0
% ruby setup.rb config
% ruby setup.rb setup
% sudo ruby setup.rb install
```

Refeの使い方

```
% refe Array     # => Array クラスの説明とメソッドの一覧が表示される
% refe ar comp   # => Array の compact メソッドの説明が表示される
```

（クラス名やメソッド名は、先頭一致でユニークになる場合は省略できるようになっています）

UNIXの基本

必須30コマンド

By Toshiaki Katayama

Ruby は Windows や Mac でも実行できますが、バイオインフォマティクスの解析は Linux, Solaris, IRIX, Mac OS X, Cygwin などの UNIX 系の OS 上で行われることが多いため、基本的な UNIX のコマンドを知っておくことは有用です。

man (manual)

コマンドのオンラインマニュアルを参照するコマンド。

例：ls コマンドのマニュアルを表示。

```
% man ls
```

mkdir (make directory)

ディレクトリを作成するコマンド。

例：practice ディレクトリを作成。

```
% mkdir practice
```

オプション

```
-p 中間のディレクトリも自動的に生成
   すでにディレクトリが存在する場合にもエラーを出さない
```

例：practice/subdir/foo/bar ディレクトリを作成。subdir や foo なども自動的に作成される。また、すでにディレクトリがある状態で何度実行してもエラーにならない。

```
% mkdir -p practice/subdir/foo/bar
% mkdir -p practice/subdir/foo/bar
```

pwd (print working directory)

今いるディレクトリをフルパスで表示。

```
% pwd
```

cd (change directory)

今いるディレクトリを変更するコマンド。

例：practice ディレクトリに移動（相対パスで現在いるディレクトリ内の practice サブディレクトリへ移動）。

```
% cd practice
```

例：フルパスで /usr/local/src ディレクトリに移動（絶対パスで / から辿って /usr/local/src に移動）。

```
% cd /usr/local/src
```

例：１つ上のディレクトリに移動。

```
% cd ..
```

例：２つ上のディレクトリに移動。

```
% cd ../..
```

例：現在どこにいても、ホームディレクトリに移動。

```
% cd
```

ls (list)

ファイルの一覧を表示。

```
% ls
% ls /usr
```

オプション

```
-a 全ファイル (all)
   ファイル名がドット「.」で始まる隠しファイルも表示
-l 長いフォーマットで表示 (long)
-t タイムスタンプ順にソート (time stamp)
-r 逆順にソート (reverse)
```

例：ls コマンドのオプションと、出力結果の保存

```
% ls -al
% cd practice
% ls -l /usr/bin/ > file1
% ls -ltr /var/log/ > file2
```

ファイルの種類による表示

```
-F ファイル名に続く記号で種別を表示
   ディレクトリ(/), 実行可能(*), リンク(@)
-G 色付きで表示
```

```
% ls -FG /etc/
% ls -lFG /usr/bin/
% ls -ltrFG /usr/lib/ruby/1.8/
```

cat (concatenate)

ファイルの中身を表示。

```
% cat file1
```

複数のファイルを連結。

```
% cat file1 file2 > subdir/file3
```

head

ファイルの先頭 10 行だけを表示する。

```
% head file1
```

例：ファイルの先頭から３行だけを表示。

```
% head -3 file1
```

tail

ファイルの末尾 10 行だけを表示する。

```
% tail file2
```

例：ファイルの末尾から３行だけを表示。

```
% tail -3 file2
```

cp (copy)

ファイルをコピー 。

```
% cp file1 file3
```

例：複数ファイルを別のディレクトリにコピー。

```
% cp file1 file2 subdir/
```

例：別のディレクトリ (/etc) のファイル (services) を現在いるディレクトリ (.) にコピー。

```
% cp /etc/services .
```

例：ユーザ sample さんのホームディレクトリにある .zshrc ファイルを自分のホームディレクトリにコピー（「~ユーザ名」でホームディレクトリを表し、自分のユーザ名は省略できます）。

```
% cp ~sample/.zshrc ~/
```

オプション

```
-p パーミッションを保持する
-r サブディレクトリも含めてコピー
-i 上書きするかどうかを確認する
```

例：file1 と file2 をオプションなしと -pi オプション付きでコピーした場合の比較

file1 と file2 のパーミッション、更新時間を確認。

```
% ls -l file1 file2
% cp file1 file2 subdir
% ls -l subdir
```

この時点では file1, file2 の更新時刻は保存されていない

```
% cp -pi file1 file2 subdir
% ls -l subdir
```

２回目は、上書きするかどうか確認され、コピー元の file1, file2 と同じパーミッション、更新時間になっている。

例：ディレクトリ subdir の中身を再帰的に全部 subdir2 ディレクトリにコピー。

```
% cp -pr subdir subdir2
```

mv (move)

ファイルの移動（見方を変えるとファイル名の変更と同じ意味である）。

例：subdir 以下にあるファイル file3 を、現在いるディレクトリに file4 という名前で移動。

```
% mv subdir/file3 file4
```

rm (remove)

ファイルを削除。

```
% rm file4
```

オプション

```
-r サブディレクトリ以下を再起的に削除。
-f ファイルを消してよいかどうか尋ねない。
   消そうとするファイルがない場合にエラーを出さない。
```

例：ディレクトリ subdir2 と中のファイルを全て削除。

```
% rm -rf subdir2
```

chmod (change mode)

ファイルの所有者 (u) グループ (g) 他人 (o) ごとに、読み (r) 書き (w) 実行 (x) モード（「アクセス権」または「パーミッション」という）を変更。ディレクトリの場合、実行(x)がそのディレクトリ内に移動できる（見ることができる）という意味になる。

オプション

```
* 誰に [u,g,o] 読み/書き/実行を [r, w, x]
  許可/不許可する [+,-] の組み合わせ。
* または数字３桁で [0-7][0-7][0-7] の組み合わせ。
```

例：同じグループ (g) の人と他人 (o) にはファイルを読み書き実行できなくする。

```
% chmod go-rwx file1
```

例：同じグループ(g)の人に読み書き実行を許可する。

```
% chmod g+rwx file2
```

r を 4, w を 2, x を 1 として、組み合わせた rwx を 7, rw を 6, rx を 5 など、足した数字を用いる方が柔軟。4, 2, 1 の足し算のすべての組み合わせ (0〜7) で r, w, x の組み合わせ全 8 通りのパターンを表現できる。

つまり r の有無、w の有無、x の有無、による組み合わせは 2 の 3 乗で 8 通りとなり、これを u, g, o ごとに 3 bit ずつ 8 進数に置き換えることで、数字 3 桁あれば決定できることになる。

例：誰でもが実行でき、自分だけが書き換えることもできる rwxr-xr-x のモードに変更する。

```
% chmod 755 file1
```

例：誰でもが中身を読むことができ、自分だけが書き換えることもできる rw-r--r-- のモードに変更する。

```
% chmod 644 file2
```

例：自分だけが読み書きでき、他人には見ることのできない rw------- のモードに変更する。

```
% chmod 600 file3
```

モードが変わっていることを確認する。

```
% ls -l file1 file2 file3
```

一般的には新しいファイルやディレクトリを作成した場合に、以下のようなモードにしておくと使いやすい。

```
rwxr-xr-x (755) 実行ファイル, ディレクトリ
rw-r--r-- (644) 通常のファイル
```

このためには、umask コマンドを用いて

```
% umask 022
```

としておく（通常は、シェルの設定ファイルで実行するようにしておく）。内容を他人には見られたくないファイルは、

```
rwx------ (700) 実行ファイル, ディレクトリ
rw------- (600) 通常のファイル
```

のモードに変更すればよい。

wget (www get)

HTTP, FTP プロトコルでファイルを取得。

```
% wget http://bioruby.org/index.html
```

curl (command line URL)

wget と同様の機能を持つコマンドで -O オプションにより wget の代用ができる。

```
% curl -O http://bioruby.org/index.html
```

gzip (GNU zip)

ファイルを圧縮、展開するコマンド。ファイルサイズを小さくして保存し直すことができる。圧縮されたソフトウェアやデータベースなどを取得した際に展開するためによく利用する。

例：ファイルを圧縮する。ファイル名に拡張子 .gz が加わり、サイズが小さくなる。

```
% ls -l
% gzip index.html
% ls -l
```

例：圧縮されたファイルを展開する。gzip -d と gunzip コマンドは同等。

```
% gzip -d index.html.gz
% gunzip index.html.gz
```

例：圧縮されたファイルはそのままにし、展開した内容を標準出力に表示する。gzip -dc と gzcat コマンドは同等。ファイルやコマンドにリダイレクトする場合や、元のファイルをそのまま保存しておきたい場合に使用。

```
% gzip -dc index.html.gz | less
% gzcat index.html.gz > index.html
```

tar (tape archive)

複数のファイルを tar フォーマットの 1 つのファイルにまとめたり展開したりする。

オプション

```
c アーカイブを作成 (create)
t アーカイブの中に含まれるファイル名を表示 (toc)
x アーカイブの中に含まれるファイルを展開 (extract)
v 表示を冗長にする (verbose)
f アーカイブのファイル名を指定 (file)
```

歴史的経緯により、普通はオプションにハイフン - をつけない。また、tape archive の名前は、かつてファイルのバックアップをテープに保存していた時代の名残。

例：複数のファイルやディレクトリを一つのアーカイブ archive.tar ファイルにまとめる。

```
% tar cvf archive.tar file1 file2 subdir/
```

例：tar アーカイブを展開し中身のファイルを取り出す。

```
% tar xvf archive.tar
```

例：tar アーカイブの中身を確認するが展開はしない。

```
% tar tvf archive.tar
```

フリーソフトウェアなどでは、tar アーカイブを gzip で圧縮したものを配布している場合が多い。この場合の拡張子は .tar.gz や .tgz などになっている。

圧縮ファイルの展開

```
z アーカイブを圧縮または解凍 (gzip)
```

例：gzip で圧縮されている tar アーカイブを展開しながら中身を取り出す。GNU 以外の tar コマンドには z オプションがないため、gzip -dc と併用することもある。

```
% tar zxvf archive.tar.gz
% gzip -dc archive.tar.gz | tar xvf -
```

wc (word count)

ファイル内の行数、ワード数、文字数をカウント。

```
% wc file1
```

オプション

```
-l ファイルの行数のみをカウント。
```

例：ls コマンドの結果が何行あるかをカウント。

```
% ls | wc -l
```

more

テキストファイルの中身を 1 画面ごとに停止して、ページ毎に表示するコマンド。スペースキーで進む。

```
% more file1
```

less (less is more)

名前は less だが more より高機能なコマンド。このようなコマンドをページャーと呼ぶ。他のページャーでは、他言語に対応した lv コマンドや、HTML にも対応するコマンドラインの WWW ブラウザ w3m や lynx などもよく使われる（デフォルトでどのページャーを使うかは環境変数 PAGER に設定しておくことができる）。

```
% less file1
```

オプション

```
-S 長い行を折り返さずに表示。
```

例：横長いファイルを折り返さずに表示。画面に表示しきれない部分へは、カーソルキーで左右に移動できる。

```
% less -S file1
```

less の主なキー操作

```
スペース 次のページへ
b        前のページへ
j        1行下へ（カーソルキーの↓でもOK）
k        1行上へ（カーソルキーの↑でもOK）
g        ファイルの先頭へ
100g     ファイルの100行目へ
/        検索
n        検索の次のマッチへ進む
N        検索の前のマッチへ戻る
```

grep (g/RE/p)

テキストファイルから指定したパターン（RE: regular expression, 正規表現) にマッチする行を抜き出すコマンド。

```
% grep system file1
```

例：行の先頭 ^ が > で始まる行を抜き出す。

```
% grep '^>' fastafile
```

オプション

```
-i 大文字小文字を区別せずにマッチ
-v 指定した正規表現を含まない行にマッチ
-c マッチした行数を表示
```

例：大文字小文字を区別しないため kinase にも Kinase にも kInAsE にもマッチ。

```
% grep -i kinase fastafile
```

例：kinase という文字が含まれない行を表示

```
% grep -v kinase fastafile
```

GNU grep 2.5 以降で有効なオプション

```
--color
--perl-regexp (-P)
--only-matching
```

これらのオプションは環境変数 GREP_OPTIONS に指定しておくことができる。

diff (difference)

２つのファイルの異なる行（差分）を表示する。

```
% diff file1 file2
```

オプション

```
-i 大文字小文字を区別せずに比較
-w 空白文字 (white space) の違いを無視
-B 空行の違いを無視
-c context diff 形式で出力
-u unified diff 形式で出力
-y 左右に並べた side-by-side 形式で出力
-r ディレクトリどうしを recursive に比較
-N 存在しないファイルを空ファイルとして扱う
```

patch

diff の結果をもとに差分を適用しファイルを更新する。フリーソフトウェアのバグ修正等でよく用いられた。

```
% patch -p 1 -s < update.diff
```

find

指定した名前や日付などを持つファイルを検索する。

オプション

```
-name ファイル名のパターンを指定
-type ファイルの種別を指定
        f 通常のファイル
        d ディレクトリ
        l シンボリックリンク
-size ファイルのサイズを指定
```

例：カレントディレクトリ . 以下から、file で始まるファイル名のファイルを検索。

```
% find . -name 'file*'
```

例：/etc ディレクトリから、ディレクトリ等ではなく通常のファイルだけを検索。

```
% find /etc/ -type f
```

例：/var ディレクトリからファイルサイズが 0 のファイルだけを検索。

```
% find /var/ -size 0
```

ちなみに、zsh では **/file*(@) などの表記で、find を使わなくても再帰的なファイル名検索ができる。

```
% ls **/file*          # dir/**/ で dir 以下全て
% ls /etc/**/*(.)      # *(.) で通常のファイル
% ls /etc/**/*(@)      # *(@) でシンボリックリンク
% ls -d /etc/**/*(/)   # *(/) でディレクトリ
% ls -l *(L0)          # *(L数字) でファイルサイズ
```

ln (link)

リンクファイルを作成する（実体が同じファイルに別の名前をつける）。ハードリンクの場合は、リンク元のファイルが消えても実体は残るが、シンボリックリンクの場合はアクセスできなくなる。

例：file3 を link1 にハードリンク。

```
% ln file3 link1
```

例：file3 を symlink1 にシンボリックリンク。

```
% ln -s file3 symlink1
```

例：ハードリンクのリンク数は ls の -l オプションで、i-node 番号は -i オプションでそれぞれ表示される。リンク元の file3 を消して、ハードリンクとシンボリックリンクの違いを確認する。

```
% ls -li
% rm fil3
% less link1
% less symlink1
```

touch

ファイルの更新日時を変更する。ファイルがなければ作成、あれば更新時間を現在時刻に設定する。

```
% touch newfile
```

オプション

```
-r 更新日時を参照するファイルを指定
```

例：newfile の更新時刻を file1 と同じにする。

```
% touch -r file1 newfile
```

echo

指定した文字列を標準出力に表示。

```
% echo "hello"
% echo "done" >> finished.txt
```

オプション

```
-n 改行を出力しない
```

ps (process)

現在実行中のプロセスの一覧を表示。

```
% ps
```

他人の動かしているものも含めてすべてのプロセスを表示。詳しい情報を表示するためのオプションは、aux 系と -elf 系があり、OS によって異なる。

```
% ps aux    (BSD系) # auxww など w の数で横幅が増加
% ps -elf   (SYSV系)
```

kill

指定したプロセス番号にシグナルを送る。通常はそのプロセスを強制終了したい場合に使用。

例：プロセス ID (ps コマンドで確認) 12345 のプロセスを止める。

```
% kill 12345
```

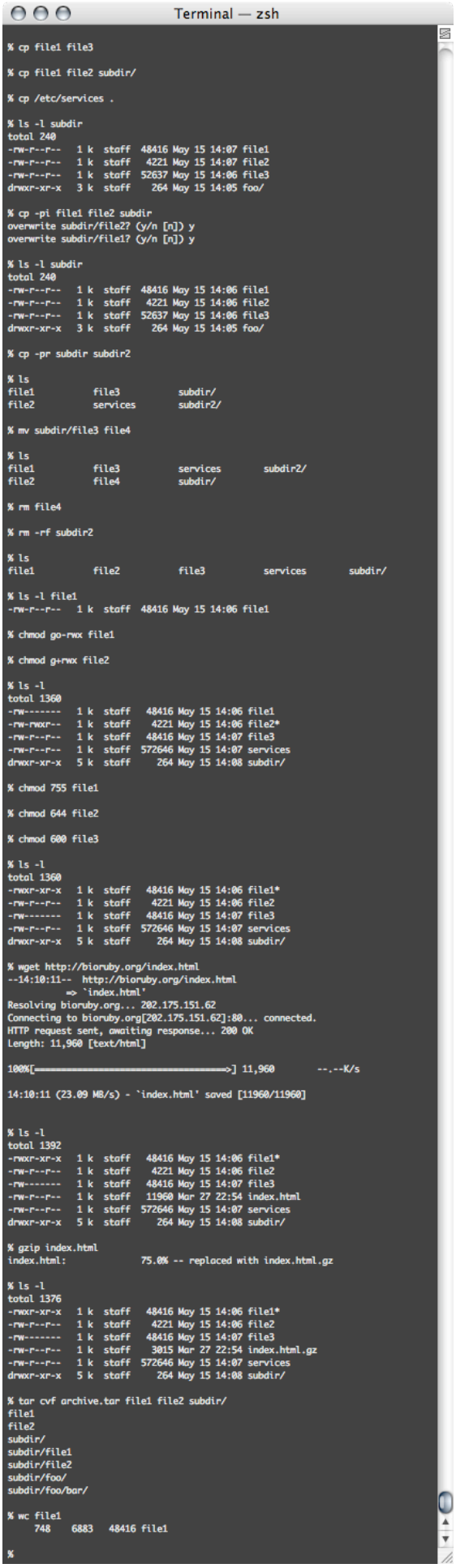
kill だけでは止まらない暴走プロセスの場合も、強制終了に相当する -9 (-KILL) シグナルを送ると止められる場合が多い。

```
% kill -9 12345
```

主なシグナルの種類

```
-1  HUP (hang up)
-2  INT (interrupt)
-3  QUIT (quit)
-6  ABRT (abort)
-9  KILL (non-ignorable kill)
-14 ALRM (alarm clock)
-15 TERM (software termination signal)
```

これらのシグナルを受けた時のプロセスの挙動はプログラムに依存する。ネームサーバの named など、HUP シグナルを受けることで設定ファイルの再読み込みを行うソフトウェアもある。



シェルの基本

効率的なコマンド操作

By Toshiaki Katayama

シェルとは

シェルは UNIX のコマンドを実行するためにターミナルの中で利用するインターフェイスです。Bシェル系の sh, bash, zsh や Cシェル系の csh, tcsh などがありますが、それぞれ文法と機能が若干異なります。機能が豊富で作業効率の高い zsh がお勧めです。

> (redirect)

コマンドの出力結果をファイルにリダイレクト。既存の内容は上書きされる。

```
% date
% date > file1
% cat file1  (←表示して確認)
```

>> (append)

コマンドの出力結果をファイルに追記。

```
% date >> file1
% cat file1  (←表示して確認)
```

| (pipe)

コマンドの出力結果を別のコマンドに入力。

```
% ls -l | less
% grep http /etc/services | less
```

Ctrl-C (stop)

コマンドの実行を中断。

```
% blastall -p blastp -i q.seq -d t.seq
Ctrl-C
```

ずっと終了しないコマンドを停止させる例。

```
% ruby -e 'while true; puts "hoge"; end'
hoge
hoge
 :
Ctrl-C
```

Ctrl-Z (suspend)

コマンドの実行を一時停止。

```
% blastall -p blastp -i q.seq -d t.seq
Ctrl-Z
% bg  (←計算をバックグラウンドで再開)
% jobs  (←表示してジョブを確認)
```

ファイルの編集中に一時的にコマンドラインに戻る例。

```
% vi hoge.txt
Ctrl-Z
```

jobs

実行中のジョブ（コマンド）の一覧。

```
% jobs
```

プロセス ID なども含めた長いフォーマットでの表示。

```
% jobs -l
```

bg (background)

一時停止中のジョブをバックグラウンドに。

```
% bg
% jobs  (←表示して確認)
```

fg (foreground)

一時停止中またはバックグラウンドのジョブをフォアグラウンドに。

```
% fg
```

停止中のジョブが複数ある場合に、２つ目のジョブをフォアグラウンドにする例。

```
% fg %2
```

& (background)

最初からジョブをバックグラウンドで実行。jobs コマンドでバックグラウンドジョブを表示して確認。

```
% blastall -p blastp -i q.seq -d t.seq &
% jobs  (←表示して確認)
```

export

環境変数の設定（正確には、子プロセスにも引き継がれるように設定）。

コマンドを検索するディレクトリのリスト PATH に /usr/local/bin を追加。

```
% echo $PATH  (←表示して確認)
% export PATH="$PATH:/usr/local/bin"
% echo $PATH  (←表示して確認)
```

優先的に使用するページャーを less に設定。

```
% export PAGER=less
% echo $PAGER  (←表示して確認)
```

for *var* in *files*

複数ファイル（項目）を順に処理する。

大腸菌の遺伝子 b0001〜b0004 までの配列を順に取得（項目を順に処理する例）。

```
% for gene in b0001 b0002 b0003 b0004
do
  echo $gene
  bget.rb -f -n a eco $gene > ${gene}.fa
done
```

取得した各配列ファイルを EMBOSS の pepstats コマンドで処理（ファイルを順に処理する例）。

```
% for file in b000*.fa
do
  pepstats $file -auto
done
```

pepstats の出力結果ファイル名を変更してみる（basename コマンドの使用例）。

```
% for file in b000*.pepstats
do
  gene=`basename $file .pepstats`
  mv $file ${gene}.out
done
```

ZshのTips

zshには、他のシェルにはない便利な機能が数多くあります。

- 状況に応じた補完候補のメニュー選択機能
- Ctrl-r, Ctrl-s によるヒストリ検索機能
- マルチラインエディタ zle を内蔵
- 環境変数エディタ vared を内蔵
- 再帰的で分かりやすいファイルマッチ **/*(@)
- kill-region など多彩なコマンドライン編集機能
- コマンドのフルパス展開機能 =blastall

などなど。以下では zsh のちょっとした小技を紹介します。

ファイルの属性による選択

「UNIXの基本」の find コマンドの項でも挙げましたが、ワイルドカードを () で修飾して

```
% ls /etc/**/*          # /etc 以下の全ファイル
% ls /etc/**/*(.)       # /etc 以下の通常ファイル
% ls /etc/**/*(@)       # /etc 以下のリンク
% ls -d /etc/**/*(/)    # /etc 以下のディレクトリ
```

のように、特定の属性を持つファイルだけを選択することができます。さらに

```
% ls -l *(L0)       # サイズが 0 のファイルのみ
% ls -l *(f700)     # rwx------ のファイルのみ
% ls -l *(u[hoge]) # ユーザ hoge のファイルのみ
% ls -l *(g[db])    # グループ db のファイルのみ
```

といったファイルのモードでの選択や、

```
% ls -l *(mw-3) # 3週間以内に更新されたファイル
% ls -l *(m0)   # 今日更新されたファイル
% ls -l *(mM+6) # 6ヶ月以上古いファイル
% ls -l *(ch-2) # 2時間以内に移動等したファイル
```

のようにタイムスタンプによる選択が可能です。

リンクファイルそのものではなく、リンク先のファイルを指す場合には先頭に - をつけます。ディレクトリとディレクトリへのリンクのみの場合 (-/) など。

```
% ls -d *(-/)
% ls -d *(-m0)
```

m番目からn番目までのファイル

さらに、パターンにマッチするファイルのうち何個目のもの、という指定も可能です。DSCで始まるファイルのうち２番目から５番目のファイルにマッチする場合：

```
% rm DSC*([2,5])
```

ちなみに、以下のようにするとファイル名の末尾が 2 か 5 で終わるファイルになります。

```
% rm DSC*[2,5]
```

数字リスト展開

数字の並びを {m..n} で生成することができますので、連番のファイルやその一部を処理する際に便利です。

```
% mv DSC0{3769..3822}.jpg photo/
% for i in {1..10}
do
  cp foo$i.txt bar$i.txt
end
```

「このパターン以外」のファイル

ほとんど全部のファイルにマッチしたいけど、このパターンのファイル名だけは除外という場合は ~ を使うことができます。

```
% ls *~CVS       # CVS 以外の全てのファイル
% rm *.o~hoge.o  # hoge.o 以外の全ての.oファイル
% ls *~*[0-9]*   # 数字を含むファイル以外
```

ファイル名からパス名を取り去る

ファイル名に対する basename や dirname コマンドと同様の操作を zsh の機能だけで実現できます。

```
# ruby コマンドのパス (=ruby) を変数に格納
% file==ruby
# basename コマンド相当
% echo $file            # /usr/local/bin/ruby
# basename コマンド相当
% echo $file:t          # ruby
# dirname コマンド相当
% echo $file:h          # /usr/local/bin
```

標準エラー出力の取り扱い

sh, bash など (Bシェル系) では、標準出力 (1) と標準エラー出力 (2) を使い分けることができます。混ぜるには 2>&1 を使います。

```
% command 1> out.log 2> err.log  # 別のfileに
% command 2>&1 | less            # 混ぜてpipeに
% command > outerr.log 2>&1      # 混ぜてfileに
```

タイムスタンプ

1．時刻の種類を指定
ファイルのタイムスタンプは３種類あります。

- **m** ファイルの更新時刻
- **a** 最終アクセス時刻
- **c** i-node の変更時刻

2．期間の指定
指定なしの場合の数字は日数を表します。

- **M** 月 (month)
- **w** 週 (week)
- **h** 時 (hour)
- **m** 分 (minute)
- **s** 秒 (second)

3．時間軸の指定
指定なしの場合ちょうどその時間（n日前、m分前など）を表します。

- **+** 指定した時刻以前
- **-** 指定した時刻以後

4．数値の指定
時間を表す数字を指定します。

csh, tcsh など (Cシェル系) では、標準エラー出力だけを使い分ける機能がないためサブシェルの構文を使います。一方、混ぜる場合は >& や |& と簡潔に書くことができます。

```
% (command > out.log) >& err.log # 別のfileに
% command |& less               # 混ぜてpipeに
% command >& outerr.log         # 混ぜてfileに
```

zsh では、Bシェル系の 1>, 2> を使った使い分け構文と |&, >& の混ぜ構文のどちらも利用できます。

複数コマンドとの入出力

paste, cut と zsh の <(コマンド) 構文を使うと「file1 の１カラム目と file2 の３カラム目を連結」といった操作も中間ファイルを使わずに実行可能です。

```
% paste <(cut -f1 file1) <(cut -f3 file2)
```

さらに、>(コマンド) 構文を使って同じ結果を複数のプロセスに渡すこともできます。このため | と tee の組み合わせよりも柔軟に出力を加工することが可能です（この機能を利用するには setopt multios の設定が必要）。

```
% paste <(cut -f1 file1) <(cut -f3 file2)
  > >(ruby -e 'p ARGF.read')
  > >(perl -e 's/^/perl:/')
% date > >(tac) >(cat)
```

サブプロセスの完了を待つ

wait 文を使うとシェルスクリプトなどで並列に実行したサブプロセスの終了を待つことができます。

```
# FASTA フォーマットの配列を準備
  :
# BLAST を実行
for s in *.seq
do
  blastall -p blastp -i $s -d db > $s.out &
done
wait
  :
# 結果を解析
  :
```

Zshの環境設定

zshの設定ファイルはいくつかありますが、基本的には ~/.zshrc ファイルに必要な設定を書いておけばよいでしょう。

~/.zshrc

メインの設定ファイルで、以下のような項目を記述します（rc は UNIX の起動時に実行されるファイル /etc/*rc 系の名残で run command の略）。

- limit, umask, stty などの設定
- PATH, RUBYLIB など環境変数の設定
- PROMPT, SAVEHIST などシェル変数の設定
- alias (エイリアス) の設定
- function (関数) の設定
- autoload (追加機能) の設定
- bindkey (キーバインド) の設定
- zstyle (補完などのスタイル) の設定
- zshoptions (オプション) の設定

また、必要に応じてシェル変数 OSTYPE により OS 毎の場合分けをします。次ページに設定のサンプルを掲載しますので、適宜カスタマイズして使ってください。

export は変数を環境変数として宣言します。alias はコマンドに別名を設定します。複雑なコマンドは **function** で関数として設定します。なかでも **precmd** は、プロンプトが表示される前に毎回必ず実行される特別な関数です。例として負荷に応じて顔文字を変更する関数を設定しています。

その他 zsh 特有の設定は後半にまとめてありますが、**zstyle** と **setopt** は zsh の挙動を大きく変更する設定で、さまざまな選択肢がありますので

```
% man zshoptions
% man zshall
```

などを参照して、使いやすい設定に変更してください。

~/.zlogout

シェルを終了する際に１度だけ実行されるファイル。たとえば clear と書いておけばログアウト時に画面がクリアされます。

~/.zhistory (自動生成)

使用したコマンドのヒストリを記録しておくファイル。シェル変数 SAVEHIST に指定した行数のコマンドが記録されます。次回 zsh を起動した際にも、矢印↑キーや Ctrl+R キーによる検索で以前の履歴を遡ることができるため、長いコマンドラインを入力し直す手間が省けます。

Emacs や vi エディタのキーバインドに慣れている人は、bindkey -e や -v でどちらかのキーバインドを設定しておくことで、zsh のマルチラインエディタと高度なコマンドライン編集機能を活用することができます。

サンプルの ~/.zshrc

ホームディレクトリに置いてカスタマイズ

```
#
# Sample ~/.zshrc (C) 2006 Toshiaki Katayama <k@bioruby.org>
#


### shell

umask 022

unlimit
limit -s
#limit core 0

stty intr '^C'
stty susp '^Z'


### environmental variables

#export EDITOR=vi
#export PAGER=less
#export LESS='-iqSMM'

export BIOROOT="/bio"
PATH="/usr/local/bin:/sw/bin:/opt/sfw/bin:/usr/bin:/bin"
PATH="$PATH:/usr/ucb:/usr/ccs/bin"
PATH="$PATH:/usr/sbin:/sbin"
PATH="$PATH:/usr/openwin/bin"
PATH="$PATH:$BIOROOT/bin"
export PATH
#export LD_LIBRARY_PATH
#export MANPATH

export CVS_RSH=ssh
export RSYNC_RSH=ssh
#export RUBYLIB="$HOME/lib/ruby"
#export PERLLIB="$HOME/lib/perl"
#export SQL_TRACE=1


### aliases

alias ls='ls -F'
#alias ls='ls -F --color=auto'
alias la='ls -a'
alias ll='ls -l'
alias lla='ls -la'
alias lld='ls -lad'
alias llt='ls -lat'
alias lls='ls -laS'
alias dir='ls -l'

#alias df='df -h'
#alias du='du -h'

#alias pu=pushd
#alias po=popd

alias res='export TERM=xterm; resize > /dev/null'
alias irb='irb -r irb/completion --simple-prompt'


### functions

function setenv() { export $1=$2 }

function psgrep () {
  case $OSTYPE {
    solaris*)
      ps -ef | head -1
      ps -ef | grep -i $* ;;
    linux*)
      ps auxw | head -1
      ps auxw | grep -i $* ;;
    darwin*)
      ps auxw | head -1
      ps auxw | grep -i $* ;;
    *)
      ps auxw | head -1
      ps auxw | grep -i $* ;;
  }
}

function precmd () {
  load=`uptime | sed -e 's/.*: \([^, ]*\).*/\1/'` 2> /dev/null
  case $load {
    0.0*)     face="-----"   ;;
    0.[123]*) face="(^-^)"   ;;
    0.[456]*) face="(^^;)"   ;;
    0.[789]*) face="(^^;;)"  ;;
    1*)       face="(^^;;;)" ;;
    2*)       face="(;_;)"   ;;
    [3-9]*)   face="(T_T)"   ;;
    *)        return         ;;
  }
  echo "$face load:$load"
}


### zsh specific

## man zshparam

PROMPT="%n@%m:%~%# "

HISTSIZE=100000
SAVEHIST=100000

## man zshcompsys

autoload -U compinit
compinit -u

## man zshmodules (zsh/zle)

bindkey -e
bindkey '^U' backward-kill-line
bindkey '^W' kill-region
bindkey '^[h' vi-backward-kill-word
bindkey '^[.' copy-prev-word
bindkey '^O' vi-open-line-below

## man zshmodules (zsh/zutil)

zstyle ':completion:*' menu select
zstyle ':completion:*' list-colors 1

## man zshopts

setopt correct
setopt equals
setopt list_packed
#setopt multios
#setopt share_history
```

Rubyの実行のしかた

#!/usr/bin/env ruby とワンライナー

By Toshiaki Katayama

Ruby スクリプト

Rubyのプログラムは、通常ファイルに保存して実行します。

```
#!/usr/bin/env ruby

puts "Hello, World!"
```

エディタで上記の内容の test.rb ファイルを作成し、実行してみましょう。

```
% ruby test.rb
Hello, World!
```

または、

```
% chmod 755 test.rb
% ./test.rb
Hello, World!
```

この場合、1行目の #!/usr/bin/env ruby により、環境変数 PATH の通っているディレクトリから ruby コマンドを探し出してプログラムを実行してくれます。

ruby コマンドがインストールされている場所は OS や環境によって違いますが、この記法を使うとスクリプトを修正する必要がなくなります。一方で、複数の ruby コマンドがインストールされている場合、どれが使われるかは環境変数 PATH の順番に依存して決まります。

インタラクティブな実行

短いプログラムの場合、いちいちファイルに書いて実行するのは面倒なため、コマンドラインでプログラムを記述して実行する「ワンライナー」を使うことがあります。

```
% ruby -e 'puts 2**8'
256
```

オマケとして、Perl と Ruby で巨大な数字の扱いの違いを見てみましょう。

```
% perl -e 'print 2**49'
562949953421312
% ruby -e 'print 2**49'
562949953421312
% perl -e 'print 2**50'
1.12589990684262e+15
% ruby -e 'print 2**50'
1125899906842624
% perl -e '2**2**2**2**2'
inf
% ruby -e '2**2**2**2**2'
200352993040684646497...
(19729桁つづく)
```

また、irb (interactive ruby) を使うと、実行結果を確認しながらインタラクティブに実行していくことができます。

```
% irb
irb(main):001:0> a = 2 + 3
=> 5
irb(main):002:0> a + a
=> 10
irb(main):003:0> a * a
=> 25
irb(main):004:0>
```

メソッドや変数名などを Tab キーで補完したい場合は、irb コマンドに completion ライブラリを読み込ませます。

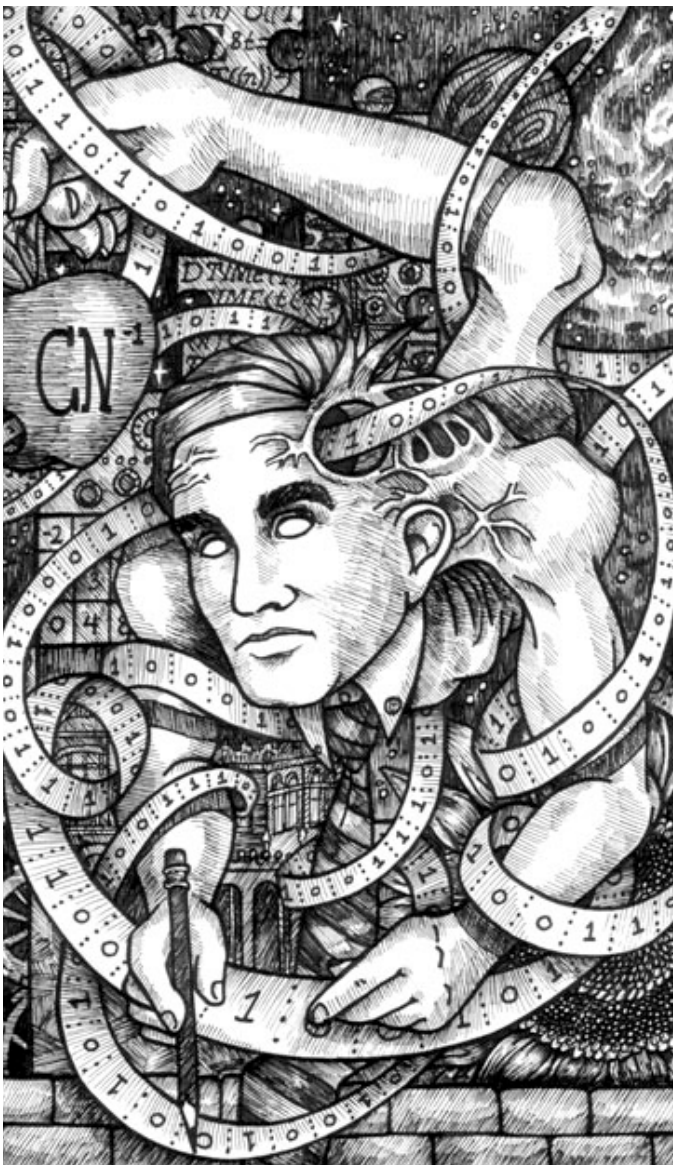
```
% irb -r irb/completion
```

補完機能は readline ライブラリに依存しているため、環境によっては使えないこともあります。irbを起動したあとに補完が必要になった場合はirb上でrequireすることも可能です。

```
% irb --simple-prompt
>> require "irb/completion"
```

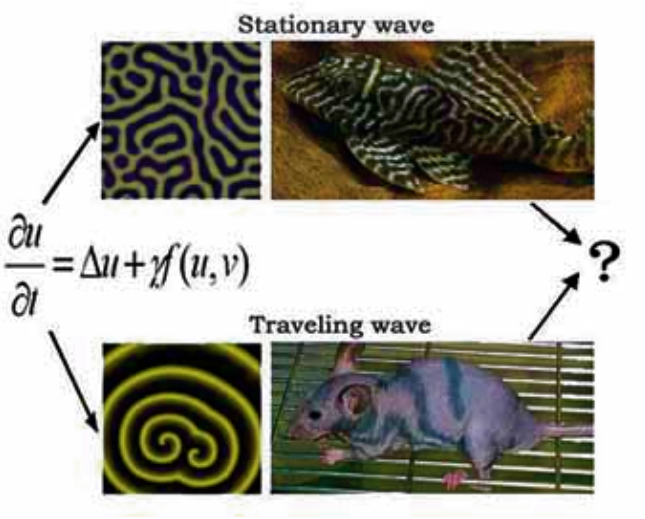
よく使う場合はシェルの設定ファイルで alias を設定しておいてもよいでしょう。

```
alias irb='irb -r irb/completion'
```



「ヒトの代わりに考える機械を作ることはできない」

チューリング・マシンの考案者
Alan M. Turing (1912 - 1954)



第２回 データリソース活用 I

フラットファイルと正規表現

By Masumi Itoh, Toshiaki Katayama

コンテンツ

1. フラットファイルデータベース
   1.1. フラットファイルとは
   1.2. さまざまなフォーマット
      GenBank, KEGG, UniProt, PROSITE, Pfam, Rfam, Ensembl, PDB
   1.3. フラットファイルの利用法
2. Rubyのおやくそく
   2.1. ARGVとARGF
   2.2. 正規表現

第２回目は、Rubyプログラムで最初に扱う生命科学データベースのフラットファイルについて学ぶ。代表的なものとして、塩基配列データベースのGenBank、アミノ酸配列データベースのUniProt、タンパク質立体構造データベースのPDB、遺伝子や化合物とパスウェイなどを統合したデータベースKEGGなどのエントリを概観する。つづいて、これらのデータベースを利用する際に基本となるファイル入出力や、コマンドライン引数、正規表現について実習する。

フラットファイルデータベース

GenBank, KEGG, PDB ...

By Masumi Itoh

フラットファイルとは？

生物学のデータベースの多くは、ウェブインターフェイスで公開されているものと併せてフラットファイル形式としても公開されています。フラットファイル形式のデータベースは、単一もしくは複数のエントリをシーケンシャルに並べたテキストファイルやその集合です。もともとデータ量が少ない時期には、フラットファイル自体を管理する自前のプログラムを作ってデータベース作成が行われていましたが、その後多くのデータベースではリレーショナルデータベース管理システムを導入してデータベース作成が行われるようになり、そこからフラットファイルを生成してデータベース配布が行われるようになっています。多くのDNAやタンパク質のデータベースではデータに版権はなく、通常、学術目的には自由に再利用できるため、データベース全体を入手して自分のシステムに組み込んだり、解析することができます。フラットファイルは、特定の環境に依存せず、誰にでも容易に利用可能であるというメリットがあり、データベースに限らず、データの共有や結果の出力などに広く利用されています。

テキストファイルの扱い方

【テキストファイル】
文字データだけで構成されたファイル。どんな機種のコンピューターでも共通して利用できる数少ないファイル形式の一つ。 (IT用語辞典 e-Words http://e-words.jp/ より)

テキストファイルはどんな機種のコンピュータでも共通して利用できるのが利点ですが、実はWindowsやMacで作成する場合は注意が必要で、油断するとテキストファイルじゃなくなっていることがあります。なお、前回のプログラミング入門で説明があったとおり、プログラムもテキストファイルで記述します。

テキストファイルの閲覧は様々なプログラムを用いて出来ますが、今回はプログラミング実習らしく 、ターミナルからless コマンドを用いることにします。

まず、サンプルファイルをブラウザを使って

http://web.kuicr.kyoto-u.ac.jp/egis/prog/ からダウンロードしてきます。ダウンロードしたファイルはデスクトップに現れるので、作業ディレクトリ（フォルダ）をデスクトップに移動しましょう。

cdは作業ディレクトリの移動、lsはディレクトリ内のファイルのリストを表示するコマンドです（「UNIXの基本」参照）。

```
% cd ~/Desktop/samples/
% ls
```

ファイル名ががずらずらと並んで見えるので好きなものを選んで（例えばgenes）、

```
% less genes
```

と入力するとテキストファイルを閲覧することが出来ます。Emacs、テキストエディット、Windowsのメモ帳等のエディタでもテキストファイルを開くことが出来ますが、間違って編集してしまう可能性があることや、大きいファイルは扱いづらい、大げさ、という理由でlessなどのページャーがよく用いられます。それでは各データベースのファイルをみてゆきましょう。

GenBank

GenBank は米国 NCBI により公開されているDNA配列データベースで、世界最大のDNAデータレポジトリです。GenBankは元々、生物種グループに大まかに対応するディビジョン(division)に分類されており、現在ではESTなど特殊なカテゴリを含めた18のディビジョンに分けられています。欧州EMBL、国立遺伝学研究所DDBJと連携・データ共有し、国際塩基配列データベース(INSD)を構築しています。データはGenBankフォーマットと呼ばれる形式で配布され、この形式はRefSeqなどNCBIで公開される他のデータベースでも利用されています。

エントリ

DDBJ も形式は同じ。Feature table は DDBJ, EMBL, GenBank で共通の仕様を用いています。

- **LOCUS**
  エントリ名、配列長、タイプ、分子形態 (環状/線状) 、ディビジョン、データの最終更新日
- **DEFINITION**
  遺伝子・タンパク質名
- **ACCESSION**
  アクセッション番号
- **VERSION**
  アクセッション番号 + バージョン
- **SOURCE**
  由来する生物種・オルガネラ名
- **ORGANISM**
  由来する生物種の系統分類
- **REFERENCE**
  文献（登録者）情報。著者、ジャーナル、タイトルなど。最初の一つは登録者の情報。
- **FEATURES**
  配列の特徴を記述する。feature key (source, CDS, tRNA, rRNA等)、location、qualifier からなる。locationのフォーマットも結構複雑。
- **ORIGIN**
  この次の行から塩基配列が始まる。配列は60塩基ごとに改行され、行の先頭の数字は行頭の塩基が何塩基目かを示す。
- **//**
  エントリの区切り

KEGG

KEGG (Kyoto Encyclopedia of Genes and Genomes) は、京都大学バイオインフォマティクスセンターで開発・公開されている生命情報統合データベースです。KEGGは主に、代謝系・制御系の化合物、タンパク質のネットワーク情報を蓄積するPATHWAY、遺伝子やゲノム情報を蓄積するGENES、化合物情報を蓄積するLIGAND、生命知識の階層分類データベースBRITEからなります。今回は例としてGENESのエントリを示しますが、LIGAND など、他の KEGG データベースのエントリも類似の形式で記載されています。

GENESエントリ

- **ENTRY**フィールド
  エントリ名、エントリーの種類、生物種名。エントリの単位は遺伝子産物 (GENES、DGENES)、 コンティグ (EGENES) など
- **NAME**フィールド
  遺伝子・タンパク質名、別名
- **DEFINITION**フィールド
  機能アノテーション
- **ORTHOLOG**フィールド
  KO (KEGG Orthology) アノテーション（KEGG によるオーソログ遺伝子グループの機能分類）
- **PATHWAY**フィールド
  機能するパスウェイ (KEGG PATHWAYへのリンク)
- **POSITION**フィールド
  通常 KEGG GENOME の対応するエントリのゲノム上での位置。KEGG GENOME にエントリがない場合（ゲノム配列が未決定）、染色体番号やバンドなどのみがのっている場合もある。
- **MOTIF**フィールド
  エントリが持つモチーフ（Pfam、PROSITE、TIGERFAM など）
- **DBLINKS**フィールド
  ほかのデータベースへのリンク
- **CODON_USAGE** フィールド
  コドン使用頻度
- **AASEQ**フィールド (CDSのみ)
  アミノ酸配列長、アミノ酸配列
- **NTSEQ**フィールド
  塩基配列長、塩基配列
- **///**
  エントリの区切り

UniProt/Swiss-Prot

UniProt は欧州 EBI、スイス SIB (Institute of Bioinformatics)、米国 Georgetown University がそれぞれ開発・公開していたTrEMBL、Swiss-Prot、PIR を統合した、タンパク質配列データベースです。登録されているタンパク質配列は Swiss-Prot に由来する UniProt/Swiss-Prot と TrEMBL に由来する UniProt/TrEMBL に大きく分けれていて、このうちUniProt/Swiss-Protは以前のSwiss-Prot同様、キュレーターによる高い精度の機能アノテーションが付加されており、バイオインフォマティクス分野ではしばしば機能アノテーションの正解セットとして使用されます。一方のUniProt/TrEMBLはEMBL Nucleotide Sequence Databaseの配列を翻訳したものです。

エントリ

TrEMBLも同じ形式。また、基本的にEMBLやPROSITEなど欧州のデータベースは類似した形式なので、以下では同じ点は省略します。

- **ID (IDentification)**
  エントリ名、エントリのタイプ、配列長など
- **AC (ACcession number)**
  エントリのアクセッション番号。ID と紛らわしいが、リリースが変わっても同一のタンパク質を示す番号はこちら。ID は変更されないとは限らない。
- **DT (DaTe)**
  エントリの作成、更新日時
- **DE (DEscription)**
  エントリの説明。
- **GN (Gene Names)**
  遺伝子名
- **OS, OG, OC, OX**
  由来する種、オルガネラ、分類情報
- **RN, RP, RC, RX, RG, RA, RT, RL**
  文献情報
- **CC (Comment)**
  コメントそのほか。
- **DR (Database cross-References)**
  他のデータベースへのリンク。
- **KW (Key Word)**
  キーワード
- **FT (Feture Table data)**
  配列の特徴
- **SQ (SeQuence header)**
  この次の行から配列が始まる。
- **//**
  エントリの区切り

PROSITE

Swiss-Prot 同様、スイス SIB によって開発・運営されているタンパク質のファミリー、ドメインのデータベースで、タンパク質配列中にみられる各ファミリー・ドメインの特徴配列の PATTERN と MATRIX (PROFILE) を蓄積しています。また、特徴を自然言語 (英語) で記したRULE も登録されています。PATTERN はタンパク質の活性中心や、修飾部位などを短い正規表現（Rubyのものとは異なる）として表現したもので、MATRIX はファミリー、ドメインなどの配列のマルチプルアライメントから作成した Position Specific Score Matrix (PSSM) として表現されています。また、特異性の高くない PATTERN や MATRIX, RULE も登録されていますが、それらはスキップフラグが真 (true) になっています。また、Swiss-Prot中の擬陽性のタンパク質のリストも記載されています。

エントリ

- **ID (IDentification)**
  エントリ名。タイプとしてPATTERN, MATRIX (profile)、RULEのいずれかを示す。
- **PA (PAtterm)**
  アミノ酸パターン。正規表現で示される。
- **MA (MAtrix)**
  ドメインの配列プロファイル。
- **RU (RULE)**
  判別のためのルール。自然言語で記載される。
- **NR (Numerical Results)**
  PATTERN/MATRIX/RULE に該当する Swiss-Prot のエントリの統計情報。陽性・擬陽性等の情報も記載されている。
- **CC (Comments)**
  ファミリーを持つ種の分布 (super-kingdomごと) や、最大リピート数、スキップフラグなどの情報。
- **DR (Database References)**
  PATTERN/MATRIX/RULE に該当する Swiss-Prot のエントリ名。エントリ名の後の T/P/N/F/? は True positive、Potential hit、 False negative、False positive、 Unknown を示している。
- **3D (3D-structure)**
  PDB に登録されている三次元構造のエントリ名。
- **DO (DOcumentation)**
  PROSITE DOC のエントリ番号。

Pfam

英国 Sanger Institute によって開発・運営されているタンパク質のファミリー・ドメインのデータベース。各ファミリーのマルチプルアライメントとそのHMM (隠れマルコフモデル) を蓄積しています。バイオインフォマティクス研究において、もっとも利用頻度も高いドメインデータベースの一つです（隠れマルコフモデルについては、人材養成講義「バイオスタティスティクス特論」で学びます）。登録されている HMM は HMMER パッケージで作成されており、ローカルの環境で自前のデーターベースに対する検索などに使用することが出来ます。Pfam に登録されているファミリーは、専門家によるキュレーションされたファミリーのマルチプルアライメントからなる Pfam-A と、ProDom によりタンパク質をクラスタリングすることで自動生成されたファミリーからなる Pfam-Bに分けられます。また、進化的に類縁のファミリーもサブファミリーごとに別のエントリに登録されていることがあり、それらのファミリーの類縁関係は立体構造に基づき定義されておりPfam-Clan (Pfam-C) に記載されています。

エントリ

マルチプルアライメントを記述するために作成されたStockholm形式で記述されています。Stockholm形式ではマルチプルアライメントに対しての注釈が#で始まる行に記されています。文頭の#=GFがエントリ全体、#=GCがカラムごと、#=GSが配列ごと、#=GRが特定の配列に対するカラムごとの注釈をしめしています。各アノテーションの先頭（つまり#=Gxの次）はSwiss-Protの項で説明したものと類似しています。特徴的なものを以下に示します。また、次に紹介するRfamも基本的に同じ構成です。

- **AL (ALignment method of seed)**
  マルチプルアライメント作成に使用された手法。ClustalW, ProDom, T-Coffee, MAFFTなど。
- **BM (HMM Building command)**
  登録されている HMM を作成する際の hmmbuild コマンドのオプション設定。
- **AM (build method)**
  globalfirst/byscore/localfirst
- **SE (source of SEed)**
  ファミリーの基となる配列の取得もと。PROSITE, ProDom など。
- **GA, NC, TC**
  HMMER でドメインを検出する際のスコアの閾値やカットオフ。
- **TP (TyPe)**
  エントリの種別。Family/Domain/Repeat/Motif がある。
- **DC, DR**
  他のデータベースに登録される同じファミリーのエントリ。SCOP のフォールド、PDB の三次元構造など。

Rfam

Pfamを開発する英国 Sanger Institute と米国ワシントン大学で開発されているnon-coding RNA ファミリーのデータベース。Pfamと同様Stockholm形式で記載され、各エントリはINFERNALパッケージを用いて検索・解析できます。rRNA, tRNA, miRNA, intron, antisense, IRES, riboswitch 等が登録されています。

EMBL Nucleotide Sequence Database

欧州の EMBL (European Molecular Biology Laboratory)-EBIで運営されている塩基配列データベース。ここに含まれる遺伝子を翻訳したものがTrEMBLです。上述のようにGenBankおよびDDBJと連携し登録されている遺伝子情報を共有しています。データベースそのものも EMBL と呼ばれ、組織としてのEMBLと区別をつけるときには EMBL Nucleotide Sequence Database などといわれます。エントリの記載に使われているフォーマットはEMBL形式と呼ばれています。

Ensembl

欧州 EMBL-EBI および Sanger Institute で運営されている真核生物のアノテーションプロジェクト。多くの真核生物のゲノム配列から独自に遺伝子を予測し、機能予測した結果と既存の情報を種ごとに収集し蓄積しています。フラットファイルはGenBank形式とEMBL形式の両方が用意されています。

PDB

Protein Data Bank は国際標準となっているタンパク質の立体構造データベースで、米国 RCSB PDB を中心に欧州 MSD-EBI 、大阪大学 PDBj の連携のもとに運営されています。X線結晶構造解析やNMRなどの実験から得られたタンパク質の各原子の座標情報が蓄積されています。

エントリ

- **HEADER**
  ヘッダー情報。変更日時、エントリ名。
- **TITLE**
  エントリのタイトル
- **COMPND**
  タンパク質名
- **SOURCE**
  由来する生物種
- **KEYWDS**
  キーワード
- **EXPDTA**
  データの種類
- **AUTHOR**
  データ登録者
- **JRNL**
  文献情報
- **REMERK**
  実験条件など
- **SEQREF**
  アミノ酸配列
- **HELIX/SHEET/SSBOND**
  αヘリックス、βシート、ジスルフィド結合の残基の情報
- **ATOM**
  各原子の座標。

参考となるWEBサイト

全てのフォーマットの詳細を網羅することは出来ないので、以下に参考となるサイトをあげておきます。

GenBank

- GenBank
  `http://www.ncbi.nih.gov/Genbank/index.html`
- DDBJによるフォーマットの説明 (日本語)
  `http://www.ddbj.nig.ac.jp/sub/ref10-j.html`
- Feature tableの書式
  `http://www.insdc.org/feature_table.html`

KEGG

- KEGG
  `http://www.genome.jp/kegg/`

UniProt

- UniProt
  `http://us.expasy.org/sprot/`
- Swis-Protマニュアル
  `http://us.expasy.org/sprot/userman.html`
- Expasy (SIBのバイオインフォマティクスサーバー)
  `http://us.expasy.org/`

PROSITE

- PROSITE
  `http://us.expacy.org/prosite/`
- PROSITEマニュアル
  `http://br.expasy.org/prosite/prosuser.html`
- PROFILEマニュアル
  `http://us.expasy.org/txt/profile.txt`

Pfam/Rfam

- Pfam
  `http://www.sanger.ac.uk/Software/Pfam/`
- Rfam
  `http://www.sanger.ac.uk/Software/Rfam/`
- Stockholmフォーマット
  `http://www.cgr.ki.se/cgr/groups/sonnhammer/Stockholm.html`
- HMMER
  `http://hmmer.wustl.edu/`
  `http://bioweb.pasteur.fr/seqanal/motif/hmmer-uk.html` (**解説サイト**)
- INFERNAL
  `http://www.genetics.wustl.edu/eddy/infernal/`

EMBL

- EMBL
  `http://www.embl.org/`
- EMBL- EBI
  `http://www.ebi.ac.uk/`
  `http://www.ebi.ac.uk/embl/`
- EMBL- HEIDELBERG
  `http://www.embl-heidelberg.de/`

Ensembl

- Ensembl
  `http://www.ensembl.org/index.html`

PDB

- wwPDB
  `http://www.wwpdb.org/`
- RCSB PDB
  `http://www.pdb.org/pdb/Welcome.do`
- MSD - EBI
  `http://www.ebi.ac.uk/msd/index.html`
- PDBj
  `http://www.pdbj.org/index_j.html`
- PDBフォーマット (解説サイト)
  `http://www.dna.affrc.go.jp/misc/mm/pdb`

GenBankエントリ

エントリ情報

Feature Table

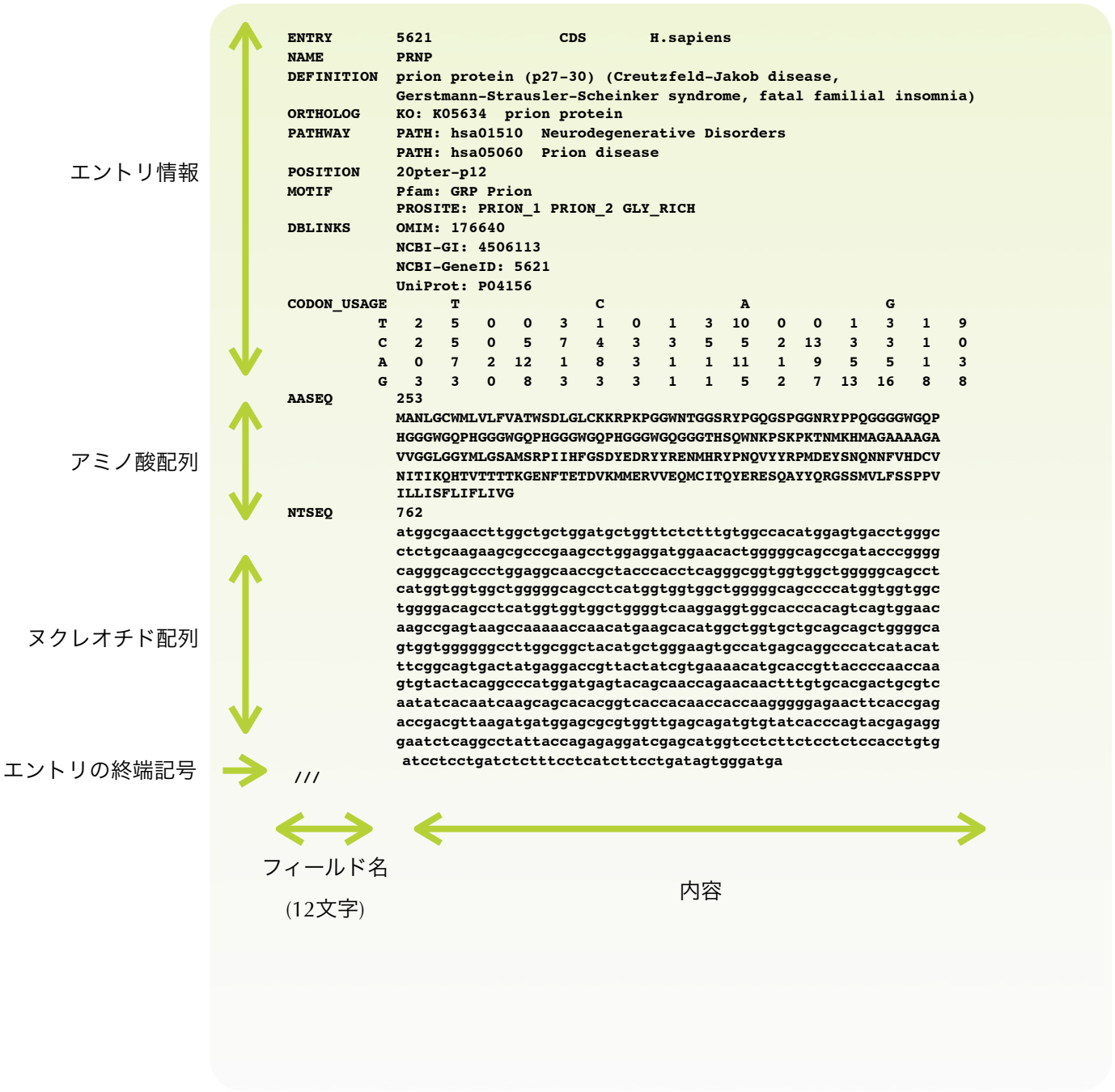
http://www.insdc.org/feature_table.html

配列

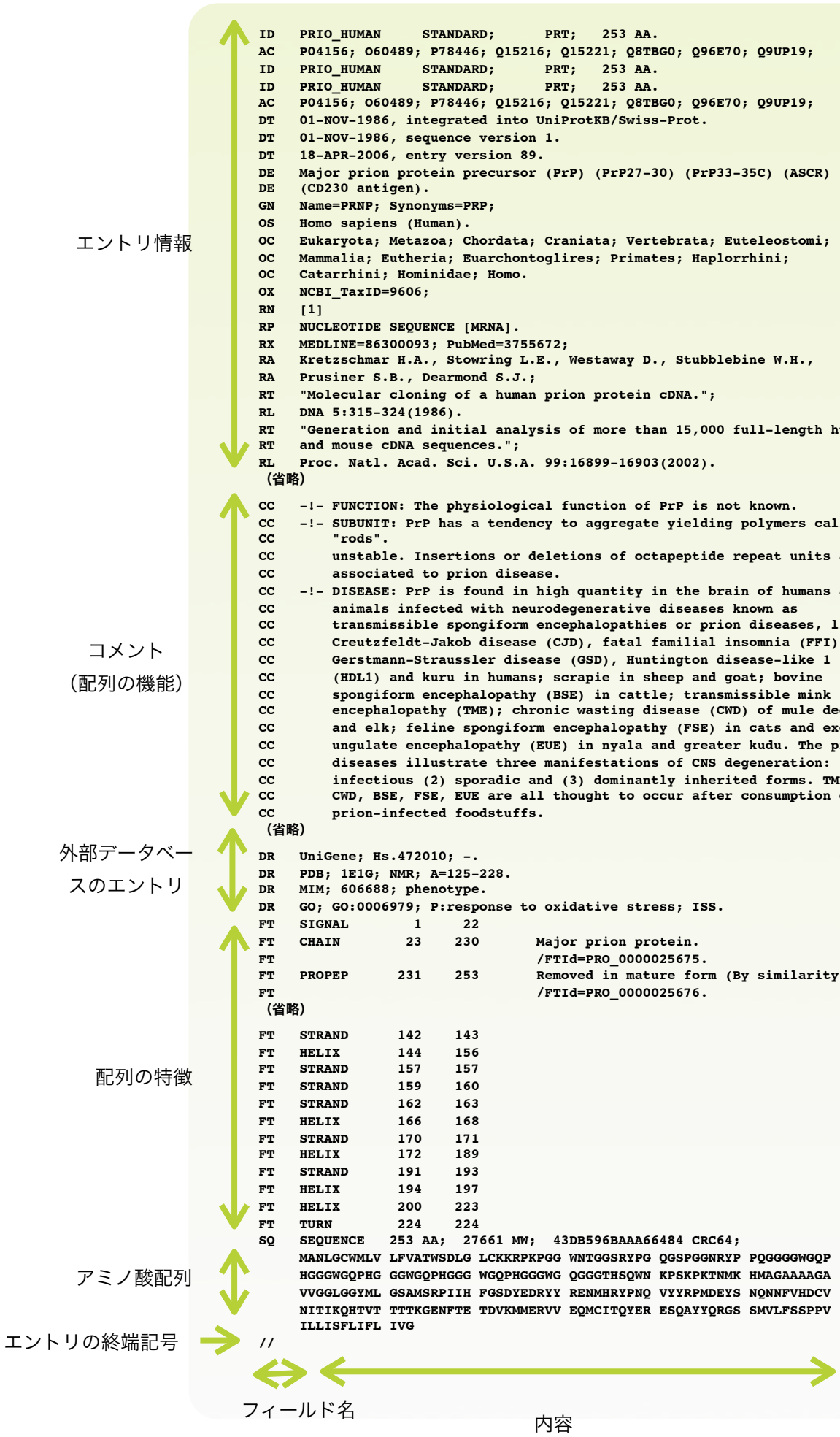
エントリの終端記号

```
LOCUS       NM_000311               2483 bp    mRNA    linear   PRI 07-APR-2006
DEFINITION  Homo sapiens prion protein (p27-30) (Creutzfeld-Jakob disease,
            Gerstmann-Strausler-Scheinker syndrome, fatal familial insomnia)
            (PRNP), transcript variant 1, mRNA.
ACCESSION   NM_000311
VERSION     NM_000311.2  GI:34335258
KEYWORDS    .
SOURCE      Homo sapiens (human)
  ORGANISM  Homo sapiens
            Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
            Mammalia; Eutheria; Euarchontoglires; Primates; Haplorrhini;
            Catarrhini; Hominidae; Homo.
REFERENCE   1  (bases 1 to 2483)
  AUTHORS   Gobbi,M., Colombo,L., Morbin,M., Mazzoleni,G., Accardo,E.,
            Vanoni,M., Del Favero,E., Cantu,L., Kirschner,D.A., Manzoni,C.,
            Beeg,M., Ceci,P., Ubezio,P., Forloni,G., Tagliavini,F. and
            Salmona,M.
  TITLE     Gerstmann-Straussler-Scheinker disease amyloid protein polymerizes
            according to the 'dock-and-lock' model
  JOURNAL   J. Biol. Chem. 281 (2), 843-849 (2006)
   PUBMED   16286452
  REMARK    GeneRIF: PrP82-146 polymerizes according to the 'dock-and-lock'
            model
(省略)
FEATURES             Location/Qualifiers
     source          1..2483
                     /organism="Homo sapiens"
                     /mol_type="mRNA"
                     /db_xref="taxon:9606"
                     /chromosome="20"
                     /map="20pter-p12"
     gene            1..2483
                     /gene="PRNP"
                     /note="synonyms: CJD, GSS, PrP, ASCR, PRIP, PrPc, CD230,
                     MGC26679, PrP27-30, PrP33-35C"
                     /db_xref="GeneID:5621"
                     /db_xref="HGNC:9449"
                     /db_xref="HPRD:01453"
                     /db_xref="MIM:176640"
(省略)
     CDS             101..862
                     /gene="PRNP"
                     /go_component="endoplasmic reticulum; extrinsic to
                     membrane [pmid 16004966]; Golgi apparatus; lipid raft"
                     /go_function="copper ion binding [pmid 16294306];
                     microtubule binding [pmid 16004966]"
                     /go_process="copper ion homeostasis [pmid 16004966];
                     metabolism [pmid 3755672]; response to oxidative stress"
                     /note="prion-related protein; major prion protein; CD230
                     antigen; prion protein PrP"
                     /codon_start=1
                     /product="prion protein preproprotein"
                     /protein_id="NP_000302.1"
                     /db_xref="GI:4506113"
                     /db_xref="GeneID:5621"
                     /db_xref="HGNC:9449"
                     /db_xref="HPRD:01453"
                     /db_xref="MIM:176640"
                     /translation="MANLGCWMLVLFVATWSDLGLCKKRPKPGGWNTGGSRYPGQGSP
                     GGNRYPPQGGGGWGQPHGGGWGQPHGGGWGQPHGGGWGQPHGGGWGQGGGTHSQWNKP
                     SKPKTNMKHMAGAAAAGAVVGGLGGYMLGSAMSRPIIHFGSDYEDRYYRENMHRYPNQ
                     VYYRPMDEYSNQNNFVHDCVNITIKQHTVTTTTKGENFTETDVKMMERVVEQMCITQY
                     ERESQAYYQRGSSMVLFSSPPVILLISFLIFLIVG"
     sig_peptide     101..166
                     /gene="PRNP"
                     /standard_name="D20S1014"
                     /db_xref="UniSTS:21619"
     STS             2105..2344
                     /gene="PRNP"
                     /standard_name="RH70248"
                     /db_xref="UniSTS:43453"
(省略)
ORIGIN
        1 ccccctcggc cccgcgcgtc gcctgtcctc cgagccagtc gctgacagcc gcggcgccgc
       61 gagcttctcc tctcctcacg accgaggcag agcagtcatt atggcgaacc ttggctgctg
      121 gatgctggtt ctctttgtgg ccacatggag tgacctgggc ctctgcaaga agcgcccgaa
      181 gcctggagga tggaacactg ggggcagccg ataccgggg cagggcagcc ctggaggcaa
(省略)
     2161 tattgcatag gacagactta ggagttttgt ttagagcagt taacatctga agtgtctaat
     2221 gcattaactt ttgtaaggta ctgaatactt aatatgtggg aaacccttt gcgtggtcct
     2281 taggcttaca atgtgcactg aatcgtttca tgtaagaatc caaagtggac accattaaca
     2341 ggtctttgaa atatgcatgt actttatatt ttctatattt gtaactttgc atgttcttgt
     2401 tttgttatat aaaaaaattg taaatgttta atatctgact gaaattaaac gagcgaagat
     2461 gagcaccaaa aaaaaaaaaa aaa
//
```

KEGG GENESエントリ



UniProt/Swiss-Protエントリ




```
ID   PRIO_HUMAN     STANDARD;      PRT;   253 AA.
AC   P04156; O60489; P78446; Q15216; Q15221; Q8TBG0; Q96E70; Q9UP19;
ID   PRIO_HUMAN     STANDARD;      PRT;   253 AA.
ID   PRIO_HUMAN     STANDARD;      PRT;   253 AA.
AC   P04156; O60489; P78446; Q15216; Q15221; Q8TBG0; Q96E70; Q9UP19;
DT   01-NOV-1986, integrated into UniProtKB/Swiss-Prot.
DT   01-NOV-1986, sequence version 1.
DT   18-APR-2006, entry version 89.
DE   Major prion protein precursor (PrP) (PrP27-30) (PrP33-35C) (ASCR)
DE   (CD230 antigen).
GN   Name=PRNP; Synonyms=PRP;
OS   Homo sapiens (Human).
OC   Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
OC   Mammalia; Eutheria; Euarchontoglires; Primates; Haplorrhini;
OC   Catarrhini; Hominidae; Homo.
OX   NCBI_TaxID=9606;
RN   [1]
RP   NUCLEOTIDE SEQUENCE [MRNA].
RX   MEDLINE=86300093; PubMed=3755672;
RA   Kretzschmar H.A., Stowring L.E., Westaway D., Stubblebine W.H.,
RA   Prusiner S.B., Dearmond S.J.;
RT   "Molecular cloning of a human prion protein cDNA.";
RL   DNA 5:315-324(1986).
RT   "Generation and initial analysis of more than 15,000 full-length human
RT   and mouse cDNA sequences.";
RL   Proc. Natl. Acad. Sci. U.S.A. 99:16899-16903(2).
（省略）

CC   -!- FUNCTION: The physiological function of PrP is not known.
CC   -!- SUBUNIT: PrP has a tendency to aggregate yielding polymers called
CC       "rods".
CC       unstable. Insertions or deletions of octapeptide repeat units are
CC       associated to prion disease.
CC   -!- DISEASE: PrP is found in high quantity in the brain of humans and
CC       animals infected with neurodegenerative diseases known as
CC       transmissible spongiform encephalopathies or prion diseases, like:
CC       Creutzfeldt-Jakob disease (CJD), fatal familial insomnia (FFI),
CC       Gerstmann-Straussler disease (GSD), Huntington disease-like 1
CC       (HDL1) and kuru in humans; scrapie in sheep and goat; bovine
CC       spongiform encephalopathy (BSE) in cattle; transmissible mink
CC       encephalopathy (TME); chronic wasting disease (CWD) of mule deer
CC       and elk; feline spongiform encephalopathy (FSE) in cats and exotic
CC       ungulate encephalopathy (EUE) in nyala and greater kudu. The prion
CC       diseases illustrate three manifestations of CNS degeneration: (1)
CC       infectious (2) sporadic and (3) dominantly inherited forms. TME,
CC       CWD, BSE, FSE, EUE are all thought to occur after consumption of
CC       prion-infected foodstuffs.
（省略）

DR   UniGene; Hs.472010; -.
DR   PDB; 1E1G; NMR; A=125-228.
DR   MIM; 606688; phenotype.
DR   GO; GO:0006979; P:response to oxidative stress; ISS.
FT   SIGNAL        1     22
FT   CHAIN        23    230       Major prion protein.
FT                                 /FTId=PRO_0000025675.
FT   PROPEP      231    253       Removed in mature form (By similarity).
FT                                 /FTId=PRO_0000025676.
（省略）

FT   STRAND      142    143
FT   HELIX       144    156
FT   STRAND      157    157
FT   STRAND      159    160
FT   STRAND      162    163
FT   HELIX       166    168
FT   STRAND      170    171
FT   HELIX       172    189
FT   STRAND      191    193
FT   HELIX       194    197
FT   HELIX       200    223
FT   TURN        224    224
SQ   SEQUENCE   253 AA;  27661 MW;  43DB596BAAA66484 CRC64;
     MANLGCWMLV LFVATWSDLG LCKKRPKPGG WNTGGSRYPG QGSPGGNRYP PQGGGGWGQP
     HGGGWGQPHG GGWGQPHGGG WGQPHGGGWG QGGGTHSQWN KPSKPKTNMK HMAGAAAAGA
     VVGGLGGYML GSAMSRPIIH FGSDYEDRYY RENMHRYPNQ VYYRPMDEYS NQNNFVHDCV
     NITIKQHTVT TTTKGENFTE TDVKMMERVV EQMCITQYER ESQAYYQRGS SMVLFSSPPV
     ILLISFLIFL IVG
//
```

PROSITEエントリ

PATTERNエントリ

Pattern

エントリの終端記号

```
ID   PRION_2; PATTERN.
AC   PS00706;
DT   DEC-1992 (CREATED); DEC-1992 (DATA UPDATE); APR-2005 (INFO UPDATE).
DE   Prion protein signature 2.
PA   E-x-[ED]-x-K-[LIVM](2)-x-[KR]-[LIVM](2)-x-[QE]-M-C-x(2)-Q-Y.
NR   /RELEASE=46.6,180652;
NR   /TOTAL=66(66); /POSITIVE=66(66); /UNKNOWN=0(0); /FALSE_POS=0(0);
NR   /FALSE_NEG=0; /PARTIAL=0;
CC   /TAXO-RANGE=??E??; /MAX-REPEAT=1;
CC   /SITE=13,disulfide;
CC   /VERSION=1;
DR   P10279, PRIO1_BOVIN, T; P40242, PRIO1_TRAST, T; Q01880, PRIO2_BOVIN, T;
DR   P40243, PRIO2_TRAST, T; Q5UJG1, PRIO_ANTCE , T; P40245, PRIO_AOTTR , T;
DR   P40246, PRIO_ATEGE , T; P51446, PRIO_ATEPA , T; Q5UAF1, PRIO_BISBI , T;
(省略)

DR   P40257, PRIO_PREFR , T; Q95211, PRIO_RABIT , T; P13852, PRIO_RAT   , T;
DR   P40258, PRIO_SAISC , T; P23907, PRIO_SHEEP , T; Q9Z0T3, PRIO_SIGHI , T;
DR   Q95270, PRIO_THEGE , T; Q5UJG3, PRIO_TRAIM , T; P51780, PRIO_TRIVU , T;
3D   1AG2; 1B10; 1DWY; 1DWZ; 1DX0; 1DX1; 1E1G; 1E1J; 1E1P; 1E1S; 1E1U; 1E1W;
3D   1H0L; 1HJM; 1HJN; 1I4M; 1QLX; 1QLZ; 1QM0; 1QM1; 1QM2; 1QM3; 1TQB; 1TQC;
3D   1U3M; 1UW3; 1XYQ; 1XYU; 1XYW; 1XYX; 1Y15; 1Y16; 1Y2S;
DO   PDOC00263;
//
```

RULEエントリ

Rule

```
ID   SULFATION; RULE.
AC   PS00003;
DT   APR-1990 (CREATED); APR-1990 (DATA UPDATE); APR-1990 (INFO UPDATE).
DE   Tyrosine sulfation site.
RU   (1) Glu or Asp within two residues of the tyrosine (typically at -1).
RU   (2) At least three acidic residues from -5 to +5.
RU   (3) No more than 1 basic residue and 3 hydrophobic from -5 to +5
RU   (4) At least one Pro or Gly from -7 to -2 and from +1 to +7 or at least
RU       two or three Asp, Ser or Asn from -7 to +7.
RU   (5) Absence of disulfide-bonded cysteine residues from -7 to +7.
RU   (6) Absence of N-linked glycans near the tyrosine.
CC   /TAXO-RANGE=??E??;
CC   /SKIP-FLAG=TRUE;
CC   /VERSION=1;
DO   PDOC00003;
//
```

MATRIX (PROFILE) エントリ

Matrix (profile)

エントリの終端記号

```
ID   GLY_RICH; MATRIX.
AC   PS50315;
DT   APR-2002 (CREATED); APR-2002 (DATA UPDATE); APR-2002 (INFO UPDATE).
DE   Glycine-rich region profile.
MA   /GENERAL_SPEC: ALPHABET='ABCDEFGHIKLMNPQRSTVWYZ'; LENGTH=2;
MA   /DISJOINT: DEFINITION=PROTECT; N1=1; N2=2;
MA   /NORMALIZATION: MODE=1; FUNCTION=LINEAR; R1=0.9584; R2=0.13706103; TEXT='-LogE';
MA   /CUT_OFF: LEVEL=0; SCORE=55; N_SCORE=8.5; MODE=1; TEXT='!';
MA   /CUT_OFF: LEVEL=-1; SCORE=40; N_SCORE=6.5; MODE=1; TEXT='?';
MA   /DEFAULT: SY_I='G'; I0=-1; M0=-1;
MA   /M: SY='G'; M=-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1;
MA   /I: MI=0; IM=0; I=-1,-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1;
MA   /M: SY='G'; M=-1,-1,-1,-1,-1,-1,5,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1;
CC   /MATRIX_TYPE=composition;
CC   /SCALING_DB=db_global;
CC   /AUTHOR=P_Bucher;
CC   /SKIP-FLAG=TRUE;
CC   /FT_KEY=DOMAIN; /FT_DESC=Gly-rich;
CC   /VERSION=1;
DO   PDOC50099;
//
```

フィールド名

(2+3文字)

内容

Pfamエントリ

エントリ情報
#=GF

各配列の情報
#=GS

マルチプル
アライメント

マルチプルアライメントの
カラムに対する情報

エントリの終端記号

```
# STOCKHOLM 1.0
#=GF ID   Prion
#=GF AC   PF00377.9
#=GF DE   Prion/Doppel alpha-helical domain
#=GF PI   prion;
#=GF AU   Bateman A, Finn RD
#=GF SE   Prosite
#=GF GA   -26.00 -26.00; 11.80 11.80;
#=GF TC   -13.10 -13.10; 13.20 13.20;
#=GF NC   -27.90 -27.90; 11.70 11.70;
#=GF TP   Domain
#=GF BM   hmmbuild -F HMM_ls SEED
#=GF BM   hmmcalibrate --seed 0 HMM_ls
#=GF BM   hmmbuild -f -F HMM_fs SEED
#=GF BM   hmmcalibrate --seed 0 HMM_fs
#=GF AM   globalfirst
#=GF RN   [1]
#=GF RM   96317593
#=GF RT   NMR structure of the mouse prion protein domain
#=GF RT   PrP(121-321).
#=GF RA   Riek R, Hornemann S, Wider G, Billeter M, Glockshuber R,
#=GF RA   Wuthrich K;
#=GF RL   Nature 1996;382:180-182.
(省略)

#=GF DR   PROSITE; PDOC00263;
#=GF DR   PRINTS; PR00341;
#=GF DR   SCOP; 2prp; fa;
#=GF DR   TC; 1.C.48;
#=GF DR   INTERPRO; IPR000817;
#=GF DR   PDB; 1xyx A; 134; 232;
#=GF DR   PDB; 1y15 A; 134; 232;
#=GF DR   PDB; 1y16 A; 134; 232;
#=GF DR   PDB; 1ag2 ; 134; 226;
(省略)

#=GF CC   The prion protein is thought to be the infectious agent that causes
#=GF CC   transmissible spongiform encephalopathies, such as scrapie and BSE.
#=GF CC   It is thought that the prion protein can exist in two different forms:
#=GF CC   one is the normal cellular protein, and the other is the infectious
#=GF CC   form which can change the normal prion protein into the infectious form.
#=GF CC   It has been found that the prion alpha-helical domain is also found
#=GF CC   in the Doppel protein.
#=GF SQ   346
#=GS Q9Z0T5_MERUN/133-252  AC Q9Z0T5
#=GS Q8VHV6_APOSY/134-253  AC Q8VHV6
#=GS PRIO_RAT/134-253      AC P13852
#=GS Q549H6_RAT/134-253    AC Q549H6
#=GS PRIO_SIGHI/134-253    AC Q9Z0T3
#=GS Q9Z0T4_9RODE/134-253  AC Q9Z0T4
(省略)
Q9Z0T5_MERUN/133-252         MSRPMIHFGNDWEDRYYRENMYRYPNQVY...YRPV..DQYS-N···
Q8VHV6_APOSY/134-253         MSRPMIHFGNDWEDRYYRENMYRYPNQVY...YRPV..DQYS-N···
PRIO_RAT/134-253             MSRPMLHFGNDWEDRYYRENMYRYPNQVY...YRPV..DQYS-N···
Q549H6_RAT/134-253           MSRPMLHFGNDWEDRYYRENMYRYPNQVY...YRPV..DQYS-N···
PRIO_SIGHI/134-253           MSRPMIHFGNDWEDRYYRENMYRYPNQVY...YRPV..DQYN-N···
(省略)
#=GR PRND_MOUSE/64-179 SS    CCCCCCCCCHH-HHHHHHHHGGGSCSEEE...CCCX..SSTTSC···
#=GR PRND_MOUSE/64-179 SA    35529170275-03620562372001103...172X..753712···
Q8MIJ1_CYNSP/27-108          GRKLNIDFGDE-GNRYYEAHYWEFPDGIY...YDAC..SKANVT···
#=GC SS_cons                 CCCCCCCCSSHHHHHHHHHHGGGSCSCCEEEEECCC..CSCSCC···
#=GC SA_cons                 27426181757613530562374113202304144l..774614···
#=GC seq_cons                MSRPLIHFGNDYEDRYYRENMYRYPNQVY...YRPV..DQYS.N···
//
```

PDBエントリ

```
HEADER    PRION PROTEIN                           31-MAR-97   1AG2
TITLE     PRION PROTEIN DOMAIN PRP(121-231) FROM MOUSE, NMR,
TITLE    2 MINIMIZED AVERAGE STRUCTURE
COMPND    MOL_ID: 1;
COMPND   2 MOLECULE: MAJOR PRION PROTEIN;
COMPND   3 CHAIN: NULL;
COMPND   4 FRAGMENT: DOMAIN 121 - 231;
COMPND   5 SYNONYM: PRP(121-231);
COMPND   6 ENGINEERED: YES
SOURCE    MOL_ID: 1;
SOURCE   2 ORGANISM_SCIENTIFIC: MUS MUSCULUS;
SOURCE   3 ORGANISM_COMMON: MOUSE;
SOURCE   4 EXPRESSION_SYSTEM: ESCHERICHIA COLI;
SOURCE   5 EXPRESSION_SYSTEM_CELL_LINE: BL21 (DE3);
SOURCE   6 EXPRESSION_SYSTEM_PLASMID: T7 PPRP-C6
KEYWDS    PRION PROTEIN, BRAIN, GLYCOPROTEIN, GPI-ANCHOR
EXPDTA    NMR, MINIMIZED AVERAGE STRUCTURE
AUTHOR    M.BILLETER,R.RIEK,G.WIDER,K.WUTHRICH,S.HORNEMANN,
AUTHOR   2 R.GLOCKSHUBER
REVDAT   1   08-OCT-97 1AG2    0
JRNL        AUTH   R.RIEK,G.WIDER,M.BILLETER,S.HORNEMANN,
JRNL        AUTH 2 R.GLOCKSHUBER,K.WUTHRICH
JRNL        TITL   REFINED NMR STRUCTURE OF THE PRION PROTEIN DOMAIN
JRNL        TITL 2 PRP(121-231) AND INHERITED HUMAN PRION DISEASES
JRNL        REF    TO BE PUBLISHED
JRNL        REFN                                          0353
REMARK   1
REMARK   1 REFERENCE 1
REMARK   1  AUTH   R.RIEK,S.HORNEMANN,G.WIDER,M.BILLETER,
REMARK   1  AUTH 2 R.GLOCKSHUBER,K.WUTHRICH
REMARK   1  TITL   NMR STRUCTURE OF THE MOUSE PRION PROTEIN DOMAIN
REMARK   1  TITL 2 PRP(121-231)
REMARK   1  REF    NATURE                        V. 382   180 1996
REMARK   1  REFN   ASTM NATUAS  UK ISSN 0028-0836               0006
REMARK   2
REMARK   2 RESOLUTION. NOT APPLICABLE.
REMARK   3
REMARK   3 REFINEMENT.
REMARK   3   PROGRAM     : OPAL
REMARK   3   AUTHORS     : LUGINBUHL,GUNTERT,BILLETER,WUTHRICH
REMARK   3
(省略)
SEQRES   1    103  GLY LEU GLY GLY TYR MET LEU GLY SER ALA MET SER ARG
SEQRES   2    103  PRO MET ILE HIS PHE GLY ASN ASP TRP GLU ASP ARG TYR
SEQRES   3    103  TYR ARG GLU ASN MET TYR ARG TYR PRO ASN GLN VAL TYR
SEQRES   4    103  TYR ARG PRO VAL ASP GLN TYR SER ASN GLN ASN ASN PHE
SEQRES   5    103  VAL HIS ASP CYS VAL ASN ILE THR ILE LYS GLN HIS THR
SEQRES   6    103  VAL THR THR THR THR LYS GLY GLU ASN PHE THR GLU THR
SEQRES   7    103  ASP VAL LYS MET MET GLU ARG VAL VAL GLU GLN MET CYS
SEQRES   8    103  VAL THR GLN TYR GLN LYS GLU SER GLN ALA TYR TYR
HELIX    1   1 ASP    144  ASN    153  1                                  10
HELIX    2   2 GLN    172  LYS    194  1                                  23
HELIX    3   3 GLU    200  ALA    224  1                                  25
SHEET    1  S1 2 TYR   128  GLY   131  0
SHEET    2  S1 2 VAL   161  ARG   164 -1
SSBOND   1 CYS    179    CYS    214
CRYST1    1.000    1.000    1.000  90.00  90.00  90.00 P 1           1
ORIGX1      1.000000  0.000000  0.000000        0.00000
ORIGX2      0.000000  1.000000  0.000000        0.00000
ORIGX3      0.000000  0.000000  1.000000        0.00000
SCALE1      1.000000  0.000000  0.000000        0.00000
SCALE2      0.000000  1.000000  0.000000        0.00000
SCALE3      0.000000  0.000000  1.000000        0.00000
ATOM      1  N   GLY   124      -9.861  12.581   5.804  1.00  1.00           N
ATOM      2  CA  GLY   124     -10.625  12.191   4.609  1.00  1.00           C
ATOM      3  C   GLY   124      -9.855  12.324   3.299  1.00  1.00           C
ATOM      4  O   GLY   124      -9.952  11.443   2.449  1.00  1.00           O
ATOM      5 1H   GLY   124      -9.521  13.531   5.711  1.00  1.00           H
ATOM      6 2H   GLY   124     -10.470  12.563   6.611  1.00  1.00           H
(省略)
ATOM   1665  HE2 TYR   226      -4.510  -8.012   1.047  1.00  1.00           H
ATOM   1666  HH  TYR   226      -4.456  -7.286  -2.488  1.00  1.00           H
TER    1667      TYR   226
CONECT  900  899 1459
CONECT 1459  900 1458
MASTER       61    0    0    3    2    0    0    6 1666    1    2    8
END
```

アミノ酸配列

2次構造情報

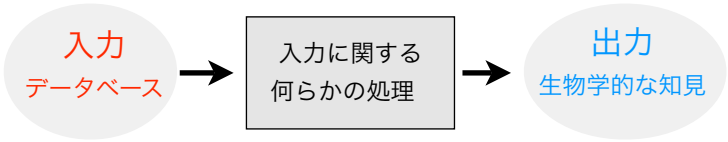
タンパク質の各原子の空間座標

エントリの終端記号

フラットファイルDBの利用法

生物学的な知見を得る

さて、ここまで主だったフラットファイルデータベースを紹介してきましたが、これらは実験結果やこれまでの生物学的な知見をプログラムから読み込みやすくするために蓄積したものです。既存のデータはプログラムでは入力に相当します。バイオインフォマティクス研究で使用するプログラムの多くは、これらのフラットファイルデータベースを直接・間接的に読み込み、もとめる生物学的な知見を取り出すことが目的となります。



1) 入力＝ゲノム配列 (GenBank)
　出力＝GCコンテンツ
2) 入力＝遺伝子配列 (GenBank)
　出力＝タンパク質配列
3) 入力＝機能未知配列 +
　　全タンパク質配列 (KEGG GENES)
　出力＝類似配列
4) 入力＝プリオンの特徴配列 (PROSITE) +
　　タンパク質配列 (UniProt/Swiss-Prot)
　出力＝プリオンの特徴をもつタンパク質のリスト
5) 入力＝ SH2ドメイン (Pfam)
　　真核生物の遺伝子配列 (Ensembl)
　出力＝SH2ドメインを持つ全タンパク質のリスト
6) 入力＝生命知識情報 (KEGG)
　出力＝論文

入力に関する何らかの処理

バイオインフォマティクスは、入力に対しどのような処理を加えれば生物学的な知見が得られるか、を研究する分野であるともいえます。 例えば、3) の例の「機能未知配列の類似配列を KEGG GENES データベースから抽出する」という問題ひとつとっても、長い研究の歴史があり一筋縄ではいきません。本実習では「どのような手法・処理が必要で、それをどのように実現するか」にはあまり触れることはできませんが、すべてのバイオインフォマティクスプログラミングで重要になるフラットファイルデータベースからの入力と簡単なフィルタリングを課題としてプログラムを学んでいきます。

　というわけで、バイオインフォマティクスで用いるプログラムで基本となるのは入出力です（必ずしもバイオインフォマティクスに限ったことではありませんが）。冒頭で、フラットファイルデータベースは「誰にでも容易に利用可能であるというメリットがある」と述べましたが、実はプログラムから利用するには、少し障壁があります。今回、紹介した主だったフラットファイル形式のデータベースは、可読性をあげるため等の理由であまり単純ではないルールで記述されていて、それを解釈しないとプログラムから読み込むことができません。例えば、

4) 入力＝プリオンの特徴配列 (PROSITE) +
　　タンパク質配列 (UniProt/Swiss-Prot)
　出力＝プリオンの特徴をもつタンパク質のリスト

の例についてですが、これを実現するにも

I. PROSITE のエントリを読み込みPAで始まる行の5文字目から最後までをPATTERNとして解釈する。
II. UniProt/Swiss-Prot を読み込み、エントリのSQの次の列から // までの空白以外の文字をアミノ酸配列として取り込む。
III. アミノ酸配列中に PATTERN に合致する配列があるか調べる。あったら出力する。
IV. II., III.の過程を繰り返す。

という手続きが必要になります。このとき、I. II. のデータを読み込む過程でファイルの文字列を調べ、必要とする部分だけを取りだしてくる必要があります。プログラミング言語Rubyにはそのために利用できる便利な手法があらかじめ用意されています。以降の実習ではその手法を紹介していきます。

Rubyのおやくそく

ARGVとARGF

By Toshiaki Katayama

ファイル入出力

ファイル入出力(I/O)は、データの読み込み、結果の書き出しという意味で、特にスクリプト系のプログラムでは基本的な機能です。このためRubyでも入出力の方法には様々なやり方がありますが、まずはフラットファイルデータベースのエントリを読み込んでみましょう。

読み込むファイル名は、プログラムに埋め込まず、コマンドライン引数で渡すようにします。これにより、ファイル名が変わってもプログラムを変更する必要がなくなりますし、複数のファイルを入力に使うこともできるようになります。

ARGVとコマンドライン引数

まずは、コマンドライン引数の受け取り方を見てみます。以下の実行例では、argv.rb というスクリプトに、意味の無い２つのコマンドライン引数hogeと123を渡しています（通常はファイル名などを渡します）。

```
% argv.rb hoge 123
hoge
123
```

Rubyではコマンドライン引数が配列としてARGVという定数に格納されます。そのため、実行例のように受け取った引数を１行ずつ表示するには、次のようなプログラムを書けば良いことになります。ここでは、ARGVのそれぞれの要素を変数argに取り出して、その値をputsで表示しています。

```
#!/usr/bin/env ruby

ARGV.each do |arg|
  puts arg
end
```

ARGVを使ったプログラムでファイルの中身を読み出すには、File.openを使う必要があります。file1, file2を順に読み込んで中身を表示するプログラムを作ってみましょう。

```
#!/usr/bin/env ruby

ARGV.each do |filename|
  File.open(filename) do |file|
    file.each do |line|
      puts line
    end
  end
end
```

このプログラムは、コマンドライン引数で与えられたファイル名を１つずつfilename変数に代入します。次にFile.openを用いてファイルを開き、eachメソッドでline変数に１行ずつ読み込みます。読み込まれた行はそのままputsで表示しています。これを引数で与えられた全てのファイルについて繰り返します。

上記のプログラムはUNIXのcatコマンドと同等なのでcat-argv.rbとします。

```
% cat-argv.rb file1 file2
```

file1, file2 の中身が表示されたと思います。

ARGFとフィルタ

上記のように、引数で指定したファイルの中身を順に処理するプログラムをフィルタと呼びます。UNIXのcatコマンドは最も単純なフィルタで、他にgrep, awk, sed, perlなど様々なコマンドがあります。

ARGFを使うとこのようなフィルタプログラムを容易に書くことができます。ARGFはARGVの各要素をファイル名として扱い、中身を順番につなげたような仮想ファイルです。

ARGFとファイル

ARGFは、コマンドライン引数で与えられた1つ以上のファイルを透過的に結合して、仮想的な1つのファイルとして扱います。ARGFを使用すると、プログラムにファイル名を埋め込まずに済むため、汎用性の高いスクリプトを書くことができます。
一方、現在処理しているファイルに関する情報が必要な場合もあるでしょう。

現在開いているファイル

現在開いているファイルの名前は `filename` メソッドで得られます。

```
ARGF.filename
```

ファイル中の行番号

ARGF全体ではなく、現在開いているファイル中での行番号は

```
ARGF.file.lineno
```

で得られます。

ファイルの終端

開いている全てのファイルを読み込み終わったかどうかの判定は

```
ARGF.eof?
```

を使うことができます。

実行ファイル名

実行中のスクリプトの名前は`__FILE__`で得られ、コマンドラインで指定したスクリプト名は `$0` で得られます。両者を比較すると、ライブラリとして require されているのか、直接実行されているのかを判定できます。
また、スクリプトのディレクトリ名は

```
dir = File.dirname(__FILE__)
```

で得られることから、スクリプトと同じディレクトリのファイルは

```
File.expand_path('otherfile', dir)
```

として指定できます。

ARGFを用いてcat-argv.rbを書き直したプログラムcat-argf.rbは以下のようになります。

```
#!/usr/bin/env ruby

ARGF.each do |line|
  puts line
end
```

実行の仕方は同じです。

```
% cat-argf.rb file1 file2
```

ついでに、行番号を表示するように変更してみましょう。以下のプログラムcat-count.rbでは1行読み込むごとに変数countの値を1増やして、行番号とともにファイルの中身を表示します。

```
#!/usr/bin/env ruby

count = 0

ARGF.each do |line|
  count += 1
  puts "#{count}:#{line}"
end
```

シェルでは、パイプ機能を用いることで、ファイルだけでなく標準入出力からもデータを受け渡すことができます。この場合、ARGVを使ったプログラムでは対応できませんが、ARGFを使っていれば大丈夫です。

```
% head file1 | cat-count.rb
```

この例では、headコマンドの出力を受け取り、行番号をつけて表示しています。

標準入力、出力、エラー出力

ここでできた標準入力は、UNIXでは標準出力、標準エラー出力との3点セットになっています。Rubyではそれぞれ定数STDIN, STDOUT, STDERRで表され、対応するIOクラスのオブジェクトが格納されています（グローバル変数 $stdin, $stdout, $stderrの初期値にもこれらのオブジェクトが対応していますが、変数なので置き換えられることもあります）。

標準入力はすでに見たように、パイプなどからのリダイレクトを受け付けます。標準出力は通常画面に表示されていますが、パイプで別のプログラムの標準入力としたり、プログラムの実行結果をファイルに保存したい場合などにリダイレクトされます。

```
% cat-count.rb file1 > file3
```

この場合は、file1に行番号をつけた出力を、画面に表示する代わりにfile3というファイルに保存しています。

```
% cat-count.rb file1 | head
```

この場合は、file1に行番号をつけた結果をheadコマンドの標準入力に渡しています。

一方で、Rubyの中から標準出力に結果を出力したりファイルに出力したり制御したい場合には、STDOUT, STDERRなどを利用することになります。

```
#!/usr/bin/env ruby

odd  = File.open("o.txt", "w")
even = File.open("e.txt", "w")

STDERR.puts "Start"

count = 0
ARGF.each do |line|
  count += 1
  if count % 2 == 0
    even.puts line
  else
    odd.puts line
  end
  if line[/^#/]
    STDOUT.puts line
  end
end

STDERR.puts "Finish"
```

このプログラムは偶数行（countが2で割り切れる）をファイルe.txtに、奇数行をo.txtに出力します。また、行の先頭が#ではじまる行は標準出力STDOUTにも表示します。開始と終了の合図は標準エラー出力STDERRに表示されます。

正規表現

Regexp.new

By Masumi Itoh, Toshiaki Katayama

正規表現とは

正規表現は、文字列中から特定のパターンを持つ部分を探すために使います。PROSITEデータベースのPATTERNなどアミノ酸配列やDNA配列のモチーフも正規表現で表すことができます。正規表現の書式はプログラミング言語によって多少異なります。次ページの表に Ruby の主な正規表現を載せましたので参考にしてください。

正規表現の記述方法

Rubyでは２つの / 記号で囲んだ文字列は正規表現として扱われます。例えば、「文字列のはじめの > 記号」という正規表現は、

```
/^>/
```

となります。ここでは、行頭にマッチする ^ という記号を用いています。文末にマッチするという記号は $ で、「文末の TGA 」という正規表現は、

```
/TGA$/
```

となります。文字列が正規表現にマッチするかどうか（その正規表現に対応する部分文字列があるか）を調べるには [] メソッドを用います。例えば文字列 line が「先頭に >記号があり、次が数字」というパターンにマッチするかどうかを調べたいときは、

```
line[/^>\d/]
```

となります。ここで \d は任意の数字１文字にマッチする記号です。結果として、lineが正規表現にマッチする場合、マッチした部分文字列が返ります。マッチしなかった場合は nil が返ります。従って、この行を if 文の条件にしてプログラムを分岐することができます。

それでは、irbで試してみましょう。まずは、上の /^>\d/ の例をいろいろな文字列で試してみましょう。

```
% irb --simple-prompt
>> str1 = ">320"
==> ">320"
>> str1[/^>\d/]
=> ">3"
>> str2 = ">X320"
==> ">X320"
>> str2[/^>\d/]
=> nil
```

このとき、マッチした一部だけをあとで取り出すために指定しておくことが出来ます。そのためには正規表現の該当する部分を () で囲ってやります。/^>(\d)/ とするとマッチした数字の部分だけがとれます。文字列から[]メソッドを使って取り出すときには、いくつ目の () かを指定するために、

```
line[/^>(\S+) (.*)/, 1]
```

のように数字を指定します。

```
>> str = ">eco:b0001 thrL; thr operon ..."
>> str[/^>\S+/]
=> ">eco:b0001"
>> str[/^>(\S+)/, 1]
=> "eco:b0001"
>> str[/^>(\S+) (\S+);/, 2]
=> "thrL"
```

こうなると、GENESのエントリ行にマッチする正規表現を書いてエントリ名だけを取り出せそうな気がします。

```
ENTRY       b0002           CDS       E.coli
NAME        thrA, Hs, thrD, thrA2, thrA1
DEFINITION  bifunctional aspartokinase I/homeserine
            dehydrogenase I [EC:2.7.2.4 1.1.1.13]
ORTHOLOG    KO: K00928  aspartate kinase
  :
///
```

GENESのエントリ行は先頭にENTRYとありスペースが続いてエントリ名がきて、また、スペースがきます。

先頭にENTRY → ^ENTRY

連続するスペース → \s+

エントリ名(スペース以外) → \S+

バックスラッシュ

バックスラッシュ (\) を表す文字コードに対し、日本では円記号 (¥) が当てられています。どちらもコンピュータの内部では同じ文字 (ASCIIコード 0x5c) なので、画面に表示される文字が ¥ の場合は本文を適宜読み替えてください。

これを参考に、GENESのエントリ名 (b0002) を抜き出す正規表現を作ってみましょう。

```
#!/usr/bin/env ruby

ARGF.each do |line|
  if entry_id = line[          ]
    puts entry_id
  end
end
```

また、種名 (E.coli) を抜き出すにはどうしたらよいでしょうか？

以下は、エントリ名と種名のどちらにするかを引数で選択できるようにした例です。

```
#!/usr/bin/env ruby

num = ARGV.shift.to_i || 1
re = /^ENTRY\s+(\S+)\s+\S+\s+(\S+)/

ARGF.each do |line|
  if substr = line[re, num]
    puts substr
  end
end
```

正規表現	説明	パターン	マッチ
^	行の先頭にマッチ	/^>/	'>eco:b0002<2>'
$	行の末尾にマッチ	/TGA$/	'...ATCTGAAAGATGA'
.	任意の１文字にマッチ	/ubq./	'ubqA: ubiquitin A'
*	直前の正規表現の０回以上の繰り返しにマッチ（貪欲マッチ）	/GGA*/ /GA*/	'ATGGAAAAATTA...' 'ATGGAAAAATTA...'
*?	直前の正規表現の０回以上の繰り返しにマッチ（最短マッチ）	/GGA*?/ /GA*?/	'ATGGAAAAATTA...' 'ATGGAAAAATTA...'
+	直前の正規表現の１回以上の繰り返しにマッチ（貪欲マッチ）	/GGA+/ /GA+/	'ATGGAAAAATTA...' 'ATGGAAAAATTA...'
+?	直前の正規表現の１回以上の繰り返しにマッチ（最短マッチ）	/GGA+?/ /GA+?/	'ATGGAAAAATTA...' 'ATGGAAAAATTA...'
{n}	直前の正規表現のn回の繰り返しにマッチ	/A{3}/	'GAATACAAAAAGAAA'
{m,n}	直前の正規表現のm回からn回の繰り返しにマッチ	/A{1,3}/	'GAATACAAAAAGAAA'
\d	数字にマッチ（[0-9]と同じ）	/2\.7\.1\.\d+/	'EC 2.7.1.12'
\s	空文字列（スペース、タブ、改行など）にマッチ	/ENTRY\s+b0002/	'ENTRY b0002 ...'
\S	\s 以外（空文字列以外）にマッチ	/ENTRY\s+\S+/	'ENTRY aq_288 ...'
\w	普通の文字（word）に使われる文字にマッチ（[A-z0-9_]と同じ）	/\w{3,4}:/	'eco:b0002' 'dcin:234641'
\W	\w 以外（普通の文字以外）にマッチ	/\W/	'eco:b0002'

ワークブック

grepコマンドの作成

By Toshiaki Katayama

フィルタと正規表現

今回学んだフィルタと正規表現を用いてgrepコマンドをRubyで作成してみましょう。まずは、ファイルから１行ずつ読み込むスクリプトからはじめます。ファイル名はgrep.rbとしました。

```
#!/usr/bin/env ruby

      .each do |line|
  puts 
end
```

つぎに、正規表現にマッチした行だけを表示するようにします。正規表現は仮に「行の先頭が > ではじまっているもの」とします。

```
#!/usr/bin/env ruby

re = /^>/

ARGF.each do |line|
  
    line[re]
    puts line
  
end
```

最後に、最初の引数としてコマンドラインで与えた任意のパターンを、正規表現として利用できるように変更します。

```
#!/usr/bin/env ruby

pattern =      .shift
re = Regexp.new(          )

ARGF.each do |line|
  if line[re]
    puts line
  end
end
```

「プログラミング言語は我々の思考習慣に深い影響を与える道具である」

構造化プログラミングの父
Edsger Wybe Dijkstra

このプログラムの使用方法は grep コマンドと同じですが、Ruby の強力な正規表現が使える点が便利です。

```
% grep.rb '(Ser|Tyr).*kin' file
```

最終的な grep.rb は以下のようになります。

```
#!/usr/bin/env ruby

pattern = ARGV.shift
re = Regexp.new(pattern)

ARGF.each do |line|
  if line[re]
    puts line
  end
end
```

さらに、grep コマンドの -i や -v オプションも実装してみるとよいでしょう。オプション解析には Ruby 標準添付の getoptlong.rb などが利用できます（他には optparse.rb, parsearg.rb, getopts.rb など）。

第３回 データリソース活用 II
正規表現の応用とオブジェクト指向

By Toshiaki Katayama

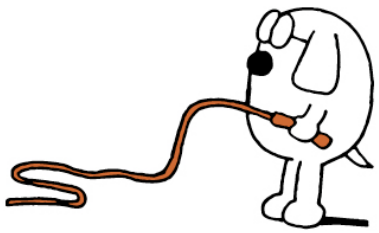
コンテンツ

1. 正規表現の応用
   1.1. PROSITEパターンのルール
   1.2. 正規表現への変換と検索
        （文字列置換のメソッド）
2. オブジェクト指向とライブラリ
   2.1. クラスの作成
   2.2. ライブラリの利用

第３回目は、前回の応用としてPROSITEデータベースを扱います。まず、パターンを正規表現に変換してみます。次に、任意の配列に対しそのパターンが見つかるかどうか検索してみましょう。便利なプログラムができたら、類似のことを行う場合に再利用したいものです。このため、プログラムでよく使う部分を整理して集めたものをライブラリといいます。少し大規模なライブラリを開発する場合には、オブジェクト指向が役に立ちます。ものごとを抽象的に捉えることで、汎用的なライブラリを作成できるからです。また、ライブラリが増えてきたときにも、階層的に名前空間を分けておくことで、あまり破綻せず整理整頓できます。簡単な例として、化合物のデータを扱うライブラリを作成してみます。

"Function means nothing. Design is everything. Quality is yesterday's news. Today we focus on the emotional impact of the product"

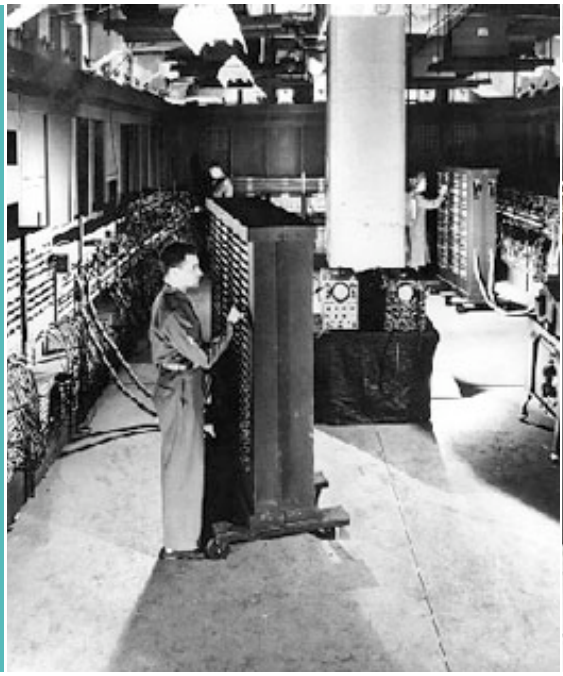
Dogbert (2004/8/17)
コミックDilbertのペット犬



PROSITE パターンのルール

前回見たように、PROSITEはモチーフのデータベースで、RULE, PATTERN, MATRIX の３種類のエントリからなります。このうち、PATTERNは配列に出現するモチーフが正規表現に似た文法で書かれたものでした。

コンピュータ
左は1946年に作られた、世界最初のコンピュータ ENIAC の写真。右は2005年末の時点で世界最速のスパコンと認定されたIBM の BlueGene/L。



```
ID   PRION_2; PATTERN.
AC   PS00706;
DT   DEC-1992 (CREATED); DEC-1992 (DATA UPDATE); APR-2005 (INFO UPDATE).
DE   Prion protein signature 2.
PA   E-x-[ED]-x-K-[LIVM](2)-x-[KR]-[LIVM](2)-x-[QE]-M-C-x(2)-Q-Y.
NR   /RELEASE=46.6,180652;
NR   /TOTAL=66(66); /POSITIVE=66(66); /UNKNOWN=0(0); /FALSE_POS=0(0);
NR   /FALSE_NEG=0; /PARTIAL=0;
CC   /TAXO-RANGE=??E??; /MAX-REPEAT=1;
CC   /SITE=13,disulfide;
CC   /VERSION=1;
DR   P10279, PRIO1_BOVIN, T; P40242, PRIO1_TRAST, T; Q01880, PRIO2_BOVIN, T;
DR   P40243, PRIO2_TRAST, T; Q5UJG1, PRIO_ANTCE , T; P40245, PRIO_AOTTR , T;
DR   P40246, PRIO_ATEGE , T; P51446, PRIO_ATEPA , T; Q5UAF1, PRIO_BISBI , T;
(省略)

DR   P40257, PRIO_PREFR , T; Q95211, PRIO_RABIT , T; P13852, PRIO_RAT   , T;
DR   P40258, PRIO_SAISC , T; P23907, PRIO_SHEEP , T; Q9Z0T3, PRIO_SIGHI , T;
DR   Q95270, PRIO_THEGE , T; Q5UJG3, PRIO_TRAIM , T; P51780, PRIO_TRIVU , T;
3D   1AG2; 1B10; 1DWY; 1DWZ; 1DX0; 1DX1; 1E1G; 1E1J; 1E1P; 1E1S; 1E1U; 1E1W;
3D   1H0L; 1HJM; 1HJN; 1I4M; 1QLX; 1QLZ; 1QM0; 1QM1; 1QM2; 1QM3; 1TQB; 1TQC;
3D   1U3M; 1UW3; 1XYQ; 1XYU; 1XYW; 1XYX; 1Y15; 1Y16; 1Y2S;
DO   PDOC00263;
//
```

上にプリオンのPATTERNエントリの例を再掲しました。

PROSITEエントリのPA行に書かれるパターンの仕様はマニュアル prosuser.txt によると以下のようになっています。

1. Ambiguities (複数のアミノ酸のうちどれか一つ、という場合) は角カッコ [] の中に並べる
   例：[ALT]
   意味：Ala か Leu か Thr

2. パターンはピリオドで終わる

3. パターンがN末にあることを示すには<記号を、C末にあることを示すには>記号をつける

4. Ambiguitiesで、いくつかのアミノ酸以外を示す場合は中カッコ {} の中に並べる
   例：{AM}
   意味：Ala と Met 以外

5. 繰り返しは(m)でm回、または(m, n)で m回からn回を用いて表現する
   例：x(2,4)
   意味：x-x か x-x-x か x-x-x-x

6. 文字 x はどのアミノ酸でもよいことを示す

7. 各要素（アミノ酸）は - でつなげる
   例：<A-x-[ST](2)-x(0,1)-{ED}.
   意味：N 末にある Ala-any-[Ser か Thr]-[Ser か Thr]-any(0〜1残基)-{Glu と Asp 以外}.

これをふまえて、プリオンのパターンを解釈してみてください。

正規表現への変換と検索

それでは、PROSITE PATTERN を Ruby の正規表現に変換する関数（メソッド）を作ってみましょう。メソッドは def 文で定義します。パターンを受け取って正規表現で返すことから pa2re という名前にしました。

```
def pa2re(pat)
end
```

受け取ったpatを処理していきます。空白文字があればゴミなので先に消しておきます。

```
pat.gsub!(/\s/, '')
```

ルール１はRubyの正規表現と同じなのでルール２から処理しましょう。行末の . は正規表現としては不要なので削除します。

```
pat.sub!(/\.$/, '')
例：<A-x-[ST](2)-x(0,1)-{ED}.
→：<A-x-[ST](2)-x(0,1)-{ED}
```

文字列の置換

String#sub
String#sub!

subとsub! は文字列中で最初にパターンにマッチする部分を置換します。sub は置換後の新しい文字列を生成して返しますが、sub!は対象の文字列そのものを変更します。

例：

```
str = "hoge"
str.sub(/h/,"m")
puts str #=> "hoge"
str.sub!(/h/,"m")
puts str #=> "moge"

s = str.sub(/h/,"m")
puts str #=> "hoge"
puts s   #=> "moge"
```

置換後の文字列を動的に生成する場合や () でマッチした部分文字列を $1, $2, .. で取り出すにはブロックが便利です。

例：

```
str = "hoge"
str.sub(/h/) { |m|
  ">" + m.upcase
}
puts str #=> ">Hoge"
```

String#gsub
String#gsub!

gsubとgsub! は文字列中でパターンにマッチする部分を全て (global) 置換します。それ以外の点は sub, sub! と同じです。

String#tr
String#tr!

trとtr! は文字列中で文字を検索し１文字ずつ対応する文字に置換します。正規表現は使えませんが、文字の範囲を指定して置換できることや、(おそらく) subより高速な点が特徴です。

例：

```
str = "hoge"
str.tr!("a-z","A-Z")
puts str #=> "HOGE"
```

次はルール３です。N末、つまり配列の先頭にマッチさせればよいので ^ を使います。

```
pat.sub!(/^</, '^')
例：^A-x-[ST](2)-x(0,1)-{ED}
意味：<★ → /^★/
```

C末は行の末尾なので $ を使います。

```
pat.sub!(/>$/, '$')
意味：★> → /★$/
```

ルール４を変換します。「〜以外」はRubyでは [] の文字セットの先頭に ^ をつけるのでした。

```
pat.gsub!(/\{(\w+)\}/) {
  '[^' + $1 + ']'
}
例：^A-x-[ST](2)-x(0,1)-[^ED]
意味：{★} → /[^★]/
```

ルール５を変換します。繰り返し回数の指定はカッコの種類が違うだけですね。ルール４のあとに変換する必要があります。

```
pat.gsub!(/\(([\d,]+)\)/) {
  '{' + $1 + '}'
}
例：^A-x-[ST]{2}-x{0,1}-[^ED]
意味：★(m,n) → /★{m,n}/
```

ルール６を変換します。任意の文字は . ですね。これはルール２の後でないとダメです。

```
pat.tr!('x', '.')
例：^A-.-[ST]{2}-.{0,1}-[^ED]
意味：x → /./
```

ルール７を変換します。文字列を繋ぐ - は不要なので削ります。

```
pat.tr!('-', '')
例：^A.[ST]{2}.{0,1}[^ED]
意味：▲-★-▼→ /▲★▼/
```

ちなみに、sub, sub!, gsub, gsub!, tr, tr! などは文字列を置換するためのメソッドです。

最期に、出来上がった文字列 pat を正規表現オブジェクトに変換します。

```
Regexp.new(pat)
意味：/^A.[ST]{2}.{0,1}[^ED]/
```

では、PROSITEのパターンと配列だけが入ったファイルを与えてマッチするかどうか調べるプログラム pat_match.rb を作ってみましょう。任意の PROSITE エントリを FASTA 形式の全ての配列に対して検索するようなプログラムは次回紹介する BioRuby などを用いると簡単に書くことができます。

```
#!/usr/bin/env ruby

pattern = ARGV.shift.dup

def pa2re(pat)
  pat.gsub!(/\s/, '')
  pat.sub!(/\.$/, '')
  pat.sub!(/^</, '^')
  pat.sub!(/>$/, '$')
  pat.gsub!(/\{(\w+)\}/) {
    '[^' + $1 + ']'
  }
  pat.gsub!(/\(([\d,]+)\)/) {
    '{' + $1 + '}'
  }
  pat.tr!('x', '.')
  pat.tr!('-', '')
  Regexp.new(pattern)
end

# 正規表現に変換
re = pa2re(pattern)

# 配列の読み込み
seq = ARGF.read
seq.gsub!(/\s+/, '')

# マッチするかどうかチェック
if seq[re]
  puts "Match!"
end
```

KEGG GENES の hsa:5621 のアミノ酸配列を prion.seq に保存して実行してみましょう。

```
% pat_match.rb 'E-x-[ED]-x-K-[LIVM]
  (2)-x-[KR]-[LIVM](2)-x-[QE]-M-C-x(2)-Q-Y.'
  prion.seq
```

オブジェクト指向入門
クラスと抽象化

By Toshiaki Katayama

オブジェクト指向とは

オブジェクト指向プログラミングについての概念的・歴史的な解説はGoogleや書籍で調べて頂くとして、現実的にはプログラムが長くなってきた時に、人間が理解しやすくするためにコードをいかに整理するか、という問題の一つの解決策だと考えることができます。

同じコードを繰り返し書かないために、アルゴリズムを抽象化してまとめる構造化プログラミングを押し進め、扱うデータに共通する性質や手続きを抽象化したプログラミング手法といえます。

クラスと差分プログラミング

扱う対象（オブジェクト）に共通の性質を抽象化したものをクラスといいます。個々のオブジェクトはそのクラスのインスタンス（具体例）とよばれます。

抽象化には階層性がありますので、上位のクラスの性質を共通に持ちながら、より具体的な下位のクラス群を考えることができます。オブジェクト指向では、上位のクラスを継承して差分だけプログラミングすることによって重複を省くことが可能です。

一方で、人によってモノゴトの捉え方や説明の仕方が違うように、 他人による対象の抽象化（クラスの定義）が分かりやすいかどうかはセンスの問題ともいえるので、クラスの設計には本質を見抜く努力が必要です。

メソッドとカプセル化

オブジェクト指向では、対象の持つ性質の抽象化に加え、対象の属性（データ）に対する操作（メソッド）をクラスに一元化することで、プログラムに意味的なまとまりをもたせることが可能です。メソッドは、外部への公開インターフェイスと、プログラムの都合で内部用に作られたものに区別されます。オブジェクト内部のデータは公開メソッドだけを通じて操作するので、勝手にいじられたくないデータはカプセル化によって情報が隠蔽されています。さらに、公開メソッドの挙動さえ変更しなければ、内部用のメソッドの改良や機能追加による悪影響はありません。

「抽象化は問題のある側面を選択的に検討することである。抽象化の目標は、ある目的にとって重要な側面を分離抽出し、重要でない側面を捨て去ることである。抽象化は常にある目的に対するものでなければならない。なぜならその目的によって、重要なもの重要でないものが決定されるからである。同じ物事に対してその目的に応じて異なった多くの抽象化が可能である」

James Rumbaugh
オブジェクトモデリング手法の開発者

なぜオブジェクト指向？

結局のところ、オブジェクト指向プログラミングを行う最大の理由は、名前から想像されるほど大げさなものではなく、単にプログラムの設計や整理整頓・再利用に便利だからでしょう。Ruby言語ではオブジェクト指向プログラミングをとても自然に行えますが、他のメジャーな言語においても、BioPerl, BioJava, BioPythonの全てがオブジェクト指向で書かれていることを考えると、現時点においては必須のスタイルといえます。

- クラス
  - 扱うオブジェクトに共通の特徴をくくり出す
  - → 抽象化：モノをどう捉えるか、理解するか
  - ・データ（オブジェクトの属性）のいれもの
  - ・データの操作方法（メソッド）を定義
- スーパークラス
  - 複数のクラスに共通の特徴をくくり出す
  - → 各サブクラスで継承し、差分プログラミング

化合物のクラスを作る

ここでは、化合物（compound）のクラスを作ってみることにします。化合物のデータを保持するいれもの、という意味でクラス名をCpdとしました。

```
class Cpd
end
```

このクラスは、とりあえず化合物の「名前」と「化学式」と「説明」を覚えておく箱とします。また、化学式から組成をもとめる機能、組成から分子量を計算する機能を作ってみようと思います。

まずは、名前、化学式、説明を格納して表示できるようにします。

```
class Cpd
  attr_accessor :name,
   :formula, :definition
end
```

アクセサ

インスタンス変数@nameに対して

```
def name=(arg)
  @name = arg
end

def name
  return @name
end
```

のように定義されたメソッドのこと。オブジェクトの内部状態（インスタンス変数）の値を参照したり書き換えたりするために使わます。

attr_accessorメソッドは、指定したシンボルと同名のインスタンス変数（:nameに対しては@name）に値を代入したり取り出したりするメソッド（アクセサ）を自動的に定義してくれます。また、attr_writerは代入する方のメソッドだけを定義し、attr_readerは取り出す方だけを定義します。

インスタンス変数

オブジェクトの属性（内部状態）を保持するための変数で、同じクラスのオブジェクトでも、インスタンス毎に内容が異なるようなデータです。Rubyでは@がついてる変数がインスタンス変数になります。

```
c1 = Cpd.new
```

Cpdクラスのあたらしいオブジェクトc1を作り、ATP分子のデータを格納してみます。

```
c1.name = "ATP"
c1.formula = "C10H16N5O13P3"
c1.definition =
  "Adenosine 5'-triphosphate"
```

覚えさせた値を表示するには以下のようにします。

```
puts c1.name       # => "ATP"
puts c1.formula    # => "C10.."
puts c1.definition # => "Ade.."
```

次は、Cpdクラスに化学式から組成を抽出するメソッドを追加します。C10H16N5O13P3という文字列から、各元素を表す文字と数字のペアをHashに格納し、

```
c1.composition     # => {
  "C" => 10, "H" => 16,
  "N" => 5,  "O" => 13,
  "P" => 3
}
```

のように"C"に対して10という値が得られるようにします。

```
def composition
  comp = Hash.new
  re = /([A-z]+)(\d+)/
  @formula.scan(re) do |e,n|
    comp[e] = n.to_i
  end
  return comp
end
```

ここで、scanは文字列中に含まれるパターンを正規表現ですべて順番に探すメソッドで、アルファベットの繰り返しと数字のペアに対してマッチさせています。

最後に、組成から全体の分子量を計算するメソッドを作ってみます。各元素の分子量

が必要になりますが、ここでは必要な元素の分だけ埋め込んでしまいます。本当は別のデータファイルから読み込むか、データベースから取得するのがよいでしょう。

```
class Cpd
  attr_accessor :name,
    :formula, :definition

  # 使用する分子量の表
  MW = {
    "C" => 12.011,
    "H" => 1.00794,
    "N" => 14.00674,
    "O" => 15.994,
    "P" => 30.973762,
  }

  # 組成から分子量の合計を計算
  def weight
    total = 0.0
    composition.each do |e,n|
      total += MW[e] * n
    end
    return total
  end

  private

  # 化学式 @formula から組成を抽出
  def composition
    comp = Hash.new
    re = /([A-z]+)(\d+)/
    @formula.scan(re) do |e,n|
      comp[e] = n.to_i
    end
    return comp
  end
end
```

ライブラリ化

作成したクラスの定義をcpd.rbという別のファイルに括り出すことでライブラリ化すると、様々なプログラムから再利用できるようになります。ライブラリは require 'cpd' で読み込むことができます。

このライブラリを利用して、いくつかの化合物を登録し、分子量を計算するプログラムcpd_test.rbを書いてみましょう。

```
#!/usr/bin/env ruby

require 'cpd'

c1 = Cpd.new
c1.name = "ATP"
c1.formula = "C10H16N5O13P3"
c1.definition = "cpd:C00002"

c2 = Cpd.new
c2.name = "Caffeine"
c2.formula = "C8H10N4O2"
c2.definition = "cpd:C07481"

c3 = Cpd.new
c3.name = "Tamiflu"
c3.formula = "C16H28N2O4"
c3.definition = "dr:D00900"

list = [ c1, c2, c3 ]

list.each do |c|
  puts ">>> Compound"
  puts "Name: #{c.name}"
  puts "Desc: #{c.definition}"
  puts "Formula: #{c.formula}"
  puts "Weight: #{c.weight}"
end
```

このプログラムの実行結果は以下のようになります。

```
% ruby cpd_test.rb
>>> Compound
Name: ATP
Desc: cpd:C00002
Formula: C10H16N5O13P3
Weight: 507.114026
>>> Compound
Name: Caffeine
Desc: cpd:C07481
Formula: C8H10N4O2
Weight: 194.18236
>>> Compound
Name: Tamiflu
Desc: dr:D00900
Formula: C16H28N2O4
Weight: 312.3878
```

ワークブック
ライブラリの活用

By Toshiaki Katayama

分子量電卓

化学式を入力すると分子量を計算して表示するようなプログラムmol_weight.rbを作成してみましょう。

計算部分は cpd.rbライブラリに任せれば良いので、少ない行数で書くことができるはずですね。

```
#!/usr/bin/env ruby

require 'cpd'
```

無限の繰り返しは loop 文などで書くことができます。

```
loop do
  print "Input formula: "
  formula = gets
  puts "Formula: #{formula}"
end
```

空っぽのあたらしいCpdオブジェクトを作り、きれいにした化学式を代入します。

```
  cpd = Cpd.new
  cpd.formula = formula
```

最期に、分子量を計算して表示します。

```
  puts "Weight: #{cpd.weight}"
```

以下のようなプログラムができました。

```
#!/usr/bin/env ruby

require 'cpd'

loop do
  print "Input formula: "
  formula = gets
  cpd = Cpd.new
  cpd.formula = formula
  puts "Weight: #{cpd.weight}"
end
```

このプログラムに水分子H_2OやATPの化学式を入力した場合の実行結果は以下のようになります。

```
% ruby mol_weight.rb
Input formula: H2O1
Weight: 18.00988
Input formula: C21H28N7O14P2
Weight: 664.364024
Input formula: ^C
```

入力を続ける限り計算してくれるため、終了する場合は Control + C キーを押します。

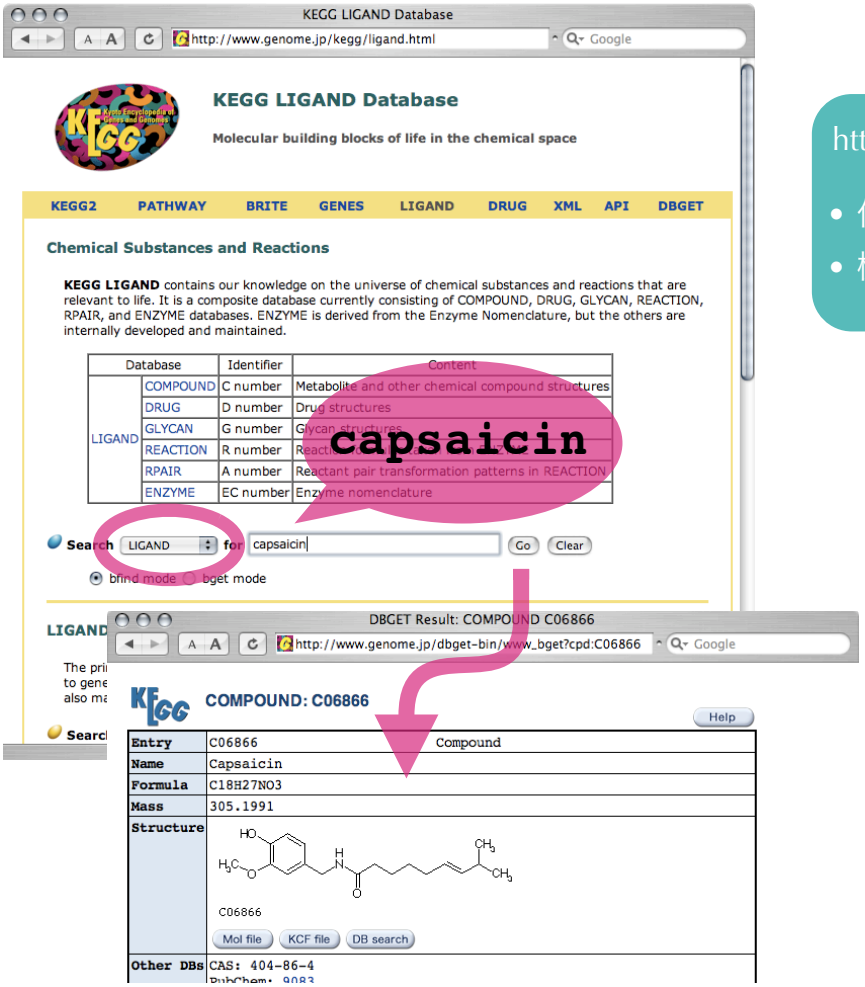
KEGG COMPOUNDの利用

応用として、化合物のデータを LIGAND の COMPOUND データベースから読み込むようにしてみてください。なお、次回以降で紹介する BioRuby や ChemRuby などのライブラリを利用するとエントリの取得やパーズを簡単に行うことができます。

```
#!/usr/bin/env ruby

require 'cpd'
require 'bio'  # BioRubyの読み込み

serv = Bio::KEGG::API.new

loop do
  print "KEGG ID (cpd:C00002): "
  entry_id = gets
  entry = serv.bget(entry_id)
  kegg_cpd = Bio::KEGG::COMPOUND.new(entry)
  cpd = Cpd.new
  cpd.formula = kegg_cpd.formula
  puts "Formula: #{cpd.formula}"
  puts "Original mass: #{kegg_cpd.mass}"
  puts "Calculated weight: #{cpd.weight}"
end
```

注：実際には、H2O → H2O1 など、1が省略されていても機能するように cpd.rb を拡張する必要があります。

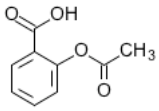
```
re = /([A-Z][a-z]*)(\d*)/
@formula.scan(re) do |e,n|
  comp[e] = n.to_i || 1
end
```



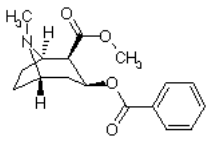
http://www.genome.jp/kegg/ligand.html

- 他の分子も試してみよう。
- 機能は？

dr:D00109
Aspirin (JP14/USP);
Easprin (TN)
C9H8O4

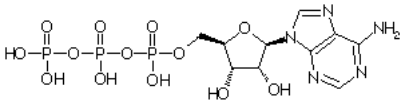


dr:D00110
Cocaine (USP);
Cocaine (TN)
C17H21NO4

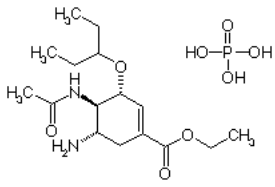


cpd:C00001
H2O; Water
H2O

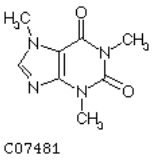
cpd:C00002
ATP; Adenosine 5'-triphosphate
C10H16N5O13P3



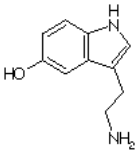
dr:D00900
Oseltamivir phosphate (JAN/USAN);
Tamiflu (TN)
C16H28N2O4.H3O4P



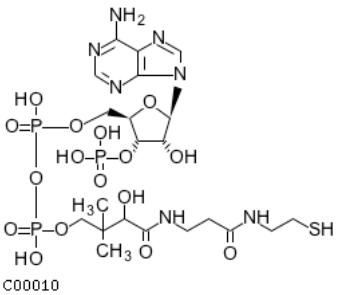
cpd:C07481
Caffeine
C8H10N4O2



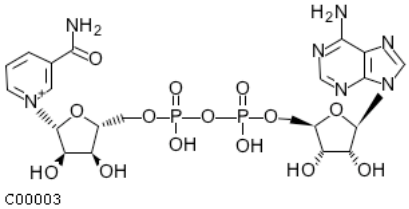
cpd:C00780
Serotonin; 5-Hydroxytryptamine
C10H12N2O



cpd:C00010
CoA; Coenzyme A; CoA-SH
C21H36N7O16P3S

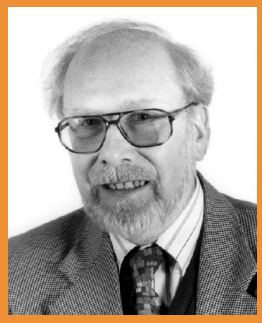


cpd:C00003
NAD+; NAD; Nicotinamide adenine dinucleotide;
DPN; Diphosphopyridine nucleotide; Nadide
C21H28N7O14P2

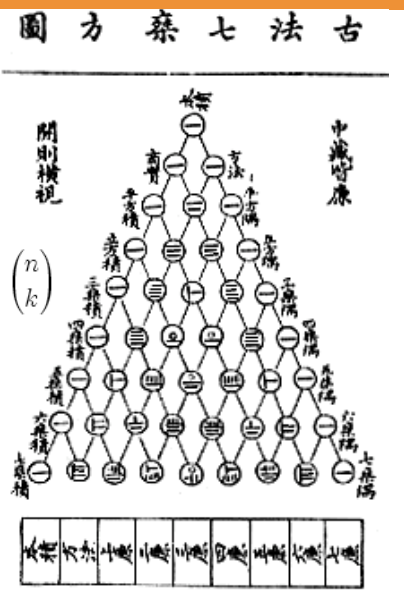


「プログラミングは、構築的（constructive）な活動である」

プログラミング言語 PASCAL の作者
Niklaus Wirth



$$\binom{n-1}{k} + \binom{n-1}{k-1} = \binom{n}{k}$$



第４回 データリソース活用 III

BioRuby とウェブサービス

By Masumi Itoh, Toshiaki Katayama

コンテンツ

1. BioRuby
   1.1. イントロダクション
   1.2. インストール
   1.3. BioRuby シェル
   1.4. フラットファイル
   1.5. BLAST の処理
2. ウェブサービス
   2.1. SOAP/WSDL
   2.2. KEGG API
   2.3. PDB, PSORT, EXPRESSION

第４回目は、BioRubyの紹介とウェブサービスについて解説します。実習としてBioRuby を利用したフラットファイルからのデータ抽出、BLAST の結果の処理等を行ってみます。さらに、KEGG API を用いてフィロジェネティックプロファイルの作成を行ってみます。応用として、パスウェイ上の遺伝子について、PDB の立体構造へリンクがたどれるかどうか、PSORT で予測したタンパク質の細胞内局在、マイクロアレイによる遺伝子発現などによって色付けしてみましょう。

BioRubyとオープンバイオ

Open Bio* プロジェクト

By Toshiaki Katayama

イントロダクション

BioRuby は Ruby 言語用のバイオインフォマティクスライブラリです。Ruby 言語を利用して塩基配列やアミノ酸配列を扱ったり、各種のデータベースやアプリケーションを簡単に利用することができます。

BioRuby の開発は 2000年11月にはじまりました。国内の学会や BOSC での発表などを経て、内外の開発者の参加によって機能追加と改良が進みました。その後、2002, 2003年の BioHackathon に招待され BioPerl, BioPython, BioJava など他の Open Bio* プロジェクトとの連携も進みました。

Open Bio Foundation (O|B|F) はこれらのプロジェクトの推進を目的とした非営利団体で、BioRuby も O|B|F の傘下で開発サーバとメーリングリストなどの運用支援を受けています。また BOSC の開催も O|B|F のミッションの１つで、毎年 ISMB 学会の SIG として世界中の開発者が一同に会し成果報告と活発な議論を行っています。

一方で、国内では Open Bio* に関する情報は少なく、人的交流も限られていました。そこで、BioPerl や EMBOSS など海外のリソースを有効に活用するとともに、BioRuby や G-language, KNOB など国内のプロジェクトの推進と技術的な情報交換をはかるという目的で、2004年に有志によって「オープンバイオ研究会」が設立されました。

2005年、BioRuby は ChemRuby とともに情報処理推進機構 IPA の未踏ソフトウェア創造事業に採択され、様々な新機能を追加したバージョン 1.0 がリリースされました。その後も継続して開発が行われており、両者を組み合わせることでポストゲノム解析が容易に行えるような環境を目指しています。

インストール

BioRuby のインストール方法はいくつかありますが、通常の方法は付属の README ファイルに書かれていますので、ここではホームディレクトリにインストールする方法を紹介します（図）。オプション --prefix を指定する部分と、使用する際に環境変数 RUBYLIB と PATH の設定が必要になる点が通常と異なります。環境変数の設定はシェルの設定ファイルに書いておくとよいでしょう。

BioRubyのサイト

```
http://bioruby.org/
```

BioRubyのホームディレクトリへのインストール方法

```
% wget http://bioruby.org/archive/bioruby-1.0.0.tar.gz
% tar xvfz bioruby-1.0.0.tar.gz
% cd bioruby-1.0.0
% ruby install.rb config --prefix=$HOME
% ruby install.rb setup
% ruby install.rb install
% export RUBYLIB="$HOME/lib/ruby/site_ruby/1.8"
% export PATH="$PATH:$HOME/bin"
% ruby -r bio -e 'p Bio::BIORUBY_VERSION'
[1, 0, 0]
```

ISMB
International Conference on Intelligent Systems for Molecular Biology

バイオインフォマティクス分野で最も規模の大きな国際学会の１つ

SIG
Special Interest Group

興味の対象を絞って併催されるミーティング

BOSC
Bioinformatics Open Source Conference

バイオインフォマティクスに関するオープンソースのソフトウェアについて発表を行うカンファレンス

O|B|F
Open Bioinformatics Foundation
http://open-bio.org/

バイオインフォマティクスに関するオープンソースのプロジェクトを支援する非営利団体

オープンバイオ研究会

http://open-bio.jp/

日本における O|B|F の役割を果たし、BOSC に相当するミーティングを国内で開催することを目的とした団体

BioRubyシェル

プログラミングの前に

By Toshiaki Katayama

BioRubyをさわってみよう

BioRuby ライブラリを使ったプログラムの作成をはじめる前に、BioRuby ではどんなことができるのか、BioRuby を内蔵した irb である「BioRubyシェル」を使って試してみましょう。新規ディレクトリ名として "testproj" を指定し bioruby コマンドを実行します。

```
% bioruby testproj

. . .  BioRuby  in  the  shell  . . .

 Version : BioRuby 1.0.0 / Ruby 1.8.4

Creating directory (session/) ... done
Creating directory (data/) ... done
Creating directory (plugin/) ... done
bioruby>
```

まずは適当な塩基配列を作ってみましょう。

```
bioruby> dna = seq("atgcatgcaaaa")
```

seq コマンドで文字列から塩基配列やアミノ酸配列を作ることができます。変数の中身を確認するには puts や print, pp, p などを用います。

```
bioruby> puts dna
atgcatgcaaaa
bioruby> p dna
"atgcatgcaaaa"
```

もしくは、設定で echo を有効にすると irb と同様に評価した値を画面に表示するようになります。

```
bioruby> config :echo
Echo on
  ==> nil
bioruby> dna
  ==> "atgcatgcaaaa"
bioruby> dna.complement
  ==> "ttttgcatgcat"
bioruby> puts dna.gc_percent
  ==> 33
bioruby> puts dna.composition
  ==> {"a"=>6, "c"=>2, "g"=>2, "t"=>2}
bioruby> puts dna.length
  ==> 12
```

では、アミノ酸配列に翻訳してみましょう。

```
bioruby> pep = dna.translate
  ==> "MHAK"
```

これらの配列オブジェクトはそれぞれどのようなクラスのインスタンスなのでしょうか。

```
bioruby> dna.class
  ==> Bio::Sequence::NA
bioruby> pep.class
  ==> Bio::Sequence::AA
```

Bio::Sequence::NA と AA クラスであることがわかりました。クラスが継承しているクラスやモジュールは ancestors メソッドで調べることができます。

print と puts と p と pp

print は明示的に \n を入れないと改行しませんが、puts は必ず改行してくれます。このため、あえて改行したくない場合以外は puts の方が便利に感じられます。

```
print "Hello\n"
puts "Hello"
```

一方で、print や puts の代わりに、p や pp を使うと 、Hash や Array など複雑なオブジェクトの中身を人間が見やすいように整形して表示してくれるので、値の確認やデバッグ用途に便利です。p よりも pp の方がより見やすい表示になる（pp は pretty print の略）ため、BioRuby シェルでは最初から pp も使えるようになっています。

```
ary = [1, 2, 3]
hash = {"a" => 1, "b" => 2, "c" => ary}
print ary, hash
puts ary, hash
pp ary, hash
p ary, hash
```

```
bioruby> Bio::Sequence::NA.ancestors
  ==> [Bio::Sequence::NA, Bio::Sequenc
  e::Common, String, Enumerable, ...]
bioruby> Bio::Sequence::AA.ancestors
 ==> [Bio::Sequence::NA, Bio::Sequenc
 e::Common, String, Enumerable, ...]
```

どちらも Bio::Sequence::Common を include しており、Ruby の String を継承しています。そのため、NA と AA は多くのメソッドを共通で持っていますが、差分プログラミングにより NA にだけ translate があるなど、多少の違いがあります。

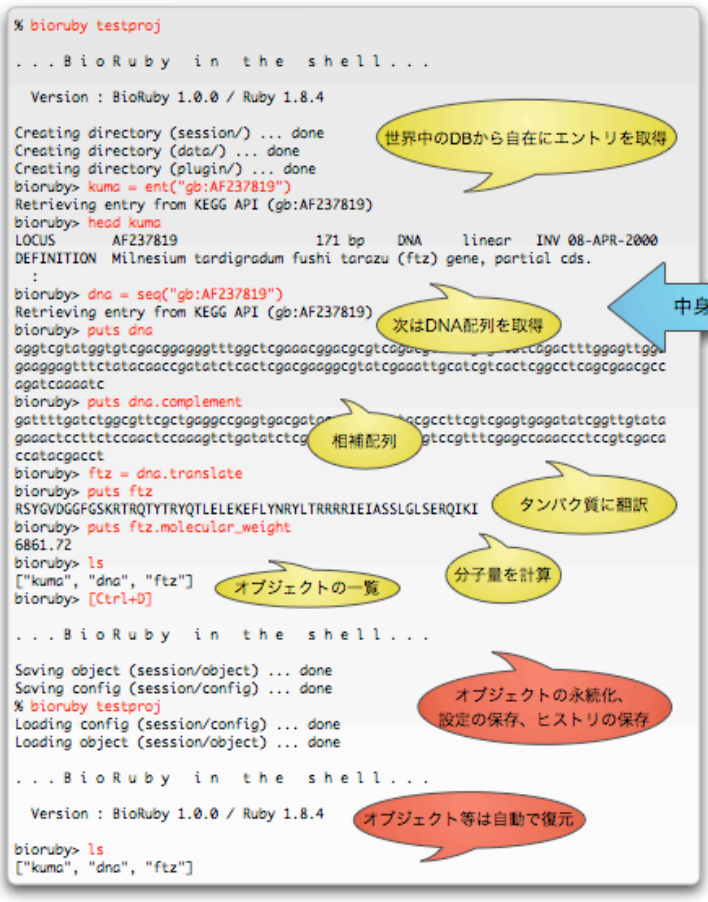
```
bioruby> dna.methods
bioruby> dna.methods - pep.methods
bioruby> pep.methods - dna.methods
```

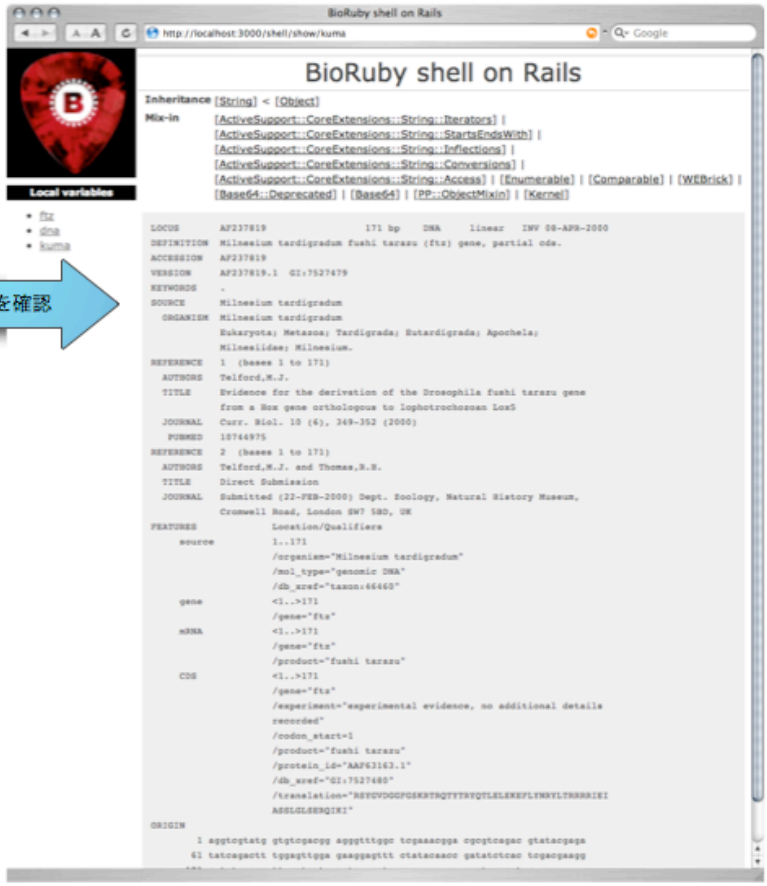
アミノ酸配列でもいくつかメソッドを試してみましょう。

```
bioruby> pep.codes
  ==> ["Met", "His", "Ala", "Lys"]
bioruby> pep.molecular_weight
  ==> 485.605
bioruby> pep * 3
  ==> "MHAKMHAKMHAK"
bioruby> pep + pep.reverse
  ==> "MHAKKAHM"
bioruby> pep[/H.K/]
  ==> "HAK"
```

配列はデータベースから取得することもできますが、その前に他の BioRuby シェルのコマンドをいくつか試してみましょう。

```
bioruby> config :echo
Echo off
bioruby> ls
["_", "dna", "pep"]
bioruby> dir
   UGO  Date                            Byte  File
------  ------------------------------  ----------  ----------
 40755  Fri Jun 02 11:10:04 JST 2006          68  "data"
 40755  Fri Jun 02 11:10:04 JST 2006          68  "plugin"
 40755  Fri Jun 02 11:10:04 JST 2006         102  "session"
bioruby> nucleicacids
a      a        Adenine
t      t        Thymine
  :
n      [atgc]
bioruby> aminoacids
?      Pyl      pyrrolysine
A      Ala      alanine
  :
Z      Glx      glutamine/glutamic acid
bioruby> codontables
1        Standard (Eukaryote)
2        Vertebrate Mitochondrial
  :
bioruby> codontable(2)
  :
bioruby> config :color
bioruby> codontable(2)
  :
bioruby> config :echo
Echo on
  ==> nil
```





BioRuby シェルのコマンド

BioRuby シェルでは open-uri, pp, yaml などの Ruby の便利な標準添付ライブラリが最初から読み込まれていますし、利用できる場合には irb の補完機能も有効になっています。また、コマンドのヒストリのタイムスタンプ付き自動保存や、作成したオブジェクトのセッションをまたいだ永続化、コマンド操作を記録したスクリプトの自動生成など irb にない機能も追加されています。利用できるコマンドを表にまとめておきます。

コマンド	機能概要
seq	塩基配列・アミノ酸配列を生成、またはデータベースから取得 •**配列取得方法** •文字列リテラル •データベースファイル名または IO オブジェクト •エントリ ID (含 EMBOSS USA) •**配列の主なメソッド** complement, translate, molecular_weight, composition, subseq, window_search, splicing, randomize, gc_percent これらの他、Ruby の強力な文字列 (String) クラスのメソッドが全て適用可能
ent	データベースエントリを様々な方法で取得 •**エントリ取得方法** •データベースファイル名または IO オブジェクト •エントリ ID •**flatindex** コマンドで作成した BioFlat データベース •OBDA (Open Bio Database Access) データベース •EMBOSS USA (Uniform Sequence Access) データベース •KEGG API の DBGET 対応データベース
obj	パース済みのエントリをオブジェクトとして取得 **ent** コマンドで取得したものを **flatparse** コマンドでパースしたものに相当
seqstat	塩基配列・アミノ酸配列の様々な統計情報をサマライズ（カラー対応）
aminoacids	アミノ酸の一覧と略称・正式名称などの情報
nucleicacids	塩基の一覧と略称・正式名称などの情報
codontables	コドンテーブルの番号一覧と定義を表示
codontable	コドンテーブルの表示と翻訳に利用するコドン表オブジェクトの取得（カラー対応）
flatparse	各種データベースエントリを自動判別してパース
flatfile	各種データベースファイルからエントリを読み出す
flatauto	各種データベースファイルからパースされたオブジェクトを取得
flatindex	各種データベースファイルを検索用にインデックス化
flatsearch	**flatindex** コマンドで作成した BioFlat データベースを検索
flatfasta	各種データベースファイルを FASTA 形式のファイルに変換

コマンド	機能概要
keggapi	KEGG API のメソッドを呼び出すためのサーバオブジェクト •**KEGG API の主なメソッド** (list_methods) **http://www.genome.jp/kegg/soap/doc/keggapi_manual_ja.html** •color_pathway_by_{elements,objects}, mark_pathway_by_objects •get_{elements,genes,enzymes,compounds,glycans,reactions,kos}_by_pathway •get_pathways_by_{genes,enzymes,compounds,glycans,reactions,kos} •get_genes_by_enzyme, get_enzymes_by_gene •get_{enzymes,compounds,glycans,reactions}_by_{enzyme,compound,glycan,reaction} •get_linkdb_by_entry, get_linked_pathway •get_motifs_by_gene, get_genes_by_motifs •get_best_best_neighbors_by_gene, get_best_neighbors_by_gene, get_reverse_best_neighbors_by_gene •get_paralogs_by_gene, get_ko_by_{gene,ko_class}, get_genes_by_{ko,ko_class} •get_oc_members_by_gene, get_pc_members_by_gene •get_genes_by_organism, get_number_of_genes_by_organism •convert_mol_to_kcf
keggdbs	KEGG API で利用できるデータベースの一覧
keggorgs	KEGG API で利用できる生物種の一覧
keggpathways	KEGG API で利用できる KEGG PATHWAY の一覧
binfo	DBGET で利用できるデータベースの情報
bfind	DBGET で利用できる各種データベースを検索
bget	DBGET のデータベースからエントリを取得
bconv	海外のデータベースと KEGG のエントリ ID の対応をとる
obdadbs	BioPerl などと共通の OBDA データベースの一覧
biofetch	BioFetch プロトコルによる OBDA データベースのエントリ取得
demo	BioRuby シェルのデモを実行
スクリプト生成	**script** 操作を記録し実行可能なプログラムとして保存
ファイル操作	**cd, pwd, dir, head, disp** ファイルやオブジェクトの確認
オブジェクト操作	**ls, rm, savefile** オブジェクトの一覧、削除、ファイル保存
各種設定変更	**config (:echo, :color, :message, :splash), pager**
ユーティリティ	**fold, fill, htmlseq, doublehelix, midifile**
Ruby on Rails	**web** コマンドによりシェルブラウザのサーバを起動

GenBankのパース

フラットファイルの例として GenBank 形式のエントリをパースして FASTA 形式に変換して出力してみましょう。ここではファイルサイズの小さい phage のディビジョンを使用します（ファイルは data ディレクトリに置くことにします）。

```
% cd testproj/data/
% wget ftp://ftp.genome.jp/pub/db/ncbi
/genbank/gbphg.seq.gz
% gunzip gbphg.seq.gz
% less gbphg.seq
```

BioRuby シェルには、配列データベースを FASTA 形式に変換するための **flatfasta** コマンドがあります。

```
bioruby> fasta = "data/gbphg.nuc"
bioruby> gbphg = "data/gbphg.seq"
bioruby> flatfasta(fasta, gbphg)
Saving fasta file (data/gbphg.nuc) ...
  converting -- data/gbphg.seq
done
```

塩基配列のファイル gbphg.nuc が生成されました。BioRuby シェルの中でもファイルの中身を確認することができます。あらかじめ **pager** コマンドで使用するページャーを設定しておきましょう。引数を省略すると環境変数 PAGER の値が使われます。

```
bioruby> pager "less"
Pager is set to 'less'
bioruby> disp fasta
  :
bioruby> disp gbphg
  :
```

今度はアミノ酸配列のファイルも作成してみましょう。この場合、GenBank のエントリをパースして、/translation="" に書かれている翻訳配列や、/protein_id="" などの情報を抽出する必要があります。

```
bioruby> flatauto(gbphg) do |entry|
  puts entry.entry_id
  entry.features.each do |feature|
    qua = feature.to_hash
    if qua["translation"]
      desc = [ entry.entry_id,
               qua["protein_id"],
               qua["product"]
             ].join(" ")
      pep = qua["translation"].first
      puts pep.to_fasta(desc, 60)
    end
  end
end
```

これを BioRuby シェルの中で実行するのではなく再利用できるスクリプトにしてみましょう。BioRuby を require して、Bio::FlatFile を使って GenBank のフラットファイルを読み込みます（フォーマットは自動判別）。あとはシェルで試したのと同じです。

```
#!/usr/bin/env ruby

require 'bio'

# 変換対象のファイル名
file = ARGV.shift  # gbphg.seq

Bio::FlatFile.auto(file) do |ff|
  # 各エントリごとに繰り返し
  ff.each do |entry|
    # 各 FEATURE ごとに繰り返し
    entry.features do |feature|
      # Feature Qualifier の Hash に変換
      qua = feature.to_hash
      # /translation="" 行があれば
      if qua["translation"]
        # コメント行を作って
        desc = [ entry.entry_id,
                 qua["protein_id"],
                 qua["product"]
               ].join(" ")
        # 翻訳配列部分を取り出し
        pep = qua["translation"].shift
        # FASTA フォーマットに変換して出力
        puts pep.to_fasta(desc, 60)
      end
    end
  end
end
```

BioFlat データベースの作成

gbphg.seq ファイルを毎回読み込んでいては処理に時間がかかるので、インデックスを作成して欲しいエントリを検索できるようにしてみましょう。

```
bioruby> flatindex("mydb", gbphg)
Creating BioFlat index (bioflat/mydb)
 ... done
bioruby> flatsearch("mydb","AB037107")
  :
```

RefSeq から GENES を作成

RefSeq も GenBank と同じフォーマットなので、同じやり方でパースできます。RefSeq のエントリを読み込んで KEGG GENES に入っていない生物種の GENES エントリを生成するプログラムを書いてみましょう。

ワークブック

データベースエントリの統計

By Toshiaki Katayama

データベースエントリの統計

課題：KEGG GENES の最新バージョンを読み込み、その中のエントリ数、全塩基数をカウントせよ。また、塩基数が最大のエントリは何か？

KEGG GENES のデータは、

```
ftp://ftp.genome.jp/pub/kegg/genomes/
```

以下にあります。ここでは大腸菌(eco)のデータを利用することにします。 まず対象となる GENES のファイルを観察してみましょう。

```
% wget ftp://ftp.genome.jp/pub/kegg/
genomes/eco/E.coli.ent
```

まずは、ベースとなるプログラムを考えましょう。記憶しておく項目は

- エントリー数 (entnum)
- 全塩基数 (seqlen)
- 配列が最長のエントリ名 (maxent)

の３つなので、それぞれ変数を用意します。

```
#!/usr/bin/env ruby

require 'bio'

entnum = 0
seqnum = 0
entmax = ""

Bio::FlatFile.auto(ARGF) do |ff|
  ff.each do |entry|
    # do something
  end
end

puts "No. of entries: #{entnum}"
puts "No. of bases:   #{seqnum}"
puts "Longest entry:  #{entmax}"
```

エントリー数と塩基数のカウント

ベースのプログラムで do something となっている部分にカウントするルーチンを追加しましょう。ここで、entry には KEGG GENES の１エントリ（１遺伝子）分のデータが入っています。フラットファイルから１エントリ読み込む毎に entnum の値を１増やせばエントリ数をカウントできますね。BioRuby で GENES のエントリから塩基配列の部分だけを取り出すには ntseq メソッドを使います。塩基配列オブジェクト (Bio::Sequence::NA) が得られますので length メソッドで長さを調べることができます (BioRuby にはこのための ntlen メソッドもあります)。この値を seqnum に足し込めば合計が求められますね。

```
ff.each do |entry|
  [        ] += 1
  [        ] += entry.ntseq.[        ]
end
```

最長の配列を持つエントリ名の出力

次に、最長エントリの探し方を考えましょう。いま entry に取り出されているエントリの塩基数が「これまでの最長」より長いかどうかを比べれば良さそうですね。そのためには「これまでの最長」を覚えておくための変数 seqmax を準備しておく必要があります。

先ほどは、塩基配列の長さを全塩基数をカウントする seqnum に直接足し込んでしまいましたが、最長との比較や、最長より長かった場合の上書きにも使いますので、いったん seqlen に取り出しておきましょう。

```
ff.each do |entry|
  # :
  seqlen = entry.ntseq.length
  seqnum += seqlen
  # :
end
```

取り出した長さを「これまでの最長」である seqmaxの値と比較します。最長よりも長かった場合はどうしたらよいでしょうか？

```
if seqlen > seqmax
  seqmax = [        ]
end
```

BioRuby ではエントリ名を entry_id メソッドで取り出すことができます。これらのことをあわせると、最終的なプログラムは以下のようになるでしょう。

```
#!/usr/bin/env ruby

require 'bio'

entnum = 0
seqnum = 0
entmax = ""
seqmax = 0

Bio::FlatFile.auto(ARGF) do |ff|
  ff.each do |entry|
    entnum += 1
    seqlen = entry.ntlen
    seqnum += seqlen
    if seqlen > seqmax
      seqmax = seqlen
      entmax = entry.entry_id
    end
  end
end

puts "No. of entries: #{entnum}"
puts "No. of bases:   #{seqnum}"
puts "Max entry:      #{entmax}"
puts "Max length:     #{seqmax}"
```

これだけではおもしろくないので、練習にコドン頻度でも数えてみましょうか。

Hashによる数え上げ

ファイル中の同じ行の数を数える、といった簡単な数え上げであれば、コマンドラインで

```
% sort file.txt | uniq -c
```

などとすれば済みますが、もう少し複雑なデータで同じものが何個ずつあるかを数えるのに Perl や Ruby では高速な Hash をよく使います。

たとえば、テキストファイル中に現れる単語の頻度を数える場合には、初期値が 0 の Hash を用意しておき、出現した word ごとにカウントを 1 増加すればよいでしょう。 この方法で、何種類出現するか分からないデータも容易に数えることができます。 結果は頻度順（必要に応じて逆順）に並べて表示してみます。

```
#!/usr/bin/env ruby

# 頻度を数えておく初期値 0 の Hash
count = Hash.new(0)

# ファイルから１行ずつ読み込む
ARGF.each do |line|
  # スペースで行を単語に分割
  words = line.chomp.split(/\s+/)
  # 単語毎にカウントを増加
  words.each do |word|
    count[word] += 1
  end
end

# 結果を頻度でソートし直す
rank = count.sort_by{|x| x.last}
# 逆順にして単語と頻度を表示
rank.reverse.each do |word, num|
  puts "#{num}\t#{word}"
end
```

コドン頻度を数える場合も全64種類分の変数を用意するのではなく、Hashのキーにコドンを、値にカウント数を記録します。結果は64カラムのタブ区切りテキストにしてみます。

```
#!/usr/bin/env ruby

require 'bio'
bases = ["t", "c", "a", "g"]

Bio::FlatFile.auto(ARGF) do |ff|
  ff.each do |entry|
    # 塩基配列を取得
    s = entry.ntseq
    # ３文字（コドン）ごとにカウント
    count = Hash.new(0)
    s.window_search(3,3) do |cdn|
      
    end

    # エントリ名を表示
    print entry.entry_id
    # 4x4x4=64通りごとに値を表示
    bases.each do |i|
      bases.each do |j|
        bases.each do |k|
          cdn = "#{i}#{j}#{k}"
          print "\t#{count[cdn]}"
        end
      end
    end
    puts  # タブ切りをエントリ毎に改行
  end
end
```

相同性検索とモチーフ

BLASTとPROSITE

By Masumi Itoh, Toshiaki Katayama

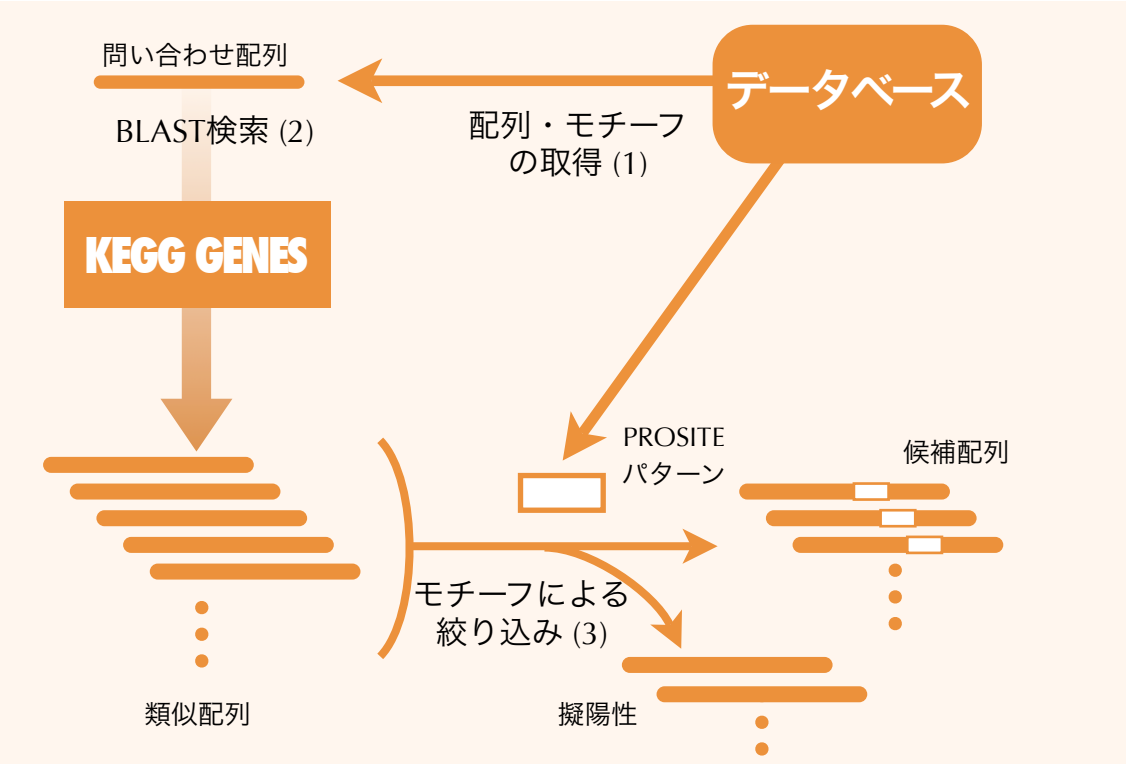
相同性検索とは

分子生物学の分野においてタンパク質・遺伝子の配列類似性（相同性 homology）はもっとも重要な指標の一つです。膨大なアミノ酸配列・塩基配列のデータベースの中から、問い合わせ配列に対して統計的に有意な類似配列を検索し、類似配列の情報をもとに遺伝子の機能推定や絞り込みを行う相同性検索の手法は、バイオインフォマティクスという言葉が使われる以前から広く用いられてきました。

この十数年間、配列データベースはコンピュータの計算能力の向上を上回るスピードで巨大化してきました。このため、既知の全ての配列に対し、厳密な最適解を求める手法での相同性検索は現実的に不可能となりました。そこで近似解として、統計的に有意な配列を高速に抽出する方法が研究されています。そのうち、もっとも広く用いられているアルゴリズムの１つが Altschul らによって開発された BLAST（Basic Local Alignment Search Tool）です。BLAST による相同性検索のサービスは米国 NCBI をはじめ世界中の様々な配列データベースで公開されています。また、このための BLAST プログラムもフリーソフトとして配布されています。京都大学の GenomeNet でも KEGG GENES など独自の配列データベースに対する BLAST 検索が提供されています。

モチーフによる検証

近年のゲノムプロジェクトの急速な進展により、上述の説明とは逆に、データベース中にも機能未知の配列が大量に蓄積されるようになってきました。そのため、既知のタンパク質を問い合わせ配列として、類似タンパク質の候補を検索し、配列群のファミリー解析を行うことがあります。しかし、配列の類似性だけでは目的の配列を分離できないことも多いため、特定のモチーフを指標に擬陽性の配列を排除することを考えてみましょう（下図）。



ここでは、問い合わせ配列とモチーフをデータベースから取得することにします。

(1) 問い合わせ配列・パターンをデータベースから取得する。
(2) BLAST検索を行い、データベースから類似配列の一覧を得る。
(3) 得られた配列にモチーフが存在するか否かを調べ、候補配列を絞り込む。

例として、ヒトのプリオン遺伝子と、前回も使用した PRION_2 のモチーフを用いることにします。相同性検索には、ゲノムの決まった全生物種の遺伝子が含まれる KEGG GENES データベースを利用します。

配列・モチーフの取得

後述の KEGG API を用いると GenomeNet でサービスしているデータベースから容易に配列やモチーフなどのエントリを取得することが出来ます。BioRuby で KEGG API を使用するためには、以下の 1 行が必要です。

```
keggapi = Bio::KEGG::API.new
```

エントリを取得するには KEGG API の bget メソッドを用います。bget は引数に "データベース名:エントリ名"（もしくは "生物種コード:遺伝子ID"）を与えることによりエントリを取得することが出来ます。 KEGG GENES から hsa:5621 遺伝子の配列を得るには

```
result = keggapi.bget("hsa:5621")
```

とします。得られた結果は result に格納しています。この中身は第２回で扱ったフラットファイル形式の文字列なので、BioRuby を用いてパースします。

```
entry = Bio::KEGG::GENES.new(result)
```

同様に PROSITE のエントリを bget で取得し、同様に BioRuby でパースします。

```
result = keggapi.bget("ps:PS00706")
motif = Bio::PROSITE.new(result)
```

BLAST検索

まず GENES のエントリからアミノ酸配列を抜き出し、FASTA 形式に変換します。

```
aaseq = entry.aaseq
fasta = aaseq.to_fasta(entry.entry_id)
```

GenomeNet の BLAST サービスで検索を行うため、サーバに接続します。

```
prog  = "blastp"
db    = "genes"
blast = Bio:Blast.remote(prog, db)
```

今回は query 配列がタンパク質で検索対象もタンパク質データベースなので (GENES はアミノ酸配列と塩基配列のどちらもエントリ内に持っています)、blastpプログラムを用います（GenomeNet の BLAST ではなく、手元のコンピュータで検索を実行する場合は remote の代わりに local メソッドを使います）。

この BLAST 検索サーバに先ほどの fuga を問い合わせ配列として相同性検索を実行します。

```
report = blast.query(fasta)
```

検索結果を report に格納しました。

モチーフによる絞り込み

第３回で説明したように、PROSITE のパターンがアミノ酸配列に存在するかどうかは正規表現を用いて調べることが出来ます。先ほど取得した PROSITE エントリからパターンを抽出して正規表現に変換します。

```
pa = motif.pattern
re = Bio::PROSITE.pa2re(pa)
```

相同性検索の結果、ヒットした類似配列のそれぞれについて、E-value が 10^{-4} 以下など有意な値を持つものに対し処理を繰り返すには以下のようにします。

```
report.each do |hit|
  if hit.evalue < 0.0001
    # それぞれのヒットに対して行う処理を書く
  end
end
```

ループの中で類似配列の ID を抽出し、上述の問い合わせ配列と同じようにアミノ酸配列を取得します。ここでは KEGG GENES に対して相同性検索をかけたので target_id は "種名:エントリ名" の形式をしています。

```
target_id = hit.target_id
target = keggapi.bget(target_id)
entry = Bio::KEGG::GENES.new(target)
aaseq = entry.aaseq
```

取得したアミノ酸配列がモチーフを持つかどうかを調べます。正規表現に変換したPROSITE パターンを用いて、正規表現によるマッチングを行いモチーフを持てばエントリ名を出力します。

```
if aaseq[re]
  puts target_id
end
```

以上をまとめ、問い合わせに用いるアミノ酸配列の ID と、絞り込みのための PROSITE の ID をコマンドライン引数で渡して検索を行なうプログラムは以下のようになりました。

```
#!/usr/bin/env ruby

require 'bio'

keggapi = Bio::KEGG::API.new

# 引数でアミノ酸配列とモチーフの ID を渡す
query_id = ARGV.shift || "hsa:5621"
motif_id = ARGV.shift || "ps:PS00706"

# 問い合わせアミノ酸配列を FASTA 形式で取得
query = "-f -n a #{query_id}"
fasta = keggapi.bget(query)

# モチーフを取得しパターンを正規表現に変換
entry = keggapi.bget(motif_id)
motif = Bio::PROSITE.new(entry)
pa = motif.pattern
re = Bio::PROSITE.pa2re(pa)

# BLAST 検索のコマンドと DB を指定
prog = "blastp"
db = "genes"
blast = Bio::Blast.remote(prog, db)

# 問い合わせ配列を与えて BLAST 検索を実行
result = blast.query(fasta)

# 結果からヒットした配列ごとに
result.each do |hit|
  tid = hit.target_id
  evalue = hit.evalue
  # E-value が十分に小さいものだけ
  if evalue < 0.0001
    # 配列からギャップ文字 - などを削除
    seq = hit.target_seq
    seq.gsub!(/[^A-Z]/, '')
    # 配列にモチーフがマッチするかどうか
    if seq[re]
      print "# motif+ "
    else
      print "# motif- "
    end
    puts [tid, evalue, seq].join("\t")
  end
end
```

プリオンでの結果

では、プリオンを例に実行してみます。

```
% ruby blast_ps.rb hsa:5621 ps:PS00706
```

結果は以下のようになりました。

```
# motif+ hsa:5621   2.75496e-83 MANLGCWML
# motif+ ptr:458076 2.33221e-82 MANLGCWML
# motif+ rno:24686  2.67459e-70 MANLGYWLL
# motif+ mmu:19122  2.67459e-70 MANLGYWLL
# motif+ bta:281427 5.04406e-69 SHIGSWILV
# motif+ cfa:485783 2.50915e-60 SHIGGWILL
# motif+ ssc:494014 4.27997e-60 SHIGGWILV
# motif+ gga:396452 3.80198e-08 HNQKPWKPP
# motif- xla:444620 8.46992e-08 NKQWKPPKS
```

問い合わせ配列自身 (hsa), チンパンジー (ptr), ラット (rno), マウス (mmu), ウシ (bta), イヌ (cfa), ブタ (ssc) のプリオンは高いスコアで保存されています。また、トリ (gga), カエル (xla) は同程度に低めのスコアですが、トリにはプリオンのモチーフが保存されているのに対し、カエルには無い（崩れている？）ということになりました。

相同性と局在

こんどは細胞内局在シグナルの有無で絞り込んでみます。例として、シロイヌナズナの uricase (ath:At2g26230) とマイクロボディ（つまりペルオキシソーム） への局在シグナルのモチーフ (ps:PS00342) で実行してみましょう。このモチーフはC末端にあるため、配列のアライメント領域のみではなく、全長の配列が必要です。そこでデータベースからアミノ酸配列の全長を取得するようにプログラムを書き換えます。先ほどのプログラムで

```
# 配列からギャップ文字 - などを削除
seq = hit.target_seq
seq.gsub!(/[^A-Z]/, '')
```

となっていた部分を、ヒットした遺伝子の ID tid を利用して以下のように変更します。

```
# ヒットした遺伝子の GENES エントリを取得
entry = keggapi.bget(tid)

# GENES のエントリからアミノ酸配列をとりだす
gene = Bio::KEGG::GENES.new(entry)
seq = gene.aaseq
```

修正したプログラムを実行してみます。

```
% ruby blast_ps.rb ath:At2g26230 ps:PS00342
```

結果は以下のようになりました。

```
# motif + ath:At2g26230       4.14877e-178 MAQEA
# motif + osa:P0423B08.30     5.33411e-117 ADRLE
# motif - xtr:496896          6.43389e-46  YGKNA
# motif - xla:399031          1.09746e-45  DTGYG
# motif + ssc:397510          6.02194e-44  YGKDM
# motif - rno:114768          7.8649e-44   YGKDM
# motif + cfa:490189          1.02719e-43  YGKDM
# motif - fra:Francci3_1910   2.28833e-43  QYGKA
# motif - mmu:22262           5.09788e-43  YGKDM
# motif - dre:436604          4.31561e-42  YGKNM
# motif + ddi:DDB0231470      7.36132e-42  IDNRY
# motif - aor:AO090011000588 7.61777e-39  RYGKD
# motif - afm:Afu2g10520      9.94912e-39  RYGKD
# motif - bta:514113          1.69706e-38  YGKDM
# motif - dge:Dgeo_2601       1.87632e-37  YGKAE
# motif - dme:CG7171-PA       1.87632e-37  DHGYG
# motif - ani:AN9470.2        2.45055e-37  RYGKD
# motif - dra:DR1160          1.75618e-35  ENNYG
# motif - sma:SAV2018         2.53593e-34  QYGKA
# motif - cne:CNI02420        9.63651e-34  RYGKD
# motif - cal:orf19.2114      2.37355e-32  YGKGN
# motif + ptr:469368          3.09996e-32  YGKDI
# motif - sco:SCO6211         2.45624e-29  QYGKA
# motif - nfa:nfa52420        2.45624e-29  DNRYG
# motif - spo:SPCC1223.09     5.47194e-29  YGKTL
# motif - dre:573479          2.63036e-23  SGLKD
# motif - bha:BH0759          1.03676e-11  LSSFT
# motif - bsu:BG13986         1.76845e-11  LPSFT
# motif - bcl:ABC3735         1.65522e-09  LSSFT
```

アクチノバクテリアの一種の Flankia sp. Ccl3 (fra) などバクテリア（もちろんペルオキシソームを持たない）のホモログが真核生物のホモログよりも高い相同性を示していますが、モチーフを持っていないことがわかります。PROSITE DOC によるとアスペルギルスの uricase にはシグナル配列が存在することになっています。しかし、麹菌 (aor) では検出されていますが、*A. fumigatus* (afm) には検出されませんでした。

また他の例として、小胞体 (ER) へのシグナル配列のモチーフ (PS00014) などもありますので試してみてください。これらの細胞局在シグナルの有無は、後述の PSORT を用いた細胞内局在予測と併せて用いることでさらに検証を行うことができます。

ファイルに保存したBLAST結果の処理

さまざまな値の取り出し方

By Masumi Itoh, Toshiaki Katayama

BLASTの結果の整理

これまでの例では BioRuby を用いて、プログラムから直接 GenomeNet のデータベース (KEGG GENES) に対して BLAST 検索をかけ、その結果をファイルに保存せずそのままパースし、必要な情報を抽出していました。しかし、BLAST の結果を目で見て解釈しながら情報を抽出したいときなどには、いったんファイルに出力されたテキストデータをパーズする必要があります。

次ページに示すように、 BLAST の標準の出力フォーマットは人間にとっての可読性を高めるため非常に複雑な形式をしています。これを自前でパーズするプログラムを書くのは大変ですが、BioRuby ではフラットファイルのデータベース同様に読み込み、データを取り出すのに便利なメソッドがあらかじめ用意されています。下のプログラムでは期待値を抽出していますが、その他様々な値を取り出すことができます(右表)。

```
#!/usr/bin/env ruby

require 'bio'

# 結果ファイルを読み込み
Bio::FlatFile.auto(ARGF) do |ff|
  # それぞれの結果について
  ff.each do |report|
    # それぞれのヒットについて
    report.each do |hit|
      # 期待値が 10-3 以下のものの
      if hit.evalue < 0.001
        # IDを表示
        puts hit.target_id
      end
    end
  end
end
```

メソッド名	取り出される情報
evalue	期待値
bit_score	bitスコア
query_seq	問い合わせ配列
midline	アライメントのmidline文字列
target_seq	ヒットした配列
identity	% identity
overlap	オーバーラップしている領域の長さ
query_id	問い合わせ配列の ID
query_def	問い合わせ配列のコメント
query_len	問い合わせ配列の長さ
target_id	ヒットした配列の ID
target_def	ヒットした配列のコメント
target_len	ヒットした配列の長さ
query_start	相同領域の問い合わせ配列での開始位置
query_end	相同領域の問い合わせ配列での終了位置
targt_start	相同領域のヒットした配列での開始位置
target_end	相同領域のヒットした配列での終了位置
lap_at	上記4つの位置の配列

BLASTの結果

ヘッダ

```
BLASTP 2.2.1 [Jul-12-2001]


Reference: Altschul, Stephen F., Thomas L. Madden, Alejandro A. Schaffer,
Jinghui Zhang, Zheng Zhang, Webb Miller, and David J. Lipman (1997),
"Gapped BLAST and PSI-BLAST: a new generation of protein database search
programs",  Nucleic Acids Res. 25:3389-3402.
```

問い合わせ配列の情報

```
Query= eco:b0002 thrA, Hs, thrD, thrA2, thrA1; bifunctional
aspartokinase I/homeserine dehydrogenase I [EC:2.7.2.4 1.1.1.13];
K00928 aspartate kinase (A)
         (820 letters)
```

query_id (eco:b0002) / query_def / query_len (820)

データベースの情報

```
Database: genes
           1,477,699 sequences; 535,924,808 total letters

Searching..................................................done
```

検索結果のランキング

```
                                                                 Score     E
Sequences producing significant alignments:                      (bits)  Value

eci:UTI89_C0002  thrA; aspartokinase I [EC:2.7.2.4 1.1.1.3] [KO:...  1567  0.0
ecj:JW0001  thrA; fused aspartokinase I and homoserine dehydroge...  1567  0.0
eco:b0002  thrA, Hs, thrD, thrA2, thrA1; bifunctional aspartokin...  1567  0.0
(省略)
```

各ヒットの情報

```
>xoo:XO01820  homoserine dehydrogenase
          Length = 224

 Score =  153 bits (386), Expect = 2e-35
 Identities = 92/224 (41%), Positives = 133/224 (59%), Gaps = 2/224 (0%)

Query: 592 LRYAAEKSRRKFLYDTNVGAGLPVIENLQNLLNAGDELMKFSGILSGSLSYIFGKLDEGM 651
           LR A    +      VGAGLPV+ +++ L+  GD +     G+LSGSL+++F + D
Sbjct: 2   LRAACHASGAHYGDSATVGAGLPVLSSVRALVAGGDHIHSIKGVLSGSLAWLFHRYDGSG 61

Query: 652 SFSEATTLAREMGYTEPDPRDDLSGMDVARKLLILARETGRELELADIEIEPVLPAEFNA 711
           +FS+     A   GYTEPDPR DLSG DV RKLLILAR  G +LE A + +E ++PA   A
Sbjct: 62  AFSDCVREAIAAGYTEPDPRIDLSGEDVRRKLLILARAAGWQLEAAQVHVESLVPAVV-A 120

Query: 712 EGDVAAFMANLSQLDDLFAARVAKARDEGKVLRYVGNIDEDGVCRVKIAEVDGNDPLFKV 771
           +  +A    A+L   LD +  AR  +AR  G+ LR+VG +D  G    V + E+  + PL
Sbjct: 121 KLPLAELDAHLRALDAVVGARWQQARAAGRCLRFVGRVDAHG-ASVGLRELAPDHPLAGG 179

Query: 772 KNGENALAFYSHYYQPLPLVLRGYGAGNDVTAAGVFADLLRTLS 815
               +N +A   S  Y+   PL+++G GAG +VTAA +   D+LR ++
Sbjct: 180 AGTDNRVAISSDRYRAQPLLIQGPGAGAEVTAAALLDDVLRIVA 223
```

target_id (xoo:XO01820) / target_def / target_len (224) / bit_score (153) / evalue (2e-35) / identity (41%) / query_start (592) / target_start (2) / query_seq / target_seq / midline / query_end (815) / target_end (223)

データベースの情報と検索結果のサマリー

```
(省略)

 Database: genes
   Posted date:  Jun 8, 2006 10:21 AM
  Number of letters in database: 535,924,808
  Number of sequences in database:  1,477,699

Lambda     K      H
   0.319    0.135    0.383

Gapped
Lambda     K      H
   0.267    0.0410    0.140


Matrix: BLOSUM62
Gap Penalties: Existence: 11, Extension: 1
Number of Hits to DB: 785,112,349
Number of Sequences: 1477699
Number of extensions: 32025520
Number of successful extensions: 87538
Number of sequences better than 10.0: 1069
Number of HSP's better than 10.0 without gapping: 920
Number of HSP's successfully gapped in prelim test: 149
Number of HSP's that attempted gapping in prelim test: 84469
Number of HSP's gapped (non-prelim): 1488
length of query: 820
length of database: 535,924,808
effective HSP length: 134
```

ウェブサービス

SOAP/WSDLとKEGG API

By Toshiaki Katayama

ウェブサービスとは

ウェブを利用したデータベース検索などのサービスは、ブラウザからのマウスやキーボード操作によって行うものが主流です。しかし、次第にウェブ上のサービスをプログラムから呼び出して使おうというニーズが高まりました。当初は HTML から CGI プログラムへの引数を解析したものが利用されていましたが、より明確なプログラム専用のインターフェイスが検討されてきました。これらを総称してウェブサービスとよび、その一つの方法が SOAP です。

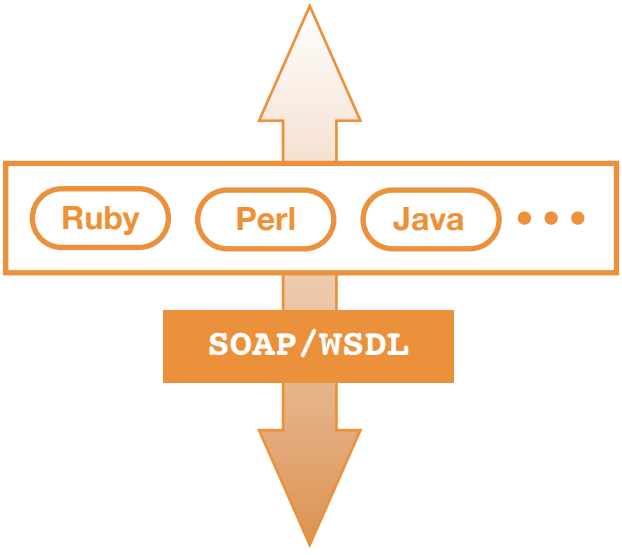
SOAP はサーバとのやり取りを XML のメッセージで行う規格で、通常はウェブと同様 HTTP で通信を行ないます。WSDL は、サーバで提供しているサービスを一覧にした XML ファイルで、SOAP サービスの利用を格段に容易にします。

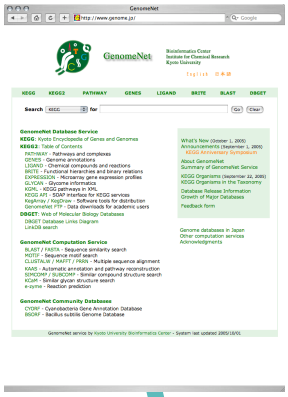
KEGG API は、この SOAP/WSDL を利用した KEGG のウェブサービスで、各種データベースの検索からパスウェイへのマッピングまで KEGG の様々な機能をプログラムから利用することが可能です。このため、全ての遺伝子やパスウェイについて同じ処理を繰り返すといったバッチ処理も簡単に実現できるようになりました。

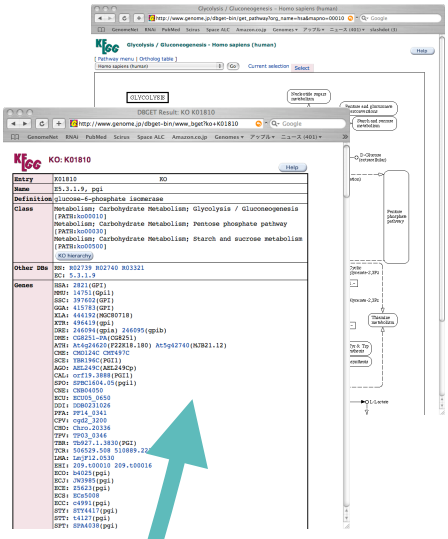
詳細は KEGG API のドキュメントを参照

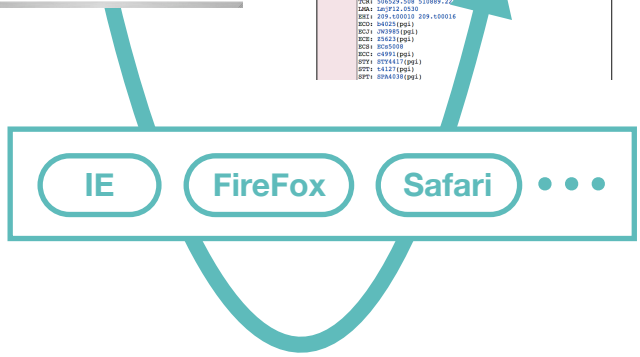
http://www.genome.jp/kegg/soap/

```
#!/usr/bin/env ruby
require 'bio'
serv = Bio::KEGG::API.new
genes = serv.get_genes_by_pathway("pa
genes.each do |gene|
  :
```









プログラムからの利用

ブラウザからの利用

KEGG API

パスウェイの種間比較

系統プロファイルの作成

By Toshiaki Katayama

系統プロファイルとは

生物種ごとに、それぞれの遺伝子をもつかどうかを表にしたものを系統プロファイルといいます（遺伝子に限らず様々な定義があります）。ここでは、指定したパスウェイに現れる酵素のリストから、生物種ごとにそれぞれの酵素に対応する遺伝子を持つかどうかを調べ、系統プロファイルを作成してみましょう。全生物種を見るのは大変なので、いくつか代表的な生物種を選ぶことにします。

```
真核生物   => [hsa, sce, ...]
光合成     => [ath, syn, ...]
バクテリア => [eco, bsu, ...]
古細菌     => [mja, sso, ...]
```

KEGG では、酵素番号にかわるものとして、パスウェイに載っている酵素のオーソログ遺伝子を KO (KEGG Orthology) とよばれるグループに分類しています。get_kos_by_pathway メソッドにより、指定したパスウェイにアサインされている KO のリストを得ることができます。さらに、各 KO に属する遺伝子は get_genes_by_ko メソッドで得ることができます。

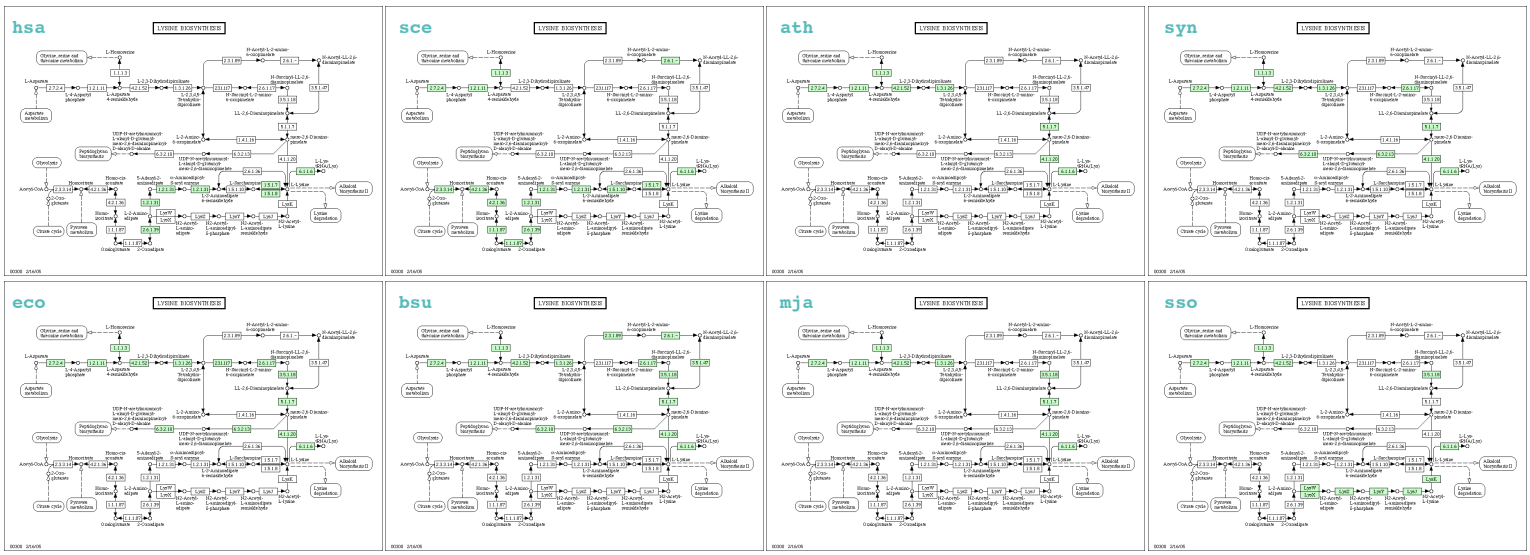
```
#!/usr/bin/env ruby

require 'bio'

path = ARGV.shift || "path:map00300"              # 引数でパスウェイを指定
serv = Bio::KEGG::API.new                         # KEGG API に接続
orgs = %w( hsa sce ath syn eco bsu mja sso )      # 注目する生物種のリスト
puts [ "KO number", *orgs ].join("\t")            # ヘッダ行の出力

list = serv.get_kos_by_pathway(path)              # パスウェイ上の KO のリストを取得
list.each do |ko|                                 # 各 KO 番号について ...
  profile = Hash.new                              # プロファイルを記憶する Hash を初期化
  genes = serv.get_genes_by_ko(ko, "all")         # KO に属する遺伝子のリストを取得
  genes.each do |gene|                            # それぞれの遺伝子について ...
    org = gene.entry_id[/^([a-z]{3}):/, 1]        # 由来する生物種３文字コード部分を抽出
    if orgs.include?(org)                         # 注目する生物種の遺伝子であれば
      profile[org] = true                         # 系統プロファイルにチェック
    end
  end
  next if profile.keys.empty?                     # 注目する生物種に遺伝子が無ければパス

  print ko                                        # そうでなければ KO 番号を表示し
  orgs.each do |org|                              # 各生物種ごとに ...
    if profile[org]
      print "\t+"                                 # 遺伝子があれば + フラグを表示
    else
      print "\t-"                                 # 遺伝子がなければ - フラグを表示
    end
  end
  puts
end
```

Lysine biosynthesis パスウェイの場合

アミノ酸のリシン合成経路で働く酵素を調べてみましょう。真核生物（ヒト hsa, 酵母 sce）と原核生物（大腸菌 eco, 枯草菌 bsu, 藍藻 syn）で使用されているパスウェイの経路が違うのが見て取れます。しかし、真核生物でも植物の ath は原核生物側しかなく、古細菌は euryarchaeota の mja は原核生物側、crenarchaeota の sso は真核生物側のパスウェイを利用しているようです。よく見るとヒトの経路は途切れていますね。実はリシンは必須アミノ酸です。

```
% ruby phylo_prof.rb path:map00300
```

KO number	hsa	sce	ath	syn	eco	bsu	mja	sso
ko:K00003	–	+	+	+	+	+	+	+
ko:K00133	–	+	+	+	+	+	+	+
ko:K00142	+	–	–	–	–	–	–	–
ko:K00143	–	+	–	–	–	–	–	–
ko:K00144	–	+	–	–	–	–	–	–
ko:K00215	–	–	+	+	+	+	+	–
ko:K00290	+	+	–	–	–	–	–	–
ko:K00291	+	–	–	–	–	–	–	–
ko:K00293	–	+	–	–	–	–	–	–
ko:K00674	–	–	–	–	+	–	–	–
ko:K00821	–	–	–	+	+	+	–	–
ko:K00825	+	–	–	–	–	–	–	–
ko:K00838	–	+	–	–	–	–	–	–
ko:K00841	–	+	–	–	–	+	–	–
ko:K00928	–	+	+	+	+	+	+	+
ko:K01439	–	–	–	–	+	+	+	+
ko:K01586	–	–	+	+	+	+	+	–
ko:K01655	–	+	–	–	–	–	–	–
ko:K01705	–	+	–	–	–	–	–	–
ko:K01714	–	–	+	+	+	+	+	+
ko:K01778	–	–	+	+	+	+	+	–
ko:K01871	–	+	–	–	–	–	–	–
ko:K01928	–	–	–	+	+	+	–	–
ko:K01929	–	–	–	+	+	+	–	–
ko:K04566	–	–	–	–	–	–	+	–
ko:K04567	+	+	+	+	+	+	–	+
ko:K04568	–	–	–	–	+	–	–	–
ko:K05822	–	–	–	–	–	+	–	–
ko:K05823	–	–	–	–	–	+	–	–
ko:K05824	–	+	–	–	–	–	–	–
ko:K05826	–	–	–	–	–	–	–	+
ko:K05827	–	–	–	–	–	–	–	+
ko:K05828	–	–	–	–	–	–	–	+
ko:K05829	–	–	–	–	–	–	–	+
ko:K05830	–	–	–	–	–	–	–	+
ko:K05831	–	–	–	–	–	–	–	+



パスウェイ上の遺伝子から立体構造へ

LinkDBとPDB

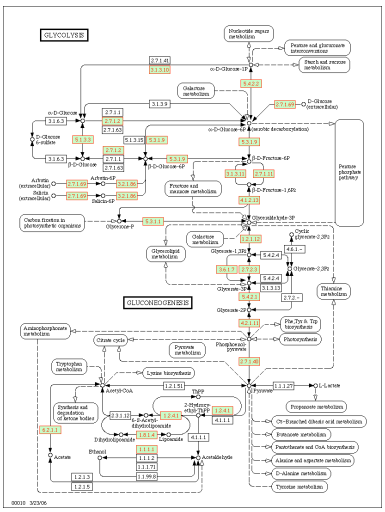
By Toshiaki Katayama

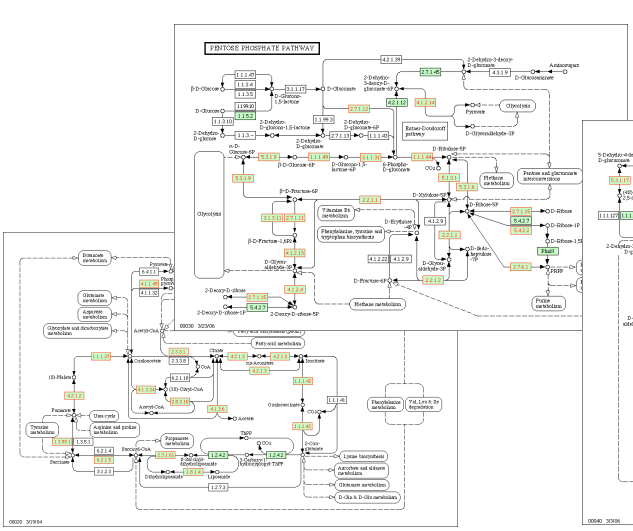
データベース間のリンク

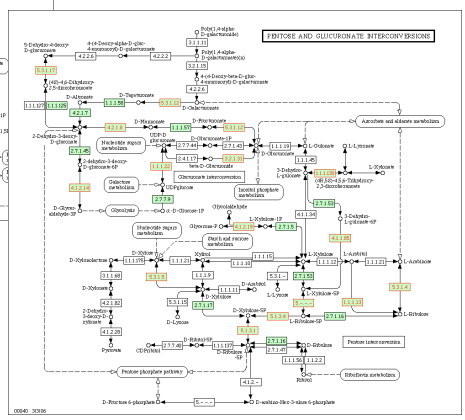
GenomeNet では、KEGG の GENES や PATHWAY など様々なデータベース、配列データベースの GenBank や UniProt、モチーフの PROSITE、立体構造の PDB など、各種のデータベース間で関連するエントリ同士の関係を、LinkDB というリンクのデータベースとして保持しています。LinkDB を用いることで、着目している KEGG GENES の遺伝子から、酵素番号などを経由して、関連する立体構造までリンクが辿れるかどうか、といった検索が可能です。

ここでは、指定したパスウェイ上の遺伝子から PDB データベースにリンクが辿れるかどうかを調べ、立体構造が見つかった遺伝子についてパスウェイに色付けしてみます。

```
#!/usr/bin/env ruby

require 'bio'

path = ARGV.shift || "path:eco00010"                    # 引数でパスウェイ番号を指定
serv = Bio::KEGG::API.new

structures = Hash.new                                   # PDB のリンク先を格納

genes = serv.get_genes_by_pathway(path)                 # パスウェイ上の遺伝子リスト
genes.each do |gene|                                    # 各遺伝子について ...
  print gene                                            # 遺伝子名を表示
  if links = serv.get_all_linkdb_by_entry(gene, 'pdb')  # PDB へのリンクを検索
    links.each do |link|                                # それぞれのリンクについて ...
      structures[gene] ||= Array.new                    # リンク先のリストを格納する配列
      structures[gene] << link.entry_id2                # リンク先 PDB のエントリを追加
      print "\t", link.entry_id2                        # PDB のエントリ名を表示
    end
  end
  puts
end

list = structures.keys                                  # 構造が見つかった遺伝子のリスト
puts serv.mark_pathway_by_objects(path, list)           # 色付けした画像を生成 URL 表示
```







タンパク質の細胞内局在

PSORTによる予測

By Toshiaki Katayama

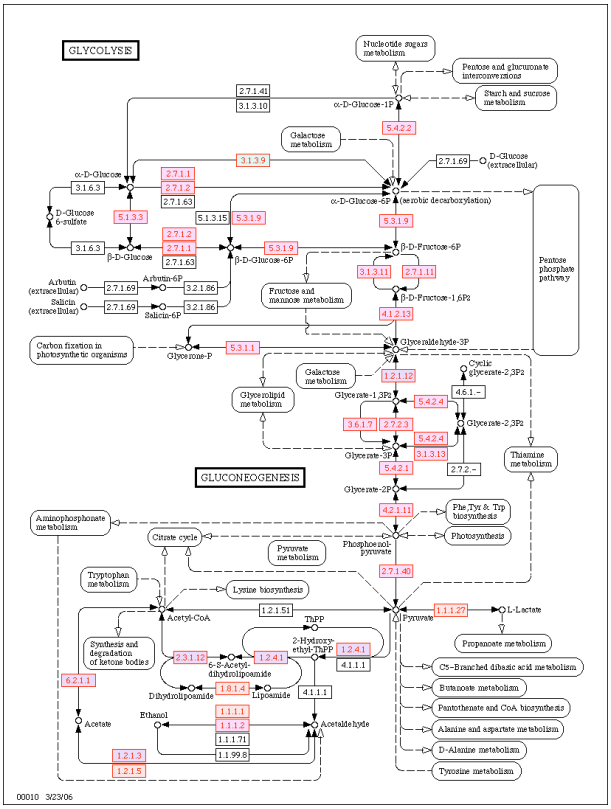
PSORTとは

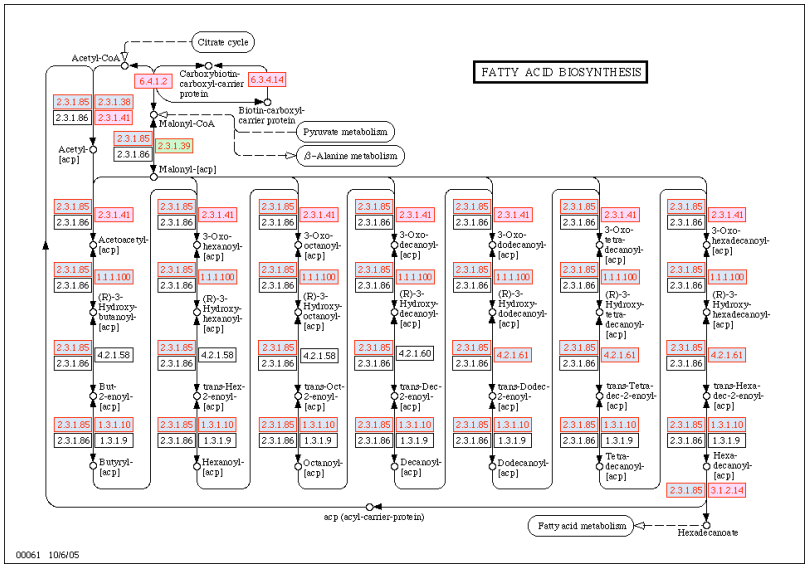
PSORT（http://psort.hgc.jp/）はアミノ酸配列の特徴からタンパク質の細胞内局在を予測するプログラムです。動物に強い PSORT II やN末のシグナル配列からオルガネラにも強い iPSORT、PSORT II の更新版である WoLF PSORT、バクテリア用の PSORTb など最適化の違いによる亜種が開発されています。

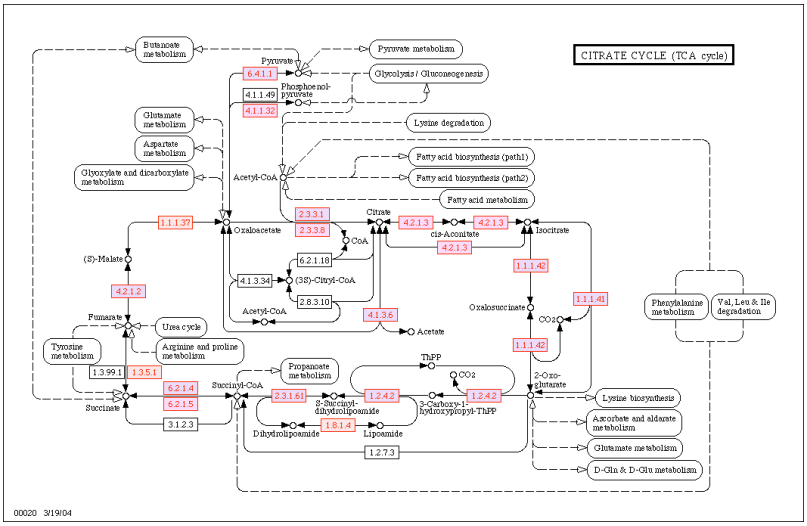
生物種	PSORT	PSORT II	iPSORT
バクテリア	○	×	×
酵母	○	◎	○
動物	○	◎	○
植物	○	△	○
精度（目安）	50%	60%	70-80%

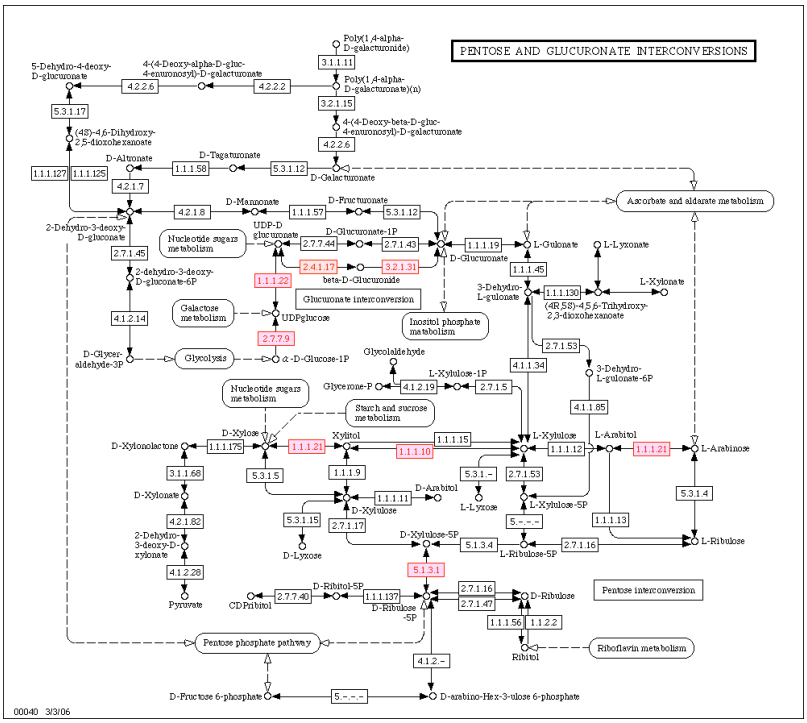
指定したパスウェイ上の遺伝子の細胞内局在を PSORT II で予測し、局在別に色分けしてパスウェイに色付けしてみます。

```
#!/usr/bin/env ruby

require 'bio'

keggapi = Bio::KEGG::API.new                    # KEGG API のサーバに接続
psort2  = Bio::PSORT::PSORT2.imsut              # 東大医科研の PSORT2 サーバに接続
pathway = ARGV.shift || "path:hsa00010"         # 引数でパスウェイ番号を指定
palette = {                                     # 細胞内局在別の色パレット
  'csk' => "#FAE1CF",   # 'cytoskeletal'
  'cyt' => "#FAD2FA",   # 'cytoplasmic'
  'nuc' => "#D3E5F5",   # 'nuclear'
  'mit' => "#E7D4FA",   # 'mitochondrial'
  'ves' => "#C3FAC3",   # 'vesicles of secretory system'
  'end' => "#FAE1E1",   # 'endoplasmic reticulum'
  'gol' => "#F7F7D7",   # 'Golgi'
  'vac' => "#DCFADC",   # 'vacuolar'
  'pla' => "#DAF5F5",   # 'plasma membrane'
  'pox' => "#EEE1FA",   # 'peroxisomal'
  'exc' => "#FAE1FA",   # 'extracellular, including cell wall'
  '---' => "#FAFAFA",   # 'other'
}
fg_list = Array.new                             # 文字色を格納する配列
bg_list = Array.new                             # 背景色を格納する配列
genes = keggapi.get_genes_by_pathway(pathway)
genes.each do |gene|
  seq = keggapi.bget("-f -n a #{gene}")         # FASTA 形式の配列を取得
  loc = psort2.exec(seq).pred                   # PSORT2 で予測
  puts "#{gene}\t#{loc}"
  fg_list << "#FF0000"                          # 文字色は一定にする
  bg_list << palette[loc]                       # 局在により背景色を変更
end

url = keggapi.color_pathway_by_objects(pathway, genes, fg_list, bg_list)
keggapi.save_image(url, "#{pathway}.gif")
```










```
% ruby psort2.rb path:hsa00010
hsa:10327       cyt
hsa:124         cyt
hsa:125         cyt
hsa:126         end
hsa:127         end
hsa:128         cyt
hsa:130         end
hsa:130589      cyt
hsa:131         end
hsa:137872      cyt
hsa:160287      end
hsa:1737        mit
hsa:1738        end
hsa:2023        cyt
hsa:2026        cyt
hsa:2027        cyt
hsa:217         mit
hsa:218         mit
hsa:219         mit
hsa:220         cyt
hsa:2203        cyt
hsa:221         end
hsa:222         gol
hsa:223         cyt
hsa:224         cyt
hsa:226         cyt
hsa:229         cyt
hsa:230         cyt
hsa:2538        pla
hsa:2597        cyt
hsa:26330       mit
hsa:2645        cyt
hsa:2821        cyt
hsa:3098        cyt
hsa:3099        cyt
hsa:3101        cyt
hsa:388642      cyt
hsa:3939        end
hsa:3945        end
hsa:3948        end
hsa:441531      cyt
hsa:501         cyt
hsa:5160        mit
hsa:5161        mit
hsa:5162        cyt
hsa:5211        cyt
hsa:5213        cyt
hsa:5214        cyt
hsa:5223        cyt
hsa:5224        cyt
hsa:5230        cyt
hsa:5232        cyt
hsa:5236        cyt
hsa:5238        cyt
hsa:5313        cyt
hsa:5315        cyt
hsa:55902       cyt
hsa:669         cyt
hsa:7167        cyt
hsa:8050        mit
hsa:84532       mit
hsa:8789        cyt
hsa:92483       mit
hsa:97          cyt
hsa:98          nuc
```

ワークブック

遺伝子発現データとパスウェイ

By Toshiaki Katayama

KEGG API と KEGG EXPRESSION

KEGG API を使って、遺伝子発現データからパスウェイに色をつけてみましょう。発現データは KEGG EXPRESSION データベース (http://www.genome.jp/kegg/expression/) から得ることにします。

```
#
# KEGG/EXPRESSION
# filename : ex0000258.dat
# created  : 2003/12/10 17:04:37
# organism : bsu
# submitter: Yasutaro FUJITA
#
#ORF     x       y       Control-sig     Control-bkg     Target-sig      Target-bkg
BG10065 1       1       938     189     725     249     dnaA, dnaH, dnaJ, dnaK
BG10066 2       1       2692    189     2253    249     dnaN, dnaG, dnaK
BG10067 3       1       958     189     444     249     yaaA
BG10068 4       1       6703    189     2533    249     recF
BG10069 5       1       496     189     327     249     yaaB
  :
```

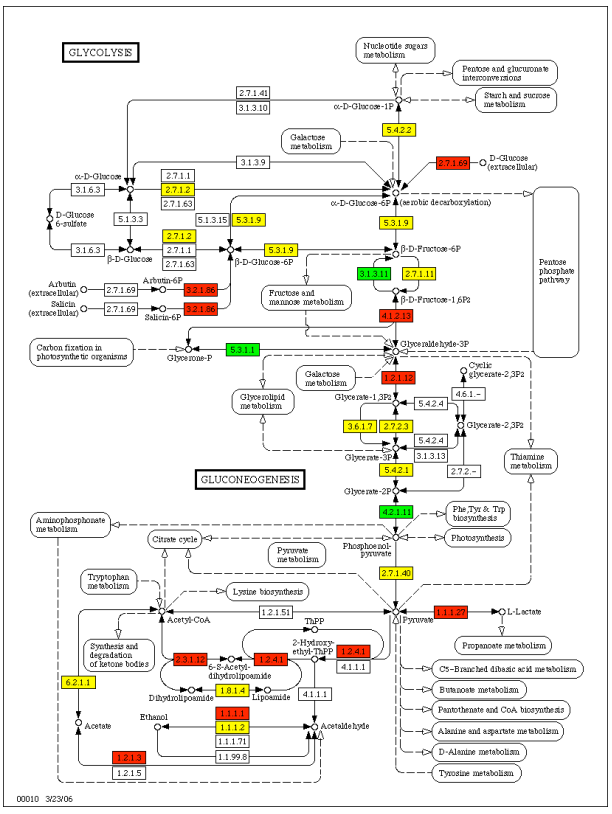
このように KEGG EXPRESSION のデータは単純で、遺伝子名とチップ上の位置、コントロールとターゲットのシグナル強度がタブ区切りで並んでいることが分かります。

```
#!/usr/bin/env ruby

require 'bio'

# 引数から、色づけを行なうパスウェイと KEGG EXPRESSION のファイル名を取得
pathway    = ARGV.shift || "path:bsu00010"
expression = ARGV.shift || "ex00258.dat"

# KEGG API のサーバに接続
keggapi = Bio::[          ]::[      ].new

# 遺伝子発現量の比を保持する Hash を用意
gene_ratio = Hash.new

# KEGG EXPRESSION のファイルを開き、１行ごとに読み込む
File.open(expression) do |file|
  file.each do |line|
    # コメント行はスキップする
    next if line[/^#/]
    # 行末の改行を取って、タブで分割
    data = line.chomp.split("\t")
    # 第１カラム目は遺伝子名
    gene_id = data[0]
    # 第４,５カラム目が control の発現量とバックグラウンドのノイズ
    c_sig = data[3].to_i - data[4].to_i
    # 第６,７カラム目が target の発現量とバックグラウンドのノイズ
    t_sig = [          ].to_i - [          ].to_i
    # ノイズを差し引いた target のシグナルと control のシグナルの比を求める
    ratio = t_sig.to_f / c_sig.to_f
    # 計算した比の値を gene_ratio に記録しておく
    gene_ratio[gene_id] = ratio
  end
end
```

```
# 文字色と背景色の配列
fg_list = Array.new
bg_list = Array.new

# KEGG PATHWAY 上の遺伝子名のリストを取得
genes = keggapi.get_[          ]_by_pathway(pathway)

genes.each do |gene|
  # KEGG EXPRESSION にあわせて遺伝子名から生物種コード部分を取り除く
  gene_id = gene.sub(/[a-z]{3}:/, '')
  # 遺伝子発現の比の値に応じて遺伝子ごとに背景色を設定
  ratio = gene_ratio[gene_id]
  if ratio < 0.5                    # 発現量が 1/2 以下に減った遺伝子は緑色に
    fg_list << "#000000"
    bg_list << "#00ff00"
  elsif ratio < 2                   # 発現量の変化があまりない遺伝子は黄色に
    fg_list << "#000000"
    bg_list << "#ffff00"
  else                              # 発現量が 2 倍以上増加している遺伝子は赤色に
    [          ] << "#000000"
    [          ] << "#ff0000"
  end
end

# 指定したパスウェイに対し、遺伝子のリスト、文字色のリスト、背景職のリストを指定
puts keggapi.color_[          ]_by_objects(pathway, genes, fg_list, bg_list)
```

このプログラムを color_pathway.rb という名前で保存し、枯草菌の Glycolysis パスウェイと枯草菌の glucose repression における遺伝子発現データを与えて実行してみましょう。

```
% wget ftp://ftp.genome.jp/pub/kegg/expression/bsu/ex0000258.dat
% ruby color_pathway.rb path:bsu00010 ex0000258.dat
```



Tips集

ちょっとした（バッド？）ノウハウ

By Toshiaki Katayama

局所変数の参照

最初とまどいがちな点ですが、ブロックローカルの変数は外から見えないため、ブロックを抜けたあとで値を取り出したい場合は、あらかじめ同じ名前のローカル変数を定義しておく必要があります。

```
a = nil
ary.each do |a|
  a.something
end
puts a
```

Array#each の場合は、この代りに

```
for a in ary
  a.something
end
puts a
```

の構文を使うと、スコープが局所化されないので、あらかじめ宣言しておかなくてもアクセスできます（が Ruby っぽくないかも）。

I/Oをバッファリングしない

時間のかかるプログラムの場合、ログファイルに処理済みのエントリ名などを出力することがあります。しかし標準出力のようにデフォルトでバッファリングされる I/O では、数文字ずつの出力をリアルタイムに tail -f など監視できません。このような場合には sync メソッドでバッファリングを止められます。

```
$stdin.sync = true
$stdout.sync = true
```

untabifyみたいなの

高林さんが perldoc -q tab のコードを変換されたもの（日記かどこかから頂きました）。

```
class String
  def untabify!
    true while self.gsub!(/\t+/) {
      ' ' * ($&.length * 8 - $`.length % 8)
    }
  end
end
```

requireしたモジュールの一覧

```
% ruby -r uri -e 'p $"'
```

定数一覧

```
% ruby -r bio -e 'p Bio.constants'
```

awkの代りに使う

```
% ruby -lan -F: -e 'p $F' /etc/passwd
```

$DEBUGをtrueにする

```
% ruby -d script.rb
```

デバッガを使う

```
% ruby -r debug script.rb
```

プロファイラを使う

```
% ruby -r profile script.rb
```

configure時の設定を調べる

```
% ruby -r rbconfig -r pp
  -e 'pp Config::CONFIG'
```

文字の前につく記号	
?文字	ASCII文字コード `p ?a` (Kernel::test も参照)
***変数名**	配列への代入 `a, *b = 1, 2, 3` や配列の展開 `foo(*b)`
&文字列	メソッドに対するブロックパラメータ `hoge(foo, &bar)`
:文字列	シンボルオブジェクトのリテラル `p :a.id`
::文字列	トップレベルの定数など `::Hoge`
\文字 (¥文字)	バックスラッシュ記法のリテラル `\n` (または `¥n`)
0数字	８進数のリテラル `p 011`
%文字	printf(format) の書式指定