8 Block 8: Mustererkennung und Reguläre Ausdrücke

8.1 Lernziele

Ziel dieses Blocks ist es zu verstehen und anzuwenden

- was *Musterkennung* im Kontext von **`Perl`** Programmieren bedeutet;
- wie Muster mithilfe *regulärer Ausdrücke* in **`Perl`** formuliert werden
- und wie die **`Perl`**-Mustererkennungs-Funktionalität in der Bioinformatik für
  - die *Daten-Extraktion aus Textdateien* und
  - und die *Sequenz-Analyse*

  eingesetzt werden

8.2 Theorie

8.2.1 Was ist Mustererkennung?

Einer der großen Vorteile der Programmiersprache **`Perl`** ist die Tatsache, dass Elemente zur *Mustererkennung* direkt in die Syntax integriert sind.

Mustererkennung im Kontext von **`Perl`** muss als *textuelle* Mustererkennung verstanden werden, also dem Suchen nach Vorkommen bestimmter Zeichen (-Kombinationen) in einem Wort (zusammengesetztem Zeichen) oder einem kompletten Text (mehrere Wörter, in mehreren Zeilen).

Wir haben bereits zwei *bioinformatische Problemstellungen* kennengelernt, bei denen *Muster* von einem **`Perl`** Programm erkannt werden müssen:

8.2.2 Analysieren von Sequenzdaten

Wie wir bereits in Block 1 besprochen haben können DNA- bzw. Proteinsequenzen vereinfacht als Wörter /Zeichenketten aufgefasst werden, die auf den Einzelzeichen-Alphabeten für *Nukleotide*

Symbol	Bedeutung	Ursprung der Bezeichnung
G	G	Guanin
A	A	Adenin
T	T	Thymin
C	C	Cytosin
U*	U	Uracil
R	G oder A	puRin
Y	T oder C	pYrimidin
M	A oder C	aMino
K	G oder T	Keto
S	G oder C	Starke Wechselwirkung (3 H Brücken)
W	A oder T	Schwache Wechselwirkung (2 H Brücken)
H	A oder C oder T	nicht-G, H folgt G in Alphabet
B	G oder T oder C	nicht-A, B folgt A
V	G oder C oder A	nicht-T (nicht-U), V folgt U
D	G oder A oder T	nicht-C, D folgt C
N	G oder A oder T oder C	aNy

* Für Ribonukleotidsequenzen (RNAs)

und *Aminosäuren*

Aminosäure	Einzelzeichen-Code	Nukleotid-Triplet (5'-3')
Glycin	**G**	GGN
Alanin	**A**	GCN
Valin	**V**	GTN
Leucin	**L**	*YTN* (CTN und TTR)
Isoleucin	**I**	ATH
Prolin	**P**	CCN
Phenylalanin	**F**	TTY
Tyrosin	**Y**	TAY
Cystein	**C**	TGY
Methionin	**M**	ATG

Histidin	**H**	CAY
Lysin	**K**	AAR
Arginin	**R**	*MGN* (CGN und AGR)
Tryptophan	**W**	TGG
Serin	**S**	*WSN* (TCN und AGY)
Threonin	**T**	ACN
Aspartat	**D**	GAY
Glutamat	**E**	GAR
Asparagin	**N**	AAY
Glutamin	**Q**	CAR
Aspartat oder Asparagin	**B**	RAY
Glutamat oder Glutamin	**Z**	SAR
Terminator	**.**	*TRR* (TAR und TGA)
Unbekannt	**X**	NNN

basieren.

D.h. wir können die textuelle Mustererkennungs-Funktionalität von **`Perl`** auch auf DNA- und Proteinsequenz-"Texte" anwenden. Anwendungsbeispiel für die Verwendung von **`Perl`** Ausdrücken zur Mustererkennung, die wir im Folgenden besprechen werden, sind:

- Finden von Restriktionsenzym-Schnittstellen auf einer DNA-Sequenz
- Finden von Proteinsequenz-Motiven auf einer Protein-Sequenz

Nehmen wir an wir betrachten eine Zeichenkette (eine fiktive Proteinsequenz)

```
LNVAQYAAAMQQHYALYPMWQSQPPAVALNVAHHQHARNNAA
AAAAGVALYRMWQGPPVDGGKYPMWQSVDGGFFLYPWMQSSL
HHHLSTLQQQHHQHLYRWQSHLHYGLQPPAVALNVAQYAAAM
QQHYAKYPMWQGAAAAAARNNAAAAAGVAAHHQHHLHYPPVDG
GVDGG
```

und wir wollen auf dieser Zeichenkette das Muster

`LYPMWQS`

und

`LYRMWQG`

und

`KYPMWQS`

und

`KYPMWQG`

(fiktive Proteinsequenz-Motive) finden,

```
LNVAQYAAAMQQHYALYPMWQSQPPAVALNVAHHQHARNNAA
AAAAGVALYRMWQGPPVDGGKYPMWQSVDGGFFLYPWMQSSL
HHHLSTLQQQHHQHLYRWQSHLHYGLQPPAVALNVAQYAAAM
QQHYAKYPMWQGAAAAAARNNAAAAAGVAAHHQHHLHYPPVDG
GVDGG
```

dann unterstützt **`Perl`** diese Aufgabenstellung ausgezeichnet durch seine Musterkennungs-Funktionalität. Wir können das zu suchende Muster folgendermaßen formulieren:

Zeichen **`L`** *oder* **`K`** *gefolgt von* **`Y, P, M, W, Q`** *und* **`S`** *oder* **`G`**.

In der **Perl**-spezifischen Notation zur Formulierung von Mustern lassen sich alle zu betrachtenden Zeichenkombinationen in einem Ausdruck zusammenfassen:

```
/[LK]YPMWQ[SG]/
```

Mehr zur **Perl**-Syntax zur Musterformulierung: siehe unten.

8.2.3 Lesen von Textdateien

Ein weiteres Anwendungsbeispiel für die Verwendung der Mustererkennungs-Funktionalität von **Perl** ist das *Lesen, Zerlegen und Analysieren* von Textdateien. Diese Anwendung haben wir bereits in Block 5 (Seite 7) am Beispiel der Sequenzdaten-Extraktion aus einer FASTA formatierten Datei

```
>Test.fasta; eine einfache Testsequenz in FASTA Format
AATGCTTGCGGGCTATATGCCTTGACGATTTGGGCCCGAT
ACCACCAC
```

kennengelernt.

Wir können mit einer **while** Schleife über alle Zeilen einer Datei in einem **if..else** Block

```
while ($Zeile = <FILE>) {
    if ($Zeile =~ /^>/) {
        next;
    }
    else {
        print $Zeile;
    }
}
```

aus dem Programm **Seq_In_Dat_raw.pl** die Kopfzeile von den Zeilen mit der Roh-Sequenz unterscheiden.

Der diskriminierende Faktor ist: Die Kopfzeile beginnt mit dem **>** Zeichen, alle anderen (und das sind *nur* Zeilen mit Roh-Sequenz) nicht!

In der Zeile

```
if ($Zeile =~ /^>/)
```

ist mit **/^>/** ein *Muster* formuliert, nämlich:

...Zeile fängt mit dem Zeichen **>** *an.*

Wenn das Muster also am Anfang (*Zeilenanfang* ist definiert durch den Operator **^**) des Inhalts von **$Zeile** (enthält die einzelne Zeilen der Sequenzdatei) gefunden ist der Ausdruck **$Zeile =~ /^>/** wahr, gibt also den Wert **true** zurück, die Bedingung ist erfüllt und der **if** Block wird ausgeführt.

Diese Art von Programmen wird verwendet wenn aus einer Eingabe-Datei Daten extrahiert werden müssen, z.B. um sie einem anderen Programm / einer anderen Anwendung zur Verfügung zu stellen.

8.3 Praxis

Muster werden in **Perl** mithilfe einer formalisierten Sprache, so genannten *regulären Ausdrücken*, formuliert. Reguläre Ausdrücke sind Formulierungen, mit denen man abstrakt den Aufbau von Zeichenketten / Texten beschreiben kann.

Das Erlernen der **Perl** Syntax zur Mustererkennung ist komplex, muss an vielen Beispielen geübt werden, und kann in diesem Kurs nicht annähernd erschöpfend behandelt werden.

8.3.1 Datenextraktion aus eine FASTA-Sequenz-Datei

Betrachten wir eine Erweiterung von **Seq_In_Dat_raw.pl**, das Programmbeispiel **Seq_In_Dat_regex.pl**, in dem die Daten aus **Test.fasta** beim Einlesen durch die Formulierung von *Mustern* mithilfe von *regulären Ausdrücken*, noch weiter *aufgegliedert* werden (der entscheidende Block ist gelb hervorgehoben):

```
#!/usr/bin/perl -w

# Definition des Namens der Datei, die geoeffnet werden soll
$Datei_Name = 'Test.fasta';
#print "Die Roh-Sequenz  von $Datei_Name ist:\n\n";
# Offnen der Datei zum Lesen mit dem  "<" Operator
open (FILE, "<$Datei_Name");
# Lesen jeder Zeile in einer Schleife:
while ($Zeile = <FILE>) {
    # Speichern der gesamten Kopfzeile in $1
    if ($Zeile =~ /^(>.*)/) {
        print "Komplette Kopfzeile:\t$1\n";
        # Extraktion des Sequenz-Namens
        $1 =~ /(\w*\.fasta);(.*)/;
        print "Sequenz-Name: $1\n";
        print "Sequenz-Annotation: $2\n";
    }
    # Speichern der Sequenzzeilen in $1
    elsif ($Zeile =~ /^([AGCT]+)/) {
        print "Sequenz-Zeile:\t$1\n";
    }
    else {
        next;
    }

}
close FILE;
exit;
```

Das Programm **Seq_In_Dat_regex.pl** liest den Inhalt der Datei **Test.fasta** aus dem aktuellen Arbeitsverzeichnis, iteriert in einer **while** Schleife über alle Zeilen des Datei-Inhalts und extrahiert aus diesen Zeilen in einem **if...elsif...else** Block folgende Daten:

- Die komplette Kopfzeile:

  >Test.fasta; eine einfache Testsequenz in FASTA Format

- Den Namen der Sequenz (aus der Kopfzeile):

  Test.fasta

- Die Annotation der Sequenz (aus der Kopfzeile):

  eine einfache Testsequenz in FASTA Format

- Die Sequenzzeilen

  AATGCTTGCGGGCTATATGCCTTGACGATTTGGGCCCGAT

  und

  ACCACCAC

8.3.1.1 Formulierung Regulärer Ausdrücke

Betrachten wir im Detail wie mithilfe von regulären Ausdrücken in einem **if..elsif..else** Block

```
if ($Zeile =~ /^(>.*)/) {
        print "Komplette Kopfzeile:\t$1\n";
        # Extraktion des Sequenz-Namens
        $1 =~ /(\w*\.fasta);(.*)/;
        print "Sequenz-Name: $1\n";
        print "Sequenz-Annotation: $2\n";
    }
    # Speichern der Sequenzzeilen in $1
    elsif ($Zeile =~ /^([AGCT]+)/) {
        print "Sequenz-Zeile:\t$1\n";
    }
    else {
        next;
    }
```

diese Daten extrahiert werden. In der ersten Zeile der Bedingung

```
if ($Zeile =~ /^(>.*)/) {
```

ist mit **/^(>.*)/** das erste Muster definiert. Muster werden immer in Schrägstrichen

/Muster/

definiert. Das Muster **/^>.*/**, - ohne runde Klammern **()** -, bedeutet:

> Alle Zeilenanfänge (definiert durch das Metazeichen **^**), unmittelbar gefolgt von dem Zeichen **>**, unmittelbar gefolgt von null oder mehreren (definiert durch das Metazeichen *****) beliebigen Zeichen (definiert durch das Metazeichen **.**).

Das Muster besteht also aus

- *Textzeichen*, die genau ein Zeichen beschreiben (in diesem Fall nur **>**) und
- sogenannten *Metazeichen* (in diesem Fall **^**, **.** und *****), die eine Gruppe oder einen allgemeinen Typ von Zeichen beschreiben.

Diese Zeichenkombination **^>.*** zur Beschreibung eines textuellen Musters verwendet wird, wird im Programmier-Jargon als *regulärer Ausdruck* bezeichnet.

Das Muster **/^>.*/**erfasst offensichtlich die FASTA-Kopfzeile, die immer mit **>** beginnt, gefolgt von einer beliebigen Anzahl weiterer Zeichen.

Durch die Verwendung von runden Klammern **/^(>.*)/** in der Musterdefinition kann ein Teil des erkannten Muster in der Standardvariablen **$1** aufgefangen werden:

Wenn das Muster **/^(>.*)/** gefunden wurde kann also in der nächsten Zeile

```
print "Komplette Kopfzeile:\t$1\n";
```

über **$1** auf das gefundene Muster zugegriffen werden. Analog würde **$1** nach **/^>(.*)/** die gesamte Kopfzeile ohne **>** enthalten. Im Beispiel ist als der Wert von **$1** nun

>Test.fasta; eine einfache Testsequenz in FASTA Format

In der nächsten Zeile

```
$1 =~ /(\w*\.fasta);(.*)/;
```

wird nun auf den aktuellen Wert von **$1** zugegriffen und auf diesem das Muster **/(\w*\.fasta);(.*)/** gesucht. Die reguläre Expression liest sich folgendermaßen:

> Null oder mehrere (definiert durch das Metazeichen *****) Wortzeichen (definiert durch das Metazeichen **\w**), das sind alle Buchstaben, Ziffern und **_**, gefolgt von einem Punkt (definiert durch das Metazeichen **** gefolgt von **.**), gefolgt von der Zeichenkette **.fasta**, gefolgt von einem Semikolon **;**, gefolgt von null oder mehreren (definiert durch das Metazeichen *****) beliebigen Zeichen (definiert durch das Metazeichen **.**).

Beachten sie die Verwendung des *Escape*-Metazeichen **** an zwei Stellen

```
/(\w*\.fasta);(.*)/
```

in dem regulären Ausdruck. Wenn sie das Muster stattdessen als

```
/(w*.fasta);(.*)/
```

formulieren würde statt nach Wortzeichen, nach null oder mehreren Vorkommen des Zeichen **w** und statt nach einem Punkt **.** nach genau EINEM beliebigen Zeichen gesucht.

In der Musterdefinition werden zwei Paare von Klammern gesetzt:

```
/(\w*\.fasta);(.*)/
```

Analog zu dem ersten Muster werden dadurch Teile des vorkommenden Muster aufgefangen: die Übereinstimmung mit **\w*\.fasta** in der Variablen **$1** und die Übereinstimmung mit **.*** in **$2**.

So können beliebig viele Klammern in Mustern gesetzt werden und auf den Inhalt über **$1, $2, $3...$n** zugegriffen werden.

$1 hat im Beispiel also den Wert

```
Test.fasta
```

und **$2** den Wert

```
eine einfache Testsequenz in FASTA Format
```

Im anschließenden **elsif** Block

```
elsif ($Zeile =~ /^([AGCT]+)/) {
    print "Sequenz-Zeile:\t$1\n";
}
```

werden über das Muster **/^([AGCT]+)/** die Zeilen mit Sequenzdaten-Inhalt extrahiert. Die reguläre Expression liest sich folgendermaßen:

> Ein oder mehrere Zeichen (definiert durch das Metazeichen **+**) der Gruppe **AGCT** Wortzeichen (definiert durch Metazeichen **[]**).

Mit diesem Ausdruck und der Verwendung runder Klammern wird **$1** iterativ (im Beispiel sind es zwei Iterationen) erst mit dem Wert

```
AATGCTTGCGGGCTATATGCCTTGACGATTTGGGCCCGAT
```

und dann mit

```
ACCACCAC
```

reinitialisiert.

8.3.1.2 Metazeichen-Übersicht

Metazeichen sind Zeichen mit einer speziellen Bedeutung bei der Formulierung von regulären Ausdrücken. Metazeichen sind *Platzhalter* (engl. *wildcards*), die spezielle Steuerungsfunktionen haben oder für Zeichenklassen oder –gruppen stehen. Im folgenden finden sie eine Übersicht und Erklärung der wichtigsten (nicht aller!) Metazeichen:

Metazeichen	Erklärung	Beispiel	Übereinstimmung
\	Escape-Zeichen für die Suche nach Sonderzeichen	/\$Variable/	$Variable
		/picker@embl\.de/	picker@embl.de
		/GenBank|NM12345/	GenBank|NM12345
|	Alternative Zeichen (**oder**)	/(H|h)all(o|i)/	Hallo, hallo, Halli, halli
()	Gruppierungen von Zeichen	/(ha)*llo/	llo, hallo, hahallo, hahahallo

		`/(ha)?llo/`	`llo, hallo`
`[]`	Liste von Zeichen	`/[hbg]allo/`	`Hallo, ballo, gallo`
`{}`	Numerische Anzahl von Zeichenwiederholungen	`/hal{2}o/`	`hallo` (genau 2 Zeichen)
		`/hal{2,}o/`	`hallo, halllo, hallllo` etc. (mindestens 2 Zeichen)
		`/hal{2,4}o/`	(mindestens 2, höchsten 4 Zeichen)

`^`	Muster am Zeichenanfang	`/^genbank/`	`blast\n`
			`genbank\n`
			`swissprot\n`

`$`	Muster am Zeichenende (direkt vor `\n`)	`/$treffer/`	`am ende treffer\n`
`*`	Null- oder mehrmaliges Vorkommen (des Zeichens links von `*` im Muster)	`/hal*o/`	`hao, halo, hallo, halllo`
`+`	Zeichenwiederholung (mindestens EIN Vorkommen des Zeichens links von `+` im Muster)	`/hal+o/`	`hallo, halllo etc.`
`?`	Null- oder einmaliges Vorkommen (des Zeichens links von `?` im Muster)	`/hal?o/`	`hao, halo`
`.`	EIN beliebiges Zeichen	`/zir.us/`	`zirkus, zirrus`

Wenn sie in einem Muster ein genaues Zeichen auffangen wollen, das mit einem Metazeichen übereinstimmt, dann muss diesem Zeichen das Escape-Zeichen `\` vorangestellt werden (siehe Beispiele oben).

Weitere Metazeichen sind (keine vollständige Liste!):

Metazeichen	Bedeutung
`\t`	Tabulator
`\n`	Zeilenumbruch
`\d`	Ziffer
`\D`	Alle Zeichen außer Ziffern
`\w`	Wortzeichen (alle Buchstaben, Ziffern und `_`)
`\W`	Alle Zeichen außer Wortzeichen
`\s`	Leerzeichen oder Tabulator oder Zeilenumbruch
`\S`	Alle Zeichen außer `\s`

8.3.1.3 Muster-Modifikatoren

Mithilfe von Muster-Modifikatoren kann das Verhalten der **`Perl`** *Musterkennungs-Maschine* verändert werden. Einige der am meisten verwendeten Modifikatoren sind (keine vollständige Liste!):

Notation	Bedeutung
`/Muster/i`	ignorieren der Groß- und Kleinschreibung
`/Muster/g`	globales Vorkommen – *ersetze so oft wie möglich*
`/Muster/s`	`.` (ein beliebiges Zeichen) stimmt auch mit `\n` überein

8.3.1.4 Standardvariablen zum Zugriff auf Muster-Übereinstimmungen

Die Standardvariablen **$1, $2, $3 ... $n** werden verwendet um durch runde Klammern auf *Teile einer Muster-Übereinstimmung* zuzugreifen. Also liefert

/(Muster1)(Muster2)(Muster3)/

die Werte

$1 = Muster1

$2 = Muster2

$3 = Muster3

während

/(Muster1Muster2Muster3)/

nur

$1 = Muster1Muster2Muster3

liefert.

Die Standardvariable **$&** dient zum Zugriff auf die letzte komplette Übereinstimmung mit dem Muster. Ein Beispiel:

Wenn

```
$Zeile = '>Test.fasta; eine einfache Testsequenz in FASTA Format';
```

dann kann über **$&**

```
$Zeile = '>Test.fasta; eine einfache Testsequenz in FASTA Format';
$Zeile =~ /^>.*/;
print $&;
```

auf die *gesamte Zeile mit der Übereinstimmung* zugegriffen werden. Mit **$1...$n** kann auf Teile der Übereinstimmung zugegriffen werden. Das Programm **Muster_ganz.pl**

```
#!/usr/bin/perl -w
$Zeile = '>Test.fasta; eine einfache Testsequenz in FASTA Format';
$Zeile =~ /^>(.*)/;
print "$&\n";
print "$1\n";
exit;
```

liefert also folgende Ausgabe auf dem Bildschirm:

```
>Test.fasta; eine einfache Testsequenz in FASTA Format
Test.fasta; eine einfache Testsequenz in FASTA Format
```

Die Variablen **$'** und **$`** dienen zum Zugriff auf die Zeichen, die einer Musterübereinstimmung folgen bzw. vorangehen.

Das Beispiel **Muster_vor_nach.pl**

```
#!/usr/bin/perl -w
$Zeichen = '123456789';
$Zeichen =~ /456/;
print "Ganzes Zeichen:\t$Zeichen\n";
print "Muster:\t456\n";
print "Zeichen vor der Musteruebereinstimmung:\t$`\n";
print "Zeichen nach der Musteruebereinstimmung:\t$'\n";
exit;
```

liefert also die Ausgabe

```
Ganzes Zeichen: 123456789
```

```
Muster: 456
Zeichen vor der Musteruebereinstimmung: 123
Zeichen nach der Musteruebereinstimmung:        789
```

8.3.1.5 Verwendungsbeispiele

`/[a-z]+/`	alle Kombinationen aus Kleinbuchstaben, die beliebig oft aber mindestens einmal vorkommen.
`/[A-Z][a-z]+/`	alle Zeichen-Kombinationen, die mit einem Grossbuchstaben beginnen, gefolgt von einer beliebigen Anzahl von Kleinbuchstaben.
`/[0-9]{6}/`	alle sechsstelligen Ganzzahlen.
`/[^0-9]/`	Alle Zeichen, die nicht (Metazeichen `^` steht in eckigen Klammern `[]` nicht für den Zeilenanfang sondern für eine *Negation der Gruppierung*) eine der Ziffern 0-9 sind.
`/\w{5,}/`	Wörter mit mindestens 5 Zeichen.
`$Variable =~ s/\s//g`	Entfernt auf dem Wert von `$Variable` alle (definiert durch den Muster-Modifikator `/g`) Leerzeichen, Tabulatoren und Zeilenumbrüche (definiert durch das Metazeichen `\s`). *Nützlich um aus einer biologischen Sequenz alle unerwünschten Zeichen zu entfernen.*
`/^\s*$/`	Übereinstimmung mit einem Zeichen, das vom Zeilenanfang (Metazeichen `^`) aus null oder mehrere (Metazeichen `*`) Leerzeichen (Metazeichen `\s`) bis zum Ende der Zeile (Metazeichen `$`) enthält. Das Muster passt also auf alle Zeichen, die keinen Zeilenumbruch haben außer einem am Ende des Zeichens. *Nützlich um zu testen ob eine interaktive Benutzereingabe nichts als Leerzeichen enthält.*

Weiteres ausführliches Referenz-Material zur Verwendung regulärer Ausdrücke finden sie in der **`Perldoc`** Online-Dokumentation unter:

http://www.perldoc.com/perl5.6/pod/perlre.html - Regular-Expressions

8.3.1.6 Aufgabe

- Schreiben sie ein neues Programm **`Lesen_Mult_FASTA.pl`**;
- Das Programm soll
  - die Datei multiple FASTA Datei **`Presenilin_hum_prot_mult.fasta`**

```
>gi|4506163|ref|NP_000012.1| presenilin 1 isoform I-467 [Homo sapiens]
MTELPAPLSYFQNAQMSEDNHLSNTVRSQNDNRERQEHNDRRSLGHPEPLSNGRPQGNSRQVVEQDEEED
EELTLKYGAKHVIMLFVPVTLCMVVVVATIKSVSFYTRKDGQLIYTPFTEDTETVGQRALHSILNAAIMI
SVIVVMTILLVVLYKYRCYKVIHAWLIISSLLLLFFFSFIYLGEVFKTYNVAVDYITVALLIWNFGVVGM
ISIHWKGPLRLQQAYLIMISALMALVFIKYLPEWTAWLILAVISVYDLVAVLCPKGPLRMLVETAQERNE
TLFPALIYSSTMVWLVNMAEGDPEAQRRVSKNSKYNAESTERESQDTVAENDDGGFSEEWEAQRDSHLGP
HRSTPESRAAVQELSSSILAGEDPEERGVKLGLGDFIFYSVLVGKASATASGDWNTTIACFVAILIGLCL
TLLLLAIFKKALPALPISITFGLVFYFATDYLVQPFMDQLAFHQFYI
>gi|4506165|ref|NP_000438.1| presenilin 2 isoform 1; Alzheimer's disease
3-like [Homo sapiens]
MLTFMASDSEEEVCDERTSLMSAESPTPRSCQEGRQGPEDGENTAQWRSQENEEDGEEDPDRYVCSGVPG
RPPGLEEELTLKYGAKHVIMLFVPVTLCMIVVVATIKSVRFYTEKNGQLIYTTFTEDTPSVGQRLLNSVL
NTLIMISVIVVMTIFLVVLYKYRCYKFIHGWLIMSSLMLLFLFTYIYLGEVLKTYNVAMDYPTLLLTVWN
FGAVGMVCIHWKGPLVLQQAYLIMISALMALVFIKYLPEWSAWVILGAISVYDLVAVLCPKGPLRMLVET
AQERNEPIFPALIYSSAMVWTVGMAKLDPSSQGALQLPYDPEMEEDSYDSFGEPSYPEVFEPPLTGYPGE
ELEEEEERGVKLGLGDFIFYSVLVGKAAATGSGDWNTTLACFVAILIGLCLTLLLLAVFKKALPALPISI
TFGLIFYFSTDNLVRPFMDTLASHQLYI
```

    aus dem aktuellen Arbeitsverzeichnis einlesen;
  - die oben farbig markierten Inhalte für beide Proteine mithilfe regulärer Ausdrücke extrahieren;
  - und ungefähr wie folgt auf dem Bildschirm ausgeben:

```
Extrahierte Daten aus:  Presenilin_hum_prot_mult.fasta

Protein Description:  presenilin 1 isoform I-467
Organism Source: Homo sapiens
Genbank Identifier: 4506163
Accession Number: NP_000012.1
```

```
MTELPAPLSYFQNAQMSEDNHLSNTVRSQNDNRERQEHNDRRSLGHPEPLSNGRPQGNSRQVVEQDEEED
EELTLKYGAKHVIMLFVPVTLCMVVVVATIKSVSFYTRKDGQLIYTPFTEDTETVGQRALHSILNAAIMI
SVIVVMTILLVVLYKYRCYKVIHAWLIISSLLLLFFFSFIYLGEVFKTYNVAVDYITVALLIWNFGVVGM
ISIHWKGPLRLQQAYLIMISALMALVFIKYLPEWTAWLILAVISVYDLVAVLCPKGPLRMLVETAQERNE
TLFPALIYSSTMVWLVNMAEGDPEAQRRVSKNSKYNAESTERESQDTVAENDDGGFSEEWEAQRDSHLGP
HRSTPESRAAVQELSSSILAGEDPEERGVKLGLGDFIFYSVLVGKASATASGDWNTTIACFVAILIGLCL
TLLLLAIFKKALPALPISITFGLVFYFATDYLVQPFMDQLAFHQFYI


Protein Description:  presenilin 2 isoform 1; Alzheimer's disease 3-
like
Organism Source: Homo sapiens
Genbank Identifier: 4506165
Accession Number: NP_000438.1
MLTFMASDSEEEVCDERTSLMSAESPTPRSCQEGRQGPEDGENTAQWRSQENEEDGEEDPDRYVCSGVPG
RPPGLEEELTLKYGAKHVIMLFVPVTLCMIVVVATIKSVRFYTEKNGQLIYTTFTEDTPSVGQRLLNSVL
NTLIMISVIVVMTIFLVVLYKYRCYKFIHGWLIMSSLMLLFLFTYIYLGEVLKTYNVAMDYPTLLLTVWN
FGAVGMVCIHWKGPLVLQQAYLIMISALMALVFIKYLPEWSAWVILGAISVYDLVAVLCPKGPLRMLVET
AQERNEPIFPALIYSSAMVWTVGMAKLDPSSQGALQLPYDPEMEEDSYDSFGEPSYPEVFEPPLTGYPGE
ELEEEEERGVKLGLGDFIFYSVLVGKAAATGSGDWNTTLACFVAILIGLCLTLLLLAVFKKALPALPISI
TFGLIFYFSTDNLVRPFMDTLASHQLYI


Anzahl der gelesenen Sequenzen: 2
```

8.3.1.7 Lösung 1

Programmtext von **`Lesen_Mult_FASTA.pl`**:

```
#!/usr/bin/perl -w
$Datei_Name = 'Presenilin_hum_prot_mult.fasta';
$Sequenzen_Anzahl = 0;
print "Extrahierte Daten aus:\t$Datei_Name";
open (FILE, "<$Datei_Name");
while ($Zeile = <FILE>) {
    if ($Zeile =~ /^>.*/) {
        $& =~ /^>gi\|([0-9]*)\|.*\|(.*)\|(.*)\s\[(.*)\]/;
        print "\n\nProtein Description: $3\n";
        print "Organism Source: $4\n";
        print "Genbank Identifier: $1\n";
        print "Accession Number: $2\n";
        $Sequenzen_Anzahl++;
    }
    # Speichern der Sequenzzeilen in $1
    elsif ($Zeile =~ /([GAVLIPFYCMHKRWSTDENQBZ\.X]+)/) {
        print $1;
    }
    else {
        print "Warning: unrecognized character found!\n";
    }
}
close FILE;
print "\n\nAnzahl der gelesenen Sequenzen:\t$Sequenzen_Anzahl\n";
exit;
```

Im Block

```
if ($Zeile =~ /^>.*/) {
        $& =~ /^>gi\|([0-9]*)\|.*\|(.*)\|(.*)\s\[(.*)\]/;
        print "\n\nProtein Description: $3\n";
        print "Organism Source: $4\n";
        print "Genbank Identifier: $1\n";
        print "Accession Number: $2\n";
        $Sequenzen_Anzahl++;
    }
```

werden mit dem ersten regulären Ausdruck

```
if ($Zeile =~ /^>.*/)
```

die FASTA Kopfzeilen der Eingabe-Datei abgefangen. Die Variable **`$&`** hat also in den Schleifen-Durchläufen erst den Wert

```
>gi|4506163|ref|NP_000012.1| presenilin 1 isoform I-467 [Homo
sapiens]
```

und dann den Wert

```
>gi|4506165|ref|NP_000438.1| presenilin 2 isoform 1;
Alzheimer's disease 3-like [Homo sapiens]
```

In der Zeile

```
$& =~ /^>gi\|([0-9]*)\|.*\|(.*)\|(.*)\s\[(.*)\]/;
```

Wird auf dem Inhalt von **$&** ein Muster gesucht und über die Variablen **$1...$4** Teile des gefunden Musters gespeichert und auf dem Bildschirm ausgegeben:

```
print "\n\nProtein Description: $3\n";
print "Organism Source: $4\n";
print "Genbank Identifier: $1\n";
print "Accession Number: $2\n";
```

Im **elsif** Block

```
    elsif ($Zeile =~ /([GAVLIPFYCMHKRWSTDENQBZ\.X]+)/) {
        print $1;
    }
```

Werden dann alle EINZELNEN (!) Sequenz-Zeilen in **$1** aufgefangen und direkt auf dem Bildschirm ausgegeben. Beachten sie, dass **$1** nur mit einem neuen Wert reinitialisiert wird.

8.3.1.8 Lösung 2

Eine alternative und elegantere Lösung finden sie im Programmtext von **Lesen_Mult_FASTA_slurp.pl**:

```
#!/usr/bin/perl -w
$Datei_Name = 'Presenilin_hum_prot_mult.fasta';
open (FILE, "<$Datei_Name");
@Datei_Inhalt = <FILE>;
close FILE;
$Inhalt = join ('', @Datei_Inhalt);
@Eintraege = split (/>/, $Inhalt);
shift @Eintraege;
foreach $Wert (@Eintraege) {
    if ($Wert =~ /^gi\|([0-9]*)\|.*\|(.*)\|(.*)\s\[(.*)\]\n/) {
        print "\nProtein Description: $3\n";
        print "Organism Source: $4\n";
        print "Genbank Identifier: $1\n";
        print "Accession Number: $2\n";
        @Sequenz = split (/\n/, $');
        $Sequenz = join ('', @Sequenz);
        print "Raw Sequence:\n$Sequenz\n";
    }
    else {next;}
}
exit;
```

Im Unterschied zu **Lesen_Mult_FASTA.pl** wird in diesem Programm der Datei-Inhalt zuerst in einem Array gespeichert

```
@Datei_Inhalt = <FILE>;
```

und dann der Array in einen String umgewandelt

```
$Inhalt = join ('', @Datei_Inhalt);
```

Der Inhalt **$Inhalt** von ist zu diesem Zeitpunkt

```
>gi|4506163|ref|NP_000012.1| presenilin 1 isoform I-467 [Homo sapiens]
MTELPAPLSYFQNAQMSEDNHLSNTVRSQNDNRERQEHNDRRSLGHPEPLSNGRPQGNSRQVVEQDEEED
```

```
EELTLKYGAKHVIMLFVPVTLCMVVVVATIKSVSFYTRKDGQLIYTPFTEDTETVGQRALHSILNAAIMI
SVIVVMTILLVVLYKYRCYKVIHAWLIISSLLLLFFFSFIYLGEVFKTYNVAVDYITVALLIWNFGVVGM
ISIHWKGPLRLQQAYLIMISALMALVFIKYLPEWTAWLILAVISVYDLVAVLCPKGPLRMLVETAQERNE
TLFPALIYSSTMVWLVNMAEGDPEAQRRVSKNSKYNAESTERESQDTVAENDDGGFSEEWEAQRDSHLGP
HRSTPESRAAVQELSSSILAGEDPEERGVKLGLGDFIFYSVLVGKASATASGDWNTTIACFVAILIGLCL
TLLLLAIFKKALPALPISITFGLVFYFATDYLVQPFMDQLAFHQFYI
>gi|4506165|ref|NP_000438.1| presenilin 2 isoform 1; Alzheimer's
disease 3-like [Homo sapiens]
MLTFMASDSEEEVCDERTSLMSAESPTPRSCQEGRQGPEDGENTAQWRSQENEEDGEEDPDRYVCSGVPG
RPPGLEEELTLKYGAKHVIMLFVPVTLCMIVVVATIKSVRFYTEKNGQLIYTPFTEDTPSVGQRLLNSVL
NTLIMISVIVVMTIFLVVLYKYRCYKFIHGWLIMSSLMLLFLFTYIYLGEVLKTYNVAMDYPTLLLTVWN
FGAVGMVCIHWKGPLVLQQAYLIMISALMALVFIKYLPEWSAWVILGAISVYDLVAVLCPKGPLRMLVET
AQERNEPIFPALIYSSAMVWTVGMAKLDPSSQGALQLPYDPEMEEDSYDSFGEPSYPEVFEPPLTGYPGE
ELEEEEERGVKLGLGDFIFYSVLVGKAAATGSGDWNTTLACFVAILIGLCLTLLLLAVFKKALPALPISI
TFGLIFYFSTDNLVRPFMDTLASHQLYI
```

als EIN kompletter String!

Im nächsten Schritt wird der String auf dem Muster **/>/** aufgeteilt:

```
@Eintraege = split (/>/, $Inhalt);
```

Dadurch entsteht ein Array mit drei Elementen:

(' ', Block 1, Block 2)

Dadurch, dass das Muster für die Trennung direkt beim ersten Zeichen von **$Inhalt** gefunden wird produziert die Funktion ein erstes, leeres Element **' '** in dem Array. Mit dem Befehl

```
shift @Eintraege;
```

wird das erste, unerwünschte Element entfernt.

Die beiden anderen korrespondieren mit den beiden Einträgen. Beachten sie allerdings, dass **split** das Trennungsmuster (in diesem Fall also **>**) entfernt.

Der nachfolgende **if** Block

```
if ($Wert =~ /^gi\|([0-9]*)\|.*\|(.*)\|(.*)\s\[(.*)\]\n/) {
        print "\nProtein Description: $3\n";
        print "Organism Source: $4\n";
        print "Genbank Identifier: $1\n";
        print "Accession Number: $2\n";
        @Sequenz = split (/\n/, $');
        $Sequenz = join ('', @Sequenz);
        print "Raw Sequence:\n$Sequenz\n";
    }
```

ist ebenfalls modifiziert. Wie zuvor wird über ein Muster und runde Klammern auf **$1...$4** zugegriffen und die respektiven Datenelemente extrahiert. Das Muster ist so definiert, dass es genau auf die Kopfzeile passt.

Deswegen kann in Zeile

```
        @Sequenz = split (/\n/, $');
```

über die Standardvariable **$'** auf den Rest der Zeichen in **$Wert** (*rechts* vom Muster) zugegriffen werden.

In der Zeile

```
        $Sequenz = join ('', @Sequenz);
```

wird der Array **@Sequenz**, der aus der **split** Funktion resultiert wieder zusammensetzt. **$Sequenz** enthält nun den kompletten Sequenze-String ohne Zeilenumbrüche **\n** und wird auf dem Bildschirm ausgegeben.

8.3.1.9 Lösung 3

Die zweite alternative Lösung ist im Programmtext von **Lesen_Mult_FASTA_hash.pl** dargestellt:

```
#!/usr/bin/perl -w
$Datei_Name = 'Presenilin_hum_prot_mult.fasta';
open (FILE, "<$Datei_Name");
@Datei_Inhalt = <FILE>;
close FILE;
$Inhalt = join ('', @Datei_Inhalt);
@Eintraege = split (/>/, $Inhalt);
shift @Eintraege;
foreach $Wert (@Eintraege) {
    if ($Wert =~ /^gi\|([0-9]*)\|.*\|(.*)\|\s(.*)\s\[(.*)\]\n/) {
        push (@Sequenz_Daten, $3);
        push (@Sequenz_Daten, $4);
        push (@Sequenz_Daten, $2);
        @Sequenz = split (/\n/, $');
        $Sequenz = join ('', @Sequenz);
        push (@Sequenz_Daten, $Sequenz);
        push (@{$Resultate{$1}}, @Sequenz_Daten);
        @Sequenz_Daten = ();
    }
    else {next;}
}
foreach $Schluessel (keys %Resultate) {
    print "\nGenbank Identifier: $Schluessel\n";
    print "Protein Name: ${$Resultate{$Schluessel}}[0]\n";
    print "Species: ${$Resultate{$Schluessel}}[1]\n";
    print "Accession Number: ${$Resultate{$Schluessel}}[2]\n";
    print "Raw Sequence;\n${$Resultate{$Schluessel}}[3]\n";
    $length = length ${$Resultate{$Schluessel}}[3];
    print "Length: $length AA\n";
}
exit;
```

In diesem Programm sind das Einlesen und die Ausgabe auf dem Bildschirm voneinander logisch getrennt. Die extrahierten Daten werden erst in Variablen und dann in einem Hash abgelegt, auf den ur die Ausgabe zugegriffen wird.

Diese Separierung ist notwendig, wenn in einem Programm noch eine weitere Prozessierung der extrahierten Daten stattfinden soll. Für eine bioinformatische Aufgabe ist deswegen **`Lesen_Mult_FASTA_hash.pl`** die beste Lösung.

Betrachten wir das Programm im Detail. Der Anfang des Programms ist genauso geschrieben wie in **`Lesen_Mult_FASTA_slurp.pl`**. D.h. wir extrahieren die Daten wieder über eine **`foreach`** Schleife

```
foreach $Wert (@Eintraege) {
    if ($Wert =~ /^gi\|([0-9]*)\|.*\|(.*)\|\s(.*)\s\[(.*)\]\n/) {
        push (@Sequenz_Daten, $3);
        push (@Sequenz_Daten, $4);
        push (@Sequenz_Daten, $2);
        @Sequenz = split (/\n/, $');
        $Sequenz = join ('', @Sequenz);
        push (@Sequenz_Daten, $Sequenz);
        push (@{$Resultate{$1}}, @Sequenz_Daten);
        @Sequenz_Daten = ();
}
```

über einen Array mit Werten, wo jeder Wert einem Eintrag der Art

```
gi|4506165|ref|NP_000438.1| presenilin 2 isoform 1; Alzheimer's disease 3-like
[Homo sapiens]
MLTFMASDSEEEVCDERTSLMSAESPTPRSCQEGRQGPEDGENTAQWRSQENEEDGEEDPDRYVCSGVPG
RPPGLEEELTLKYGAKHVIMLFVPVTLCMIVVVATIKSVRFYTEKNGQLIYTTFTEDTPSVGQRLLNSVL
NTLIMISVIVVMTIFLVVLYKYRCYKFIHGWLIMSSLMLLFLFTYIYLGEVLKTYNVAMDYPTLLLTVWN
FGAVGMVCIHWKGPLVLQQAYLIMISALMALVFIKYLPEWSAWVILGAISVYDLVAVLCPKGPLRMLVET
AQERNEPIFPALIYSSAMVWTVGMAKLDPSSQGALQLPYDPEMEEDSYDSFGEPSYPEVFEPPLTGYPGE
ELEEEEERGVKLGLGDFIFYSVLVGKAAATGSGDWNTTLACFVAILIGLCLTLLLLAVFKKALPALPISI
TFGLIFYFSTDNLVRPFMDTLASHQLYI
```

entspricht. Das Vorgehen in der **`foreach`** Schleife ist allerdings als zuvor. Die Werte der Standardvariablen **`$3, $4`** und **`$2`** werden (für jede gefundene FASTA-Kopfzeile) sequentiell

```
    if ($Wert =~ /^gi\|([0-9]*)\|.*\|(.*)\|\s(.*)\s\[(.*)\]\n/) {
        push (@Sequenz_Daten, $3);
```

```
        push (@Sequenz_Daten, $4);
        push (@Sequenz_Daten, $2);
        @Sequenz = split (/\n/, $');
        $Sequenz = join ('', @Sequenz);
        push (@Sequenz_Daten, $Sequenz);
        push (@{$Resultate{$1}}, @Sequenz_Daten);
        @Sequenz_Daten = ();
```

in einen Array geschrieben. Dieser Array **@Sequenz_Daten** enthält nach den drei ersten **push** Ausdrücken folgende Werte:

($3, $4, $2)

was

(Protein Name, Species, Accession Number)

für den aktuell gelesenen Eintrag entspricht. Dann werden wie zuvor die **split** und **join** Funktionen aufgerufen um **$Sequenz** zu generieren. **$Sequenz** wird als nächster Wert an den Array **@Sequenz_Daten** angehängt, der nun also den Inhalt

($3, $4, $2)

oder

(Protein Name, Species, Accession Number, Sequence)

hat.

Wie gesagt, wollen wir später auf alle Daten, die wir für einen Eintrag extrahiert wurden in einer sinnvollen Weise noch (z.B. für die Bildschirm-Ausgabe) zugreifen können.

Der *Genbank Identifier* (die **gi**-Nummer in der FASTA-Kopfzeile) ist eine unveränderlicher, eindeutiger Schlüssel. Deswegen wollen wir einen Hash bilden, in dem über die **gi**-Nummer als Schlüssel alle anderen Daten (Protein-Name, Spezies, Zugriffs-Nummer und die volle Sequenz) zugänglich sind.

Wir müssen also einen Hash bilden der als Schlüssel die **gi**-Nummer besitzt und als assoziierten Wert eine Liste, einen Array.

Mit

```
        push (@{$Resultate{$1}}, @Sequenz_Daten);
```

wird der Array **@Sequenz_Daten** (mit den Daten aus einem Eintrag) also

(Protein Name, Species, Accession Number, Sequence)

als *Wert* in den Hash **%Resultate** für den *Schlüssel* **$1** (also der extrahierten **gi**-Nummer) geschrieben.

Beachten sie, dass der Wert in einem Hash immer ein *skalarer* Wert sein muss. Deswegen wird mit dem Befehl auch nicht der Array **@Sequenz_Daten** selbst sondern eine sogenannte *Referenz* auf ihn als Wert in den Hash geschrieben.

Analog muss in der **foreach** Schleife über aller Schlüssel-Wert-Paare zur Ausgabe auf dem Bildschirm

```
$Schluessel (keys %Resultate) {
    print "\nGenbank Identifier: $Schluessel\n";
    print "Protein Name: ${$Resultate{$Schluessel}}[0]\n";
    print "Species: ${$Resultate{$Schluessel}}[1]\n";
    print "Accession Number: ${$Resultate{$Schluessel}}[2]\n";
    print "Raw Sequence;\n${$Resultate{$Schluessel}}[3]\n";
    $length = length ${$Resultate{$Schluessel}}[3];
    print "Length: $length AA\n";
}
```

durch einen Aufruf des Typs

```
    print "Raw Sequence;\n${$Resultate{$Schluessel}}[3]\n";
```

auf das Element an Index-Position 3 (definiert durch `[3]`) des Arrays dessen Referenz als Wert für den Schlüssel `$Schluessel` im Hash `%Resultate` definiert ist zugegriffen werden.

8.3.2 Sequenzmotive und Mustererkennung

In den letzten Abschnitten habe wir gelernt wie Textdateien mit der Mustererkennungs-Funktionalität von `Perl` untersucht werden können.

Eine zweite, zentrale Anwendung von Musterkennung in der Bioinformatik ist die Analyse von biologischen Sequenzen zum Finden von Sequenzmotiven.

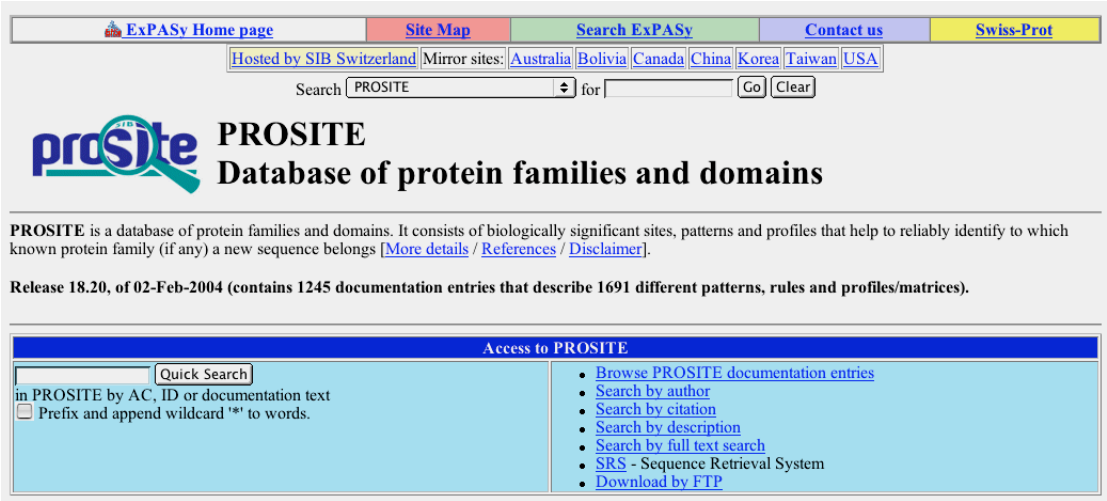
Sequenzmotive, sind evolutionär konservierte Teilbereiche von Protein- und DNA-Sequenzen. Sie haben häufig eindeutige biologische Funktion:

- *Proteinsequenzmotive* stellen häufig funktionelle oder strukturelle Bereiche eine Proteins dar (Katalytische Zentren, Regionen mit einer speziellen 3d-Struktur etc.);
- Bekannte *DNA-Sequenzmotive* sind Bindestellen für Proteine (z.B. Transkriptionsfaktoren), konservierte Bereiche an Exon-Intron-Grenzen, poly-Adenylierungsstellen etc.

Sequenzmotive können auf verschieden Art und Weise beschrieben werden, u.a. als Muster, quasi wie ein regulärer Ausdruck in `Perl`.

8.3.2.1 Signaturen, Motive und die PROSITE Datenbank

PROSITE (http://www.expasy.org/prosite/)



ist eine Datenbank von Proteinfamilien und Domänen. Sie besteht aus biologisch signifikanten Protein-*Teilsequenzen*, *Mustern* und *Profilen*, die dazu dienen können ein neue Proteinsequenz einer bereits bekannten Proteinfamilie zuzuordnen.

Betrachten wir einen beispielhaften PROSITE Datenbank-Eintrag:

```
{PDOC00032}
{PS00032; ANTENNAPEDIA}
{BEGIN}
**************************************************
* 'Homeobox' antennapedia-type protein signature *
**************************************************

The homeotic  Hox  proteins  are sequence-specific transcription factors. They
are part  of  a  developmental  regulatory  system  that  provides  cells with
specific  positional identities on the anterior-posterior (A-P) axis [1].  The
hox proteins  contain  a  'homeobox'  domain. In Drosophila and other insects,
there are eight different Hox genes  that  are  encoded in two gene complexes,
ANT-C and BX-C. In vertebrates there are 38 genes organized in four complexes.

In six of the eight Drosophila Hox genes the homeobox domain is highly similar
and a  conserved hexapeptide  is found five to sixteen amino acids upstream of
the homeobox domain. The six Drosophila proteins that belong to this group are
```

```
antennapedia (Antp),  abdominal-A (abd-A), deformed (Dfd), proboscipedia (pb),
sex combs reduced (scr)  and ultrabithorax (ubx) and are collectively known as
the 'antennapedia' subfamily.

In vertebrates  the  corresponding  Hox genes are known [2] as Hox-A2, A3, A4,
A5, A6,  A7,  Hox-B1,  B2, B3, B4, B5, B6, B7, B8, Hox-C4, C5, C6, C8, Hox-D1,
D3, D4 and D8.

Caenorhabditis elegans lin-39 and mab-5 are also members of the 'antennapedia'
subfamily.

As a signature pattern  for this subfamily of  homeobox proteins, we have used
the conserved hexapeptide.

-Consensus pattern: [LIVMFE]-[FY]-P-W-M-[KRQTA]
-Sequences known to belong to this class detected by the pattern: ALL,  except
 for 6 sequences.
-Other sequence(s) detected in Swiss-Prot: 3.

-Note: Arg  and  Lys  are  most  frequently  found in the last position of the
 hexapeptide; other amino acids are found in only a few cases.

-Last update: June 1994 / Text revised.

[ 1] McGinnis W., Krumlauf R.
     Cell 68:283-302(1992).
[ 2] Scott M.P.
     Cell 71:551-553(1992).

+----------------------------------------------------------------------------+
| This PROSITE entry  is copyright  by the Swiss Institute of Bioinformatics |
| (SIB).  There are no restrictions on its use by non-profit institutions as |
| long as its  content  is  in  no  way  modified  and this statement is not |
| removed. Usage by and for commercial entities requires a license agreement |
| (See http://www.isb-sib.ch/announce/ or email to license@isb-sib.ch).      |
+----------------------------------------------------------------------------+
{END}
```

Unter **`Consensus pattern`** finden wir eine Musterdefinition für ein Proteinsequenzmotiv:

`[LIVMFE]-[FY]-P-W-M-[KRQTA]`

Ganz ähnlich einem regulären Ausdruck in **`Perl`** dient dieses *Konsensus-Muster* dazu eine Zeichenkette abstrakt zu beschreiben. In diesem Fall lässt sich das Muster folgendermaßen in ein abstraktes *Hexapeptid* (sechs Aminosäuren) übersetzen:

> Aminosäure **`L, I, V, M, F`** oder **`E`** gefolgt von Aminosäure **`F`** oder **`Y`** gefolgt von Aminosäure **`P`**, gefolgt von Aminosäure **`W`**, gefolgt von Aminosäure **`M`**, gefolgt von Aminosäure **`K, R, Q, T`** oder **`A`**.

Der gleichwertige reguläre Ausdruck in **`Perl`** wäre also:

`[LIVMFE][FY]PWM[KRQTA]`

Anhand des Musters kann also die Sequenz verschiedener Proteine abgesucht werden, um zu bestimmen, ob dieses das Muster enthalten und deswegen in eine bekannte, funktionale Proteinklasse gruppiert werden können.

Mehr zur Musterdefinition in der PROSITE Datenbank finden sie z.B. unter:

http://www.cbc.umn.edu/gst/prosite.html

8.3.2.2 Aufgabe

- Schreiben sie ein neues Programm **`Prosite_scan.pl`**;
- Das Programm soll die
  - Die *Consensus Patterns* aus den drei Textdateien / PROSITE-Einträgen
    - **`PDOC00032`**

- **PDOC00034**
- **PDOC00033**

im aktuellen Arbeitsverzeichnis lesen;

- Die PROSITE *Consensus Patterns* in reguläre Ausdrücke / Muster in der **Perl** Notation übersetzen;
- Diese Muster auf der Sequenz **Protein_seq.pl** im aktuellen Arbeitsverzeichnis suchen;
- Eine Ausgabe auf dem Bildschirm wie folgt machen:

```
Target Sequenz: Protein_Seq.fasta
MTELPAPLSYFQNAQMSEDNHLSNTVRSQNDNRLFPWMKERQEHNDRRSLGHPEPLSNGRPQGNSRQVVEQDE
EEDEELTLKYGAKHVIMLFVPVTLCMVVVVATIKSVSFYTRKDGQLIYTPFTEDTETVGQRALHSILNAAIMI
SVIVVMTILLVVLYKYRCYKVIHAWLIISSLLLLFFFSFIYLGEVFKTYNVAVDYITVALLIWNFGVVGMLMA
QGLYNISIHWKGPLRLQQAYLIMISALMALVFIKYLPEWTAWLILAVISVYDLVAVLCPKGPLRMLVETAQER
NETLFPALIYSSTMVWLVNMAEGDPEAQRRVSKNSKYNAESTERESQDTVAENDDGGFSEEWEAQRDSHLGPH
RSTPESRRPCAAVQELSSSILCVSAGEDPEERGVKLGLGDFIFYSVLVGKASATASGDWNTTIACFVAILIGL
CLTLLLLAIFKKALPALPISITFGLVFYFATDYLVQPFMDQLAFHQFYIIYPWMQLFPWMQ

Signatur von PDOC00034.pst:     RPC[GAVLIPFYCMHKRWSTDENQBZ.X]{11}CVS
Hits auf Protein_Seq.fasta:     RPCAAVQELSSSILCVS

Signatur von PDOC00033.pst:     LMA[EQ]GLYN
Hits auf Protein_Seq.fasta:     LMAQGLYN

Signatur von PDOC00032.pst:     [LIVMFE][FY]PWM[KRQTA]
Hits auf Protein_Seq.fasta:     LFPWMK IYPWMQ LFPWMQ
```

8.3.2.3 Lösungshinweise

- Schreiben sie je eine Subroutine, die
  - die Protein Sequenz von **Protein_seq.pl** einliest und die Rohsequenz als skalaren Wert an das Hauptprogramm zurückgibt. Aufruf mit dem Datei-Namen;
  - die alle PROSITE Dokumente öffnet, den Inhalt der Dateien liest, das *Consensus Pattern* in einen regulären Ausdruck nach **Perl** Notation umwandelt (studieren sie dazu die Textdateien **PDOC00032, PDOC00034** und **PDOC00033**) und als Rückgabewert einen Hash des Typs

    ('PDOC00032.pst' => '[LIVMFE][FY]PWM[KRQTA]')

    bildet. Aufruf mit einem Array aller Namen der PROSITE Dokumente;
  - die auf der Zielsequenz nach den Mustern sucht. Aufruf in einer **foreach** Schleife über alle Muster (Schlüssel des Hash / Rückgabewert aus der zweiten Subroutine) mit der Zielsequenz (Rückgabewert der ersten Subroutine) und dem jeweiligen Muster (Werte in dem Hash aus der zweiten Subroutine) als übergebene Parameter.

8.3.2.4 Lösung

Programmtext von **Prosite_scan.pl**:

```
#!/usr/bin/perl -w

$Target_Sequenz = parse_target_seq ($ARGV[0]);
print "Target Sequenz:\t$ARGV[0]\n$Target_Sequenz\n\n";
@Prosite_Signaturen = ('PDOC00032.pst', 'PDOC00033.pst', 'PDOC00034.pst');
%Muster_Hash = parse_prosite_docs (@Prosite_Signaturen);
foreach $Schluessel (keys %Muster_Hash) {
    print "Signatur von $Schluessel:\t$Muster_Hash{$Schluessel}\n";
    @Hits = find_signature ($Target_Sequenz, $Muster_Hash{$Schluessel});
    print "Hits auf $ARGV[0]:\t@Hits\n\n";
```

```
}
exit;

# Subroutine zum Finden der Signaturen
sub find_signature {
    my ($target_sequenz, $muster) = @_;
    my @hits = ();
    while ($target_sequenz =~ /$muster/g) {
        push (@hits, $&);
        }
    return @hits;
}

# Subroutine zum Lesen der PROSITE Dateien und
# Uebersetzen der PROSITE Muster in Perl RegEx
sub parse_prosite_docs {
    my %muster_hash = ();
    foreach my $datei_name (@_) {
        open (FILE, $datei_name);
        my @datei_inhalt = <FILE>;
        close FILE;
        foreach $zeile (@datei_inhalt) {
            if ($zeile =~ /^-Consensus pattern: (.*)\n/) {
                my $signatur = $1;
                # Bindestriche in PROSITE Muster durch "" ersetzen
                $signatur =~ s/-//g;
                # "X" in PROSITE Muster ersetzen
                $signatur =~ s/x/[GAVLIPFYCMHKRWSTDENQBZ\.X]/ig;
                # Runde Klammern in PROSITE Muster ersetzen
                $signatur =~ tr/()/{}/;
                $muster_hash{$datei_name} = $signatur;
            }
            else {next;}
        }

    }
    return %muster_hash;
}

# Subroutine zum Lesen der FASTA Datei
sub parse_target_seq {
    my $sequenz = '';
    open (FILE, $_[0]);
    my @datei_inhalt = <FILE>;
    close FILE;
    foreach my $zeile (@datei_inhalt) {
        if ($zeile =~ /^>/) {
            next;
        }
        else {
            substr ($zeile, -1) = "";
            $sequenz .= $zeile;
        }
    }
    return $sequenz;
}
```