6 Block 6: Wiederholung & Zusammenfassung der Blöcke 1-5

6.1 Lernziele

Ziel dieses Blocks ist es die Inhalte der vorangegangene Blöcke 1-5 zu wiederholen und die wichtigsten, bisher behandelten programmatischen und syntaktischen Elemente von **`Perl`** zusammenzufassen.

Zusätzlich werden die

- **`for`** Schleife;
- **`unless`** Bedingung und die
- **`exists`** Funktion

neu eingeführt.

6.2 Theorie

6.2.1 Block 1

- *Nukleotidsequenzen* und *Aminosäuresequenzen* werden durch *Zeichenalphabete* als *Zeichenketten* repräsentiert.
- In der Bioinformatik und Programmierung werden diese Daten hauptsächlich mithilfe des *Datentyps String* verarbeitet.
- Der **`Perl`** Interpreter führt Programme zeilenweise aus.
- Die *Kommandozeilen-Interpretation*, z.B.

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

  erlaubt es **`Perl`** Programme direkt von der Befehlszeile des Betriebssystems aufzurufen.

- Kommentare

```
# Speichern einer Sequenz in einer Variablen
```

  werden nicht interpretiert und dienen

  (1) zur *Erläuterung des Programmtextes* und

  (2) werden durch *Auskommentieren* beim Erstellen eines Programms verwendet um gezielt Zeilen oder Blöcke für den **`Perl`** Interpreter zu blockieren und so zu testen.

- Variablendeklaration

```
$DNA_Sequenz # Variablendeklaration
```

  Wertzuweisung

```
$DNA_Sequenz = 'ACGGTGCTGGACTGATTAGCGTATATAGC'; # Wertzuweisung
```

6.2.1.1 Datenausgabe auf dem Bildschirm

- Mit dem **`print`** Befehl

```
print $DNA_Sequenz;
```

  werden Ausgaben auf die *Standard-Ausgabe* (Voreinstellung: Bildschirm) gemacht.

- Mit dem **`exit`** Befehl

```
exit;
```

  wird der Ablauf eines **`Perl`**-Programms beendet.

6.2.2 Block 2

- Viele Computer-Anwendungen / - Programme in Perl decken folgende typischen Schritte der Datenverarbeitung ab:
    - Daten einlesen;
    - Daten prozessieren / analysieren;
    - Daten / Resultate ausgeben.

6.2.2.1 Dateneingabe über die Kommandozeile (Standard-Eingabe)

```
$DNA_Sequenz = <STDIN>;
```

- Entfernen des Zeilenvorschubs **\n** mit dem **chomp** Befehl

```
chomp $DNA_Sequenz;
```

6.2.2.2 Datenausgabe in eine Datei

- Datei öffnen (zum Überschreiben des Dateiinhalts)

```
open (FILE, ">$Aus_Datei_Name");
```

- Datei öffnen (zum Anhängen an den bestehenden Dateiinhalt)

```
open (FILE, ">>$Aus_Datei_Name");
```

- Umleitung der Ausgabe des **print** Befehls in Datei

```
print FILE $Datei_Inhalt;
```

- Datei schliessen

```
close FILE;
```

6.2.2.3 Dateneingabe aus / Lesen aus einer Datei

- Datei öffnen (zum Lesen des Dateiinhalts)

```
open (FILE, "<$Datei_Name");
```

- Zeilenweises Lesen des Dateiinhalts in eine Variable des Typs *Array*

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

- Ausgabe des Inhalts einer *Array* Variablen auf dem Bildschirm

```
print @Datei_Inhalt;
```

- Programmaufruf

```
perl Res_In_Dat_Args.pl [Dateiname Sequenz] [Dateiname Enzyme]
```

- und Wertzuweisung an Variablen über Kommandozeilen-Argumente

```
$Seq_Datei_Name = $ARGV[0];
```

6.2.3 Block 3

- Einfache Datenprozessierung und –analyse in der Bioinformatik involviert die Manipulation von Daten des Typs *String*.

6.2.3.1 Konkatenierung

```
$Konkat_Sequenz = $DNA_Fragment_1.$DNA_Fragment_2;
```

6.2.3.2 Numerische Operationen

- Addition

```
$Laengen_Summe = $Fragment_1_Laenge+$Fragment_2_Laenge;
```

- Subtraktion, Multiplikation und Division

```
$Laengen_Differenz = $Fragment_1_Laenge-$Fragment_2_Laenge;
$Laengen_Produkt = $Fragment_1_Laenge*$Fragment_2_Laenge;
$Laengen_Quotient = $Fragment_1_Laenge/$Fragment_2_Laenge;
```

- Schachtelung

```
$Wert_1 = ($Fragment_1_Laenge-$Fragment_2_Laenge)*3;
```

6.2.3.3 Die `int` Funktion

- ... zum Umwandeln von Gleitkomma- in Ganzzahl-Darstellung

```
$x = int ((100/$y)*$z);
```

6.2.3.4 Die `substr` Funktion

- ... zum Zugriff auf Einzelzeichen aus einer zusammengesetzten Zeichenkette

```
substr ($x, $Anfang, $Schrittweite);
```

6.2.3.5 Suchen und Ersetzen

```
$y =~ s/i/I/g;
```

- Binde-Operator **=~**
- Ersetzen-Operator **s///**

6.2.3.6 Der `reverse` Befehl zum Umkehren von String-Variablen

```
$x_rev = reverse $x;
```

6.2.3.7 Suchen und Ersetzen

```
$x_revkom =~ tr/ACGTacgt/TGCAtgca/;
```

- Binde-Operator **=~**

- Translatierungs-Operator **`tr///`**

6.2.4 Block 4

- Neben Daten des Typs *string* (einzelne Zeichen oder Zeichenketten) werden in **`Perl`** auch Daten als *Listen* verarbeitet.
- Perl unterscheidet *Arrays* (einfache Listen) und *Hashes* (assoziative Listen).

6.2.4.1 Arrays

6.2.4.1.1 Deklarierung *eines Arrays*

```
@Sequenz = ();
```

6.2.4.1.2 Initialisierung eines Arrays mit einer Liste von Werten

```
@Sequenz = ('A', 'G', 'G', 'T', 'G', 'C', 'A');
```

6.2.4.1.3 Zugriff auf Array-Elemente über Indices.

- Das erste Element

```
print $Sequenz[0];
```

- Das zweite Element

```
print $Sequenz[1];
```

- Das letzte Element

```
print $Sequenz[$#Sequenz];
```

6.2.4.1.4 Ausgabe des Arrays

```
print @Sequenz;
```

```
print "@Sequenz";
```

6.2.4.1.5 Essentielle Befehle / Funktionen für die Verwendung von Arrays

- Die **`pop`** Funktion

```
$Position_Ende = pop @Sequenz;
```

- Die **`shift`** Funktion

```
$Position_1 = shift @Sequenz;
```

- Die **`push`** Funktion

```
push (@Sequenz, $Neues_Element);
```

- Die **`unshift`** Funktion

```
unshift (@Sequenz, $Neues_Element);
```

- Die **`reverse`** Funktion

```
@Reverse_Sequenz = reverse @Sequenz;
```

- Die **scalar** Funktion

```
$Sequenz_Laenge = scalar @Sequenz;
```

- Die **split** Funktion

```
@Sequenz_Array = split(//, $Sequenz_String);
```

- Die **join** Funktion

```
$Sequenz_String = join ('', @Sequenz_Array);
```

6.2.4.2 Hashes (Assoziative Listen)

6.2.4.2.1 Deklarierung eines Hashes

```
%Enzyme = ();
```

6.2.4.2.2 Initialisierung eines Hashes (Definition der Schlüssel-Wert-Paare)

```
%Enzyme = (
        'Acc16I'   => 'TGCGCA',
        'BamHI'    => 'GGATCC',
        'BciVI'    => 'GTATCC',
        'BmrI'     => 'ACTGGG',);
```

oder

```
%Enzyme = (
        'Acc16I', 'TGCGCA',
        'BamHI',  'GGATCC',
        'BciVI',  'GTATCC',
        'BmrI',   'ACTGGG',);
```

6.2.4.2.3 Essentielle Befehle / Funktionen für die Verwendung von Hashes

- *Zugriff auf einen Wert* über den assoziierten Schlüssel:

```
$Erkennungssequenz_Acc16I = $Enzyme{Acc16I};
```

- *Überschreiben eines Wertes* für einen bereits definierten Schlüssel:

```
$Enzyme{Acc16I} = 'Keine gueltige Erkennungssequenz!';
```

- *Definieren eines neuen Schlüssel-Wert-Paares* für einen bereits bestehenden Hash

```
$Erkennungssequenz_ PvuII = 'CAGCTG';
```

- Die **keys** Funktion liefert einen *Array aller Schlüssel* aus einem Hash

```
@Schluessel = keys %Enzyme;
```

- Die **values** Funktion liefert einen *Array aller Werte* aus einem Hash

```
@Werte = values %Enzyme;
```

6.2.4.2.4 Prüfen der Werte für einen Schlüssel in einem Hash

```
%Enzyme = (
        'Acc16I'   => 'TGCGCA',
        'BamHI'    => 'GGATCC',
        'BciVI'    => 'GTATCC',
        'BmrI'     => 'ACTGGG',);
$Auswahl = 'BamHI';
if (exists ($Enzyme{$Auswahl})) {
        print "Die Erkennungssequenz fuer $Auswahl ist $Enzyme{BamHI}\n";
}
else {
        print "Kein Wert vorhanden fuer $Auswahl.";
        exit;
}
```

6.2.5 Block 5

- *Schleifen* und *Bedingungen* sind **`Perl`** Elemente zur Kontrolle des Programmflusses.

6.2.5.1 Bedingungen

- *Bedingungen* kontrollieren die *bedingte* (gleich: *an Bedingungen geknüpfte*) Ausführung von Teilen eines Programms:
    - *Wenn* Bedingung X erfüllt, *dann* führe Programmteil Y aus. *Wenn* Bedingung X nicht erfüllt, *dann* führe Programmteil Y nicht aus – Überspringen!

6.2.5.1.1 Die **`if...else`** *Bedingung*

- Formulierung der Bedingung

```
if ($Abfrage eq 'Acc16I') {...}
```

- Numerische Operatoren und Bedingungen

```
if ($Zahl == 0) {...} # "true" wenn der Wert von $Zahl gleich 0.
if ($Zahl != 1) {...} # "true" wenn der Wert von $Zahl ungleich 0.
if ($Zahl > 0) {...} # "true" wenn der Wert von $Zahl groeser 0.
if ($Zahl < 0) {...} # "true" wenn der Wert von $Zahl kleiner 0.
```

- Beispiel für einen **`if..else`** Block

```
if ($Abfrage eq 'Acc16I') {
    print "Die Erkennungssequenz fuer $Abfrage ist $Enzyme{Acc16I}\n";
}
else {
    print "Keine Daten vorhanden fuer Enzym $Abfrage.\n";
exit;
}
```

6.2.5.1.2 Die **`if...elsif...else`** *Bedingung*

- Beispiel für einen **`if..elsif...else`** Block

```
if ($Abfrage eq 'Acc16I') {
       print "Die Erkennungssequenz fuer Acc16I ist $Enzyme{Acc16I}\n";
}
elsif ($Abfrage eq 'BamHI') {
       print "Die Erkennungssequenz fuer BamHI ist $Enzyme{BamHI}\n";
}
elsif ($Abfrage eq 'BciVI') {
       print "Die Erkennungssequenz fuer BciVI ist $Enzyme{BciVI}\n";
}
elsif ($Abfrage eq 'BmrI') {
       print "Die Erkennungssequenz fuer BmrI ist $Enzyme{BmrI}\n";
}
else {
       print "Kein Wert vorhanden fuer Enzym $Auswahl.\n";
       exit;
}
```

6.2.5.2 Schleifen

- *Schleifen* wiederholen die Ausführung eines bestimmten Programmteils, solange bis eine definierte Bedingung nicht mehr erfüllt ist also ein Test den Wert **`false`** produziert.:
    - *Solange* Bedingung X erfüllt (der assoziierte Test den Wert **`true`** produziert), *wiederhole* Programmteil Y. *Wenn* Bedingung X nicht mehr erfüllt ist (der assoziierte Test den Wert **`false`** produziert) beende den Programmteil – die *Schleife*.

6.2.5.2.1 *Schleifen des Typs* `while`

- Beispiel für eine **`while`** Schleife

```
while ($Zeile = <FILE>) {
    print $Zeile;
}
```

- Zeilenweises Lesen und kontrollierte Datenextraktion aus einer Textdatei in einer **`while`** Schleife mit *regulären Ausdrücken*

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

- Der **`next`** Befehl bewirkt einen Sprung an das Ende des aktuellen Schleifendurchlaufs.

6.2.5.2.2 *Schleifen des Typs* `foreach`

- Die **`foreach`** Schleife wird verwendet, um eine Schleife über jeden *skalaren* Einzelwert in einem Array zu bilden

```
foreach $Zeile (@Datei_Inhalt) {
    print "Zeile:\t$Zeile";
}
```

- Die **`foreach`** Schleife kann auch verwendet werden, um eine Schleife über jedes *Schlüssel-Wert-Paar* in einem Hash zu bilden

```
foreach $Schluessel (keys %Hash) {
    print "$Schluessel\t$Hash{$Schluessel}\n";
}
```

6.2.5.2.3 *Schleifen des Typs* `do...until`

- In einer **`do...until`** Schleife wird ein Programmblock ausgeführt und anschliessend eine Bedingung gestestet. Wenn die Bedingung erfüllt ist, wird der Block nochmals ausgeführt usw.

```
do {
    print "Sequenz fuer das Oligonukleotid eingeben: ";
    $Oligo_Sequenz = <STDIN>;
    chomp $Oligo_Sequenz;
    print "Name fuer das Oligonukleotid eingeben: ";
    $Oligo_Name = <STDIN>;
    chomp $Oligo_Name;
    push (@Alle_Oligos, $Oligo_Name);
    push (@Alle_Oligos, $Oligo_Sequenz);
    print "Die Sequenz von $Oligo_Name ist: $Oligo_Sequenz\n";
    print "Ein weiteres Oligonukleotid eingeben (ja / nein)? ";
    $Auswahl = <STDIN>;
} until ($Auswahl =~ /nein/);
```

6.3 Praxis / Ergänzung

6.3.1 Ergänzung: Schleifen des Typs *for*

Schleifen des Typs **for** werden eingesetzt um mithilfe einer *Schleifen-Variable* oder *Iterations-Variable* den wiederholten Ablauf eines Programmblocks zu kontrollieren.

Betrachten wir folgendes, einfache Programm **For_Schleife_Zaehlen.pl** (**for** Schleifen sind gelb hervorgehoben):

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

# Programm-Name: For_Schleife_Zaehlen.pl
# Autor: picker
# Datum: 01/04
# Aufruf: perl For_Schleife_Zaehlen.pl

$Wert = 10;
# Schleifenvariable
$i = 0;
# Variable, die die Anzahl der Schleifendurchlaeufe zaehlt
$Schleifen_Zaehler = 1;
# Schleife, die von 0 ausgehend ein Inkrement bis $Wert bildet
print "Inkrement-Schleife:\n";
print "--------------------\n";
for ($i = 0; $i<= $Wert; $i++) {
    print "Wert der Schleifenvariablen im ".$Schleifen_Zaehler."ten Durchlauf: $i\n";
    $Schleifen_Zaehler++;
}
$Schleifen_Zaehler = 1;
# Schleife, die von $Wert ausgehend ein Dekrement  bis 0 bildet
print "\nDekrement-Schleife:\n";
print "--------------------\n";
for ($i = $Wert; $i>= 0; $i--) {
    print "Wert der Schleifenvariablen im ".$Schleifen_Zaehler."ten Durchlauf: $i\n";
    $Schleifen_Zaehler++;
}
exit;
```

Die **for** Schleifen verwenden drei durch Semikolon separierte Ausdrücke. Im ersten Fall **($i = 0; $i<= $Wert; $i++)**.

Der erste Ausdruck **$i = 0** dient zur *Schleifen-Initialisierung* und besagt, dass die Schleife unter der Bedingung, dass die *Schleifenvariable* **$i** den Wert 0 hat, das erste mal ausgeführt werden soll.

Der zweite Ausdruck **$i<= $Wert** funktioniert als *Bedingung* und besagt, dass die Schleife, solange wie die *Schleifenvariable* **$i** den Wert kleiner oder gleich dem Wert der Variablen **$Wert** hat, ausgeführt werden soll.

Der dritte Ausdruck **$i++** *reinitialisiert* die Schleifen-Variable **$i** mit jedem Schleifendurchlauf mit einem Wert der um den numerischen Wert **1** größer ist als im vorangegangenen Durchlauf: mit jedem Durchlauf wird also der Wert von **$i** um **1** vergrößert.

Die erste Schleife

```
for ($i = 0; $i<= $Wert; $i++) {
    print "Wert der Schleifenvariablen im ".$Schleifen_Zaehler."ten Durchlauf: $i\n";
    $Schleifen_Zaehler++;
}
```

wird also

- das erste mal ausgeführt wenn **$i = 0**
- solange aus ausgeführt wie gilt **$i <= $Wert**

Die zweite Schleife

```
for ($i = $Wert; $i>= 0; $i--) {
```

```
    print "Wert der Schleifenvariablen im ".$Schleifen_Zaehler."ten Durchlauf: $i\n";
    $Schleifen_Zaehler++;
}
```

wird analog

- das erste mal ausgeführt wenn **`$i = Wert`**
- solange aus ausgeführt wie gilt **`$i >= 0`**

Die erste Schleife verwendet den *Inkrement-Operator* **`++`** um **`$i`** *inkrementell* um **`1`** zu erhöhen. Die zweite Schleife verwendet den *Dekrement-Operator* **`--`** um **`$i`** *dekrementell* um **`1`** zu reduzieren.

Im folgenden Beispiel **`Nukleotid_Bausteine_Zaehlen.pl`** verwenden wir eine **`for`** Schleife um die Häufigkeiten der vier Nukleotid-Bausteine **`A`** (Adenin), **`C`** (Cytosin), **`G`** (Guanin) und **`T`** (Thymin) in einer DNA-Sequenz zu bestimmen:

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

# Programm-Name: Nukleotid_Bausteine_Zaehlen.pl
# Autor: picker
# Datum: 01/04
# Aufruf: perl Nukleotid_Bausteine_Zaehlen.pl

# Die zu untersuchende Sequenz
$DNA_Sequenz = 'NAATGCGXTCGTAZCG';
# mit der Laenge
$DNA_Laenge = length $DNA_Sequenz;
# Initialisierung der Variablen fuer die Haeuigkeiten
$Anzahl_A = 0; # fuer Adenin
$Anzahl_C = 0; # fuer Cytosin
$Anzahl_G = 0; # fuer Guanin
$Anzahl_T = 0; # fuer Thymin
$Unbekannte = 0; # fuer falsche Zeichen
# Initialisierung der Schleifenvaiable
$Position = 0;
# Initialisierung des Indexwertes
$Basen_Position = 0;
for ($Position = 0 ; $Position < $DNA_Laenge ; $Position++) {
    # Inkrementelle Erhoehng des Indexwertes
    $Basen_Position++;
    # Extraktion des aktuell betrachteten Nukleotids
    $Basenpaar = substr($DNA_Sequenz, $Position, 1);
    if ($Basenpaar eq 'A') {
        ++$Anzahl_A;
    }
    elsif ($Basenpaar eq 'C') {
        ++$Anzahl_C;
    }
    elsif ($Basenpaar eq 'G') {
        ++$Anzahl_G;
    }
    elsif ($Basenpaar eq 'T') {
        ++$Anzahl_T;
    }
    else {
        print "Warnung: Unbekanntes Zeichen an Position $Basen_Position: $Basenpaar\n";
        ++$Unbekannte;
    }
}
# Ausgabe auf dem Bildschirm
print "\nUntersuchte Sequenz:\n$DNA_Sequenz\n\n";
print "Anzahl A = $Anzahl_A\n";
print "Anzahl C = $Anzahl_C\n";
print "Anzahl G = $Anzahl_G\n";
print "Anzahl T = $Anzahl_T\n";
print "Anzahl Unbekannte Zeichen  = $Unbekannte\n";
exit;
```

Betrachten wir die **for** Schleife

```
for ($Position = 0 ; $Position < $DNA_Laenge ; $Position++) {
    # Inkrementelle Erhoehng des Indexwertes
    $Basen_Position++;
    # Extraktion des aktuell betrachteten Nukleotids
    $Basenpaar = substr($DNA_Sequenz, $Position, 1);
    if ($Basenpaar eq 'A') {
        ++$Anzahl_A;
    }
    elsif ($Basenpaar eq 'C') {
        ++$Anzahl_C;
    }
    elsif ($Basenpaar eq 'G') {
        ++$Anzahl_G;
    }
    elsif ($Basenpaar eq 'T') {
        ++$Anzahl_T;
    }
    else {
        print "Warnung: Unbekanntes Zeichen an Position $Basen_Position: $Basenpaar\n";
        ++$Unbekannte;
    }
}
```

genauer.

Die Schleife wir begonnen,

```
for ($Position = 0 ; $Position < $DNA_Laenge ; $Position++) {
```

wenn die Schleifenvariable **$Position = 0** ist. Da **$Position** vorher mit dem Wert **0** initialisiert wurde, ist diese Bedingung bei jedem Programmaufruf erfüllt.

Bei jedem Schleifendurchgang wird der Wert von **$Position** um **1** erhöht:

```
for ($Position = 0 ; $Position < $DNA_Laenge ; $Position++) {
```

Die Schleife wird solange ausgeführt

```
for ($Position = 0 ; $Position < $DNA_Laenge ; $Position++) {
```

wie der, - sich inkrementell um **1** erhöhende -, Wert von **$Position** kleiner als die Länge der Sequenz (gespeichert in **$DNA_Laenge**) ist (im Beispiel: der Wert **16**).

Beim jedem Schleifendurchgang wird mit

```
    $Basen_Position++;
```

der aktuelle Indexwert für die Positionsangabe (die erste Position hat konventionsgemäß den Index **1** und nicht **0**) um den Wert **1** erhöht, damit eine Warnung für unbekannte Zeichen

```
else {
        print "Warnung: Unbekanntes Zeichen an Position $Basen_Position: $Basenpaar\n";
        ++$Unbekannte;
    }
```

ausgegeben werden kann.

Der String für das jeweils aktuell betrachtete, einzelne Nukleotid wird in der Variablen **$Basenpaar** gespeichert. Der Wert wird über die **substr** Funktion

```
$Basenpaar = substr($DNA_Sequenz, $Position, 1);
```

ermittelt. Im ersten Schleifendurchlauf ist **$Position = 0**, deswegen wird **$Basenpaar** der Wert des ersten Nukleotids (Einzelzeichens aus **$DNA_Sequenz**)zugewiesen. Im nächsten Durchlauf ist **$Position = 1**, deswegen wird **$Basenpaar** der Wert des zweiten Nukleotids zugewiesen usw.

Dann folgt ein **`if...elsif...else`** Block,

```
if ($Basenpaar eq 'A') {
        ++$Anzahl_A;
    }
    elsif ($Basenpaar eq 'C') {
        ++$Anzahl_C;
    }
    elsif ($Basenpaar eq 'G') {
        ++$Anzahl_G;
    }
    elsif ($Basenpaar eq 'T') {
        ++$Anzahl_T;
    }
    else {
        print "Warnung: Unbekanntes Zeichen an Position $Basen_Position: $Basenpaar\n";
        ++$Unbekannte;
    }
}
```

in dem die Nukleotide unterschieden werden und die Anzahl der jeweils erkannten Positionen über die inkrementelle Erhöhung der Variablen **`$Anzahl_A`** , **`$Anzahl_C`** usw. gespeichert werden. Zeichen, die nicht **`A`**, **`C`**, **`G`** oder **`T`** sind werden in **`$Unbekannte`** erfasst. Anschließend wird das Ergebnis auf dem Bildschirm ausgegeben.

6.3.1.1 Übung

6.3.1.1.1 Aufgabe

- Führen sie das Programm **`Nukleotid_Bausteine_Zaehlen.pl`** auf ihrem Computer aus. Beobachten sie die Ausgabe bei verschiedenen Eingaben und studieren sie den Programmtext;
- Schreiben sie ein neues Programm **`AA_Freq_3Char.pl`**, das
    a. in einer Peptidsequenz die Häufigkeiten, der Aminosäuren (und unbekannter Zeichen) bestimmen soll und
    b. die *Einzelzeichen-Notation* in die *3-Zeichen-Notation* (siehe Tabelle unten) für Peptidsequenzen umwandelt.

6.3.1.1.2 Lösungshinweise

- Die Peptidsequenz **`MK7TRQQQQQQQQQQQQQQNKDSMS3MR`** soll fest als Zeichenkette durch eine Zuweisung an eine Variable im Programm definiert werden.
- Initialisieren Sie einen Hash **`%AA_Freq_Hash`** mit Schlüssel-Wert-Paaren (definieren sie nur die Schlüssel / Aminosäuren, die in der oben stehenden Peptidsequenz tatsächlich auftreten), zur Speicherung der endgültigen Aminosäure-Häufigkeiten in der Art:

```
'M' => 0,
'T' => 0,
.
.
.
'Unbekannte' => 0,
```

- Initialisieren Sie einen Hash **`%AA_Code_Hash`** mit Schlüssel-Wert-Paaren zur Speicherung der 3-Zeichen-Notation (nur für die in der oben stehenden Peptidsequenz auftretenden Aminosäuren) in der Art:

```
'M' => 'Met',
'T' => 'Thr',
.
.
.
```

```
'Unbekannte' => '???',
```

- Verwenden Sie die Nomenklaturdaten aus folgender Tabelle:

Name	Abk.	Linearisierte Strukturformel
Alanin	ala A	CH3-CH(NH2)-COOH
Arginin	arg R	HN=C(NH2)-NH-(CH2)3-CH(NH2)-COOH
Asparagin	asn N	H2N-CO-CH2-CH(NH2)-COOH
Asparaginsäure	asp D	HOOC-CH2-CH(NH2)-COOH
Cystein	cys C	HS-CH2-CH(NH2)-COOH
Glutamin	gln Q	H2N-CO-(CH2)2-CH(NH2)-COOH
Glutaminsäure	glu E	HOOC-(CH2)2-CH(NH2)-COOH
Glycin	gly G	NH2-CH2-COOH
Histidin	his H	NH-CH=N-CH=C-CH2-CH(NH2)-COOH
Isoleucin	ile I	CH3-CH2-CH(CH3)-CH(NH2)-COOH
Leucin	leu L	(CH3)2-CH-CH2-CH(NH2)-COOH
Lysin	lys K	H2N-(CH2)4-CH(NH2)-COOH
Methionin	met M	CH3-S-(CH2)2-CH(NH2)-COOH
Phenylalanin	phe F	Ph-CH2-CH(NH2)-COOH
Prolin	pro P	NH-(CH2)3-CH-COOH
Serin	ser S	HO-CH2-CH(NH2)-COOH
Threonin	thr T	CH3-CH(OH)-CH(NH2)-COOH
Tryptophan	trp W	Ph-NH-CH=C-CH2-CH(NH2)-COOH
Tyrosin	tyr Y	HO-p-Ph-CH2-CH(NH2)-COOH
Valin	val V	(CH3)2-CH-CH(NH2)-COOH
Unbekannt	???	

- Schreiben Sie eine **`for`** Schleife, in der
  - über die Sequenz iteriert wird
  - und die Häufigkeiten (numerische Werte im Hash) durch einen **`if..elsif...else`** Block bestimmt werden und mithilfe des Inkrement-Operators *direkt im Häufigkeiten-Hash* **`%AA_Freq_Hash`** verändert werden
  - und gleichzeitig durch Abfrage des Hashes **`%AA_Code_Hash`** einen Array mit der Sequenz in der 3-Zeichen-Aminsäuren-Notation bildet.
- Die Ausgabe auf dem Bildschirm soll ungefähr wie folgt aussehen:

```
Warnung: Unbekanntes Zeichen an Position 3: 7
Warnung: Unbekanntes Zeichen an Position 25: 3

Untersuchte Sequenz in 1-Zeichen-Notation:
MK7TRQQQQQQQQQQQQQNKDSMS3MR

Haeufigkeit von N:      1
Haeufigkeit von Q:      13
Haeufigkeit von R:      2
Haeufigkeit von S:      2
Haeufigkeit von D:      1
Haeufigkeit von T:      1
Haeufigkeit von Unbekannte:     2
Haeufigkeit von K:      2
Haeufigkeit von M:      3

Untersuchte Sequenz in 3-Zeichen-Notation:
MetLys???ThrArgGlnGlnGlnGlnGlnGlnGlnGlnGlnGlnGlnAsnLysAspSerMetSer???MetArg
```

6.3.1.1.3 Erweiterungen

- Alternativ können Sie eine Lösung entwickeln, in der die zu untersuchende Peptidsequenz
  a. über die Kommandozeile eingegeben wird oder
  b. aus einer FASTA formatierten Datei im aktuellen Arbeitsverzeichnis gelesen wird.

- Ebenso wäre es sinnvoll, den Hash für die 3-Zeichen-Notation der Aminosäuren unter Laufzeit mit Daten aus einer Textdatei zu füllen, die alle Notationen für Aminosäuren beinhaltet.

6.3.1.1.4 Lösung

Programmtext von **`AA_Freq_3Char.pl:`**

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

# Programm-Name: AA_Freq.pl
# Autor: picker
# Datum: 01/04
# Aufruf: perl AA_Freq.pl

$Peptid_Sequenz_1Z = 'MK7TRQQQQQQQQQQQQQNKDSMS3MR';
@Peptid_Sequenz_3Z = ();
$Peptid_Laenge = length $Peptid_Sequenz_1Z;
# Hash fuer Aminosaeure-Haeufigkeiten
%AA_Freq_Hash = (
            'M' => 0,
            'K' => 0,
            'T' => 0,
            'R' => 0,
            'Q' => 0,
            'N' => 0,
            'D' => 0,
            'S' => 0,
            'Unbekannte' => 0,);
%AA_Code_Hash = (
            'M' => 'Met',
            'K' => 'Lys',
            'T' => 'Thr',
            'R' => 'Arg',
            'Q' => 'Gln',
            'N' => 'Asn',
            'D' => 'Asp',
            'S' => 'Ser',
            'Unbekannte' => '???');
# Initialisierung der Schleifenvaiable
$Position = 0;
# Initialisierung des Indexwertes
$AA_Position = 0;
for ($Position = 0 ; $Position < $Peptid_Laenge ; $Position++) {
    # Inkrementelle Erhoehng des Indexwertes
    $AA_Position++;
    # Extraktion der aktuell betrachteten Aminosaeure
    $AA = substr($Peptid_Sequenz_1Z, $Position, 1);
    if ($AA eq 'M') {
        # Inkrementelle Erhoehung der Haeufigkeit
        $AA_Freq_Hash{M}++;
    }
    elsif ($AA eq 'K') {
        $AA_Freq_Hash{K}++;
    }
    elsif ($AA eq 'T') {
        $AA_Freq_Hash{T}++;
    }
    elsif ($AA eq 'R') {
        $AA_Freq_Hash{R}++;
    }
    elsif ($AA eq 'Q') {
        $AA_Freq_Hash{Q}++;
    }
    elsif ($AA eq 'N') {
        $AA_Freq_Hash{N}++;
    }
    elsif ($AA eq 'D') {
        $AA_Freq_Hash{D}++;
    }
    elsif ($AA eq 'S') {
        $AA_Freq_Hash{S}++;
    }
    else {
```

```
        print "Warnung: Unbekanntes Zeichen an Position $AA_Position: $AA\n";
        $AA_Freq_Hash{Unbekannte}++;
        # Wenn unbekanntes Zeichen (keine 3-Zeichen-Notation vorhanden)
        push (@Peptid_Sequenz_3Z, $AA_Code_Hash{Unbekannte});
    }
    # wenn bekanntes Zeichen (3-Zeichen-Notation vorhanden)
    unless (exists $AA_Code_Hash{$AA} == 0) {
        push (@Peptid_Sequenz_3Z, $AA_Code_Hash{$AA});
    }
}

# Ausgabe auf dem Bildschirm
print "\nUntersuchte Sequenz in 1-Zeichen-Notation:\n$Peptid_Sequenz_1Z\n\n";
foreach $Schluessel (keys %AA_Freq_Hash) {
    print "Haeufigkeit von $Schluessel:\t$AA_Freq_Hash{$Schluessel}\n";
}
print "\nUntersuchte Sequenz in 3-Zeichen-Notation:\n";
print @Peptid_Sequenz_3Z;
print "\n";
exit;
```

6.3.2 Die `unless` Bedingung

Durch die Verwendung der **`unless`** (*wenn nicht*) Bedingung in **`AA_Freq_3Char.pl`**

```
unless (exists $AA_Code_Hash{$AA} == 0) {
        push (@Peptid_Sequenz_3Z, $AA_Code_Hash{$AA});
    }
```

können wir vermeiden, dass in jedem einzelnen der **`if`** und **`elsif`** Blöcke, die Abfrage auf dem Hash mit den Translations-Informationen **`%AA_Code_Hash`**, und das Auffüllen des Arrays **`@Peptid_Sequenz_3Z`** wiederholt werden muss.

6.3.3 Der `exists` Funktion

Die **`Perl`** Funktion **`exists`** gibt den Wert **`true`** (gleich: den numerische Wert **`1`**) zurück wenn in einem Hash (in diesem Fall **`%AA_Code_Hash`**) ein Wert für einen spezifizierten Schlüssel (in diesem Fall der Wert der Variablen **`$AA`**) existiert.

In dem Ausdruck

```
unless (exists $AA_Code_Hash{$AA} == 0)
```

ist also die Formulierung des Tests für die **`unless`** Bedingung an den Rückgabe-Wert der **`exists`** Funktion geknüpft. Also:

> Wenn es *nicht* zutrifft (**`unless`**), dass die **`exists`** Funktion den Wert **`0`** (**`false`**) zurückgibt (weil für den Schlüssel **`$AA`** *kein* Wert im Hash **`%AA_Code_Hash`** definiert ist), wenn also ein Schlüssel mit dem Wert **`$AA`** definiert ist, dann führe den **`unless`** Block aus.

Wir könnten also **`AA_Freq_3Char.pl`** auch noch einfacher mithilfe der **`exists`** Funktion im Programm **`AA_Freq_3Char_alt.pl`** formulieren (der neue Block in gelb):

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

# Programm-Name: AA_Freq_3Char_alt.pl
# Autor: picker
# Datum: 01/04
# Aufruf: perl AA_Freq_3Char_alt.pl

$Peptid_Sequenz_1Z = 'MK7TRQQQQQQQQQQQQQNKDSMS3MR';
@Peptid_Sequenz_3Z = ();
$Peptid_Laenge = length $Peptid_Sequenz_1Z;
# Hash fuer Aminosaeure-Haeufigkeiten
%AA_Freq_Hash = (
            'M' => 0,
            'K' => 0,
            'T' => 0,
            'R' => 0,
            'Q' => 0,
```

```
            'N' => 0,
            'D' => 0,
            'S' => 0,
                 'Unbekannte' => 0,);
%AA_Code_Hash = (
            'M' => 'Met',
            'K' => 'Lys',
            'T' => 'Thr',
            'R' => 'Arg',
            'Q' => 'Gln',
            'N' => 'Asn',
            'D' => 'Asp',
            'S' => 'Ser',
                 'Unbekannte' => '???');
# Initialisierung der Schleifenvaiable
$Position = 0;
# Initialisierung des Indexwertes
$AA_Position = 0;
for ($Position = 0 ; $Position < $Peptid_Laenge ; $Position++) {
# Inkrementelle Erhoehng des Indexwertes
    $AA_Position++;
    # Extraktion der aktuell betrachteten Aminosaeure
    $AA = substr($Peptid_Sequenz_1Z, $Position, 1);
    if (exists $AA_Code_Hash{$AA}) {
        $AA_Freq_Hash{$AA}++;
        push (@Peptid_Sequenz_3Z, $AA_Code_Hash{$AA});
    }
    else {
        print "Warnung: Unbekanntes Zeichen an Position $AA_Position: $AA\n";
        $AA_Freq_Hash{Unbekannte}++;
        # Wenn unbekanntes Zeichen (keine 3-Zeichen-Notation vorhanden)
        push (@Peptid_Sequenz_3Z, $AA_Code_Hash{Unbekannte});
    }
}
# Ausgabe auf dem Bildschirm
print "\nUntersuchte Sequenz in 1-Zeichen-Notation:\n$Peptid_Sequenz_1Z\n\n";
foreach $Schluessel (keys %AA_Freq_Hash) {
    print "Haeufigkeit von $Schluessel:\t$AA_Freq_Hash{$Schluessel}\n";
}
print "\nUntersuchte Sequenz in 3-Zeichen-Notation:\n";
print @Peptid_Sequenz_3Z;
print "\n";
exit;
```