4 Block 4: Listen in `Perl`: Daten des Typs Array `@x` und Hash `%y`

4.1 Lernziele 2
4.2 Praxis 3
4.2.1 Verarbeitung von Listen mit Variablen des Typs Array `@x` 3
4.2.1.1 Deklaration, Initialisierung und Indices eines Arrays 3
4.2.1.2 Essentielle Befehle für die Verwendung von Arrays 4
4.2.1.3 Übung 5
4.2.1.3.1 Aufgabe 5
4.2.1.3.2 Lösung 6
4.2.2 Verarbeitung von *Assoziativen Listen* mit Variablen des Typs Hash `%y` 6
4.2.2.1 Deklaration und Initialisierung eines Hashes 7
4.2.2.2 Essentielle Befehle für die Verwendung von Hashes 7
4.2.2.3 Übung 8
4.2.2.3.1 Aufgabe 8
4.2.2.3.2 Lösung 9
4.2.2.4 Prüfen des Vorhandenseins eines Wertes in einem Hash 9
4.2.2.5 Übung 10
4.2.2.5.1 Aufgabe 10
4.2.2.5.2 Lösung 10

4.1 Lernziele

Ziel dieses Blocks ist es zu verstehen, wie mit `Perl` Daten, die als *Listen* (Arrays) und *assoziative Listen* (Hashes) vorliegen, auf einfache Art manipuliert werden können. Im Detail werden folgende programmatischen und syntaktischen Elemente behandelt:

- *Deklaration*, *Initialisierung* und *Indices* eines Arrays;
- Essentielle Befehle für die Verwendung von Arrays**:**
    - Die **`pop`** Funktion;
    - Die **`shift`** Funktion;
    - Die **`push`** Funktion;
    - Die **`unshift`** Funktion;
    - Die **`reverse`** Funktion;
    - Die **`scalar`** Funktion;
    - Die **`split`** Funktion;
    - Die **`join`** Funktion.
- *Deklaration* und *Initialisierung* eines Hashes;
- Essentielle Befehle für die Verwendung von Hashes:
    - Zugriff auf einen definierten Wert;
    - Überschreiben eines definierten Wertes;
    - Definition eines neuen Schlüssel-Wert-Paares;
    - Zugriff auf alle Schlüssel und Werte.
- Prüfen des Vorhandenseins eines Wertes in einem Hash mit der **`exists`** Funktion.

4.2 Praxis

Bisher haben wir uns vor allem mit *skalaren* Variablen des Typs **$x** beschäftigt. Viele Anwendungen in der Bioinformatik beschäftigen sich mit der Verarbeitungen von Daten, die keine *Einzelwerte* (einzelne oder zusammengesetzte Zeichen, numerische Werte) sind sondern eher als *Listen* betrachtet werden können.

Eine Sequenz von sieben Nukleotiden

AGGTGCA

kann so nicht nur als zusammengesetzte Variable des Typs String

```
$Sequenz = 'AGGTGCA';
```

sondern auch als *Liste* von Einzelzeichen

(A, G, G, T, G, C, A)

betrachtet werden.

Daten wie unsere Tabelle mit Restriktionsenyzm-Erkennungssequenzen

BamHI GGATCC

EcoRI GAATTC

EcoRV GATATC

werden in `Perl` gemeinhin als *assoziative Listen*

```
( "BamHI" => "GGATCC",
  "EcoRI" => "GAATTC",
  "EcoRV" => "GATATC")
```

dargestellt. In solchen *assoziativen* Listen ist ein Wert (in unserem Fall der *Name* des Enzyms auf der linken Seite) fest mit einem anderen Wert (in unserem Fall die *Erkennungssequenz* des Enzyms auf der rechten Seite) assoziiert.

Im Folgenden werden wir uns mit den `Perl`-Datentypen beschäftigen die *Listen* und *assoziative Listen* darstellen und verarbeiten können.

4.2.1 Verarbeitung von Listen mit Variablen des Typs Array `@x`

Ein *Array* ist eine Variable die mehrere *skalare* Werte in form einer *Liste* speichern kann. Die Werte können Zahlen, Zeichen oder Zeichenketten sein. Auch ein *Array von Arrays* (also eine Liste, die als Listenelemente wiederum Listen enthält) ist denkbar.

4.2.1.1 Deklaration, Initialisierung und Indices eines Arrays

Die *Deklarierung* (*Auszeichnen*, *Zur-Verfügung-Stellen* innerhalb eines Programms) eines leeren Arrays folgt der Notation:

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

Die *Initialisierung* (anfängliche Zuweisung von Werten) von Arrays erfolgt in `Perl` folgendermassen:

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

Das Zeichen `@` gefolgt von einem Namen für die Array-Variable im Gegensatz zu der `$y` Notation für skalare Variablen. Die Werte in der zugewiesenen Liste stehen in runden Klammern `()` und sich durch Kommas `,..,..,` separiert.

Die Liste der Werte eines Arrays ist numerisch *indiziert*. D.h. jeder Wert ist mit einem *Index* assoziiert, und über diesen Index kann auf ein einzelnes Array-Element zugegriffen werden:

```
# Deklaration der Liste
```

```
@Sequenz = ('A', 'G', 'G', 'T', 'G', 'C', 'A');
# Zugriff auf das erste Element mit dem Wert 'A'
print $Sequenz[0];
# Zugriff auf das zweite Element mit dem Wert 'G'
print $Sequenz[1];
```

Beachten sie, dass der erste Wert den Index `0` und nicht `1` besitzt!

Mit der Notation

```
$#Sequenz;
```

kann auf den *Index* des letzten Array-Elements zugegriffen werden. Für den Array `@Sequenz` liefert `$#Sequenz` also den Wert `6`, weil `@Sequenz` sieben Elemente enthält. Um den Wert für die letzte Index-Position zu erhalten kann folgende Notation verwendet werden:

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

Im Beispiel

```
@Sequenz = ('A', 'G', 'G', 'T', 'G', 'C', 'A');
print $Sequenz[$#Sequenz];
```

würde also `A` auf dem Bildschirm ausgegeben werden.

Über den Befehl

```
print @Sequenz;
```

kann auf den gesamten Array zugegriffen werden. Mit diesem Befehl werden alle Einzelzeichen der Liste ausgegeben:

```
AGGTGCA
```

Mit dem Befehl

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

Werden alle Einzelzeichen der Liste separiert durch Leerzeichen ausgegeben:

```
A G G T G C A
```

4.2.1.2 Essentielle Befehle für die Verwendung von Arrays

Mit der `pop` Funktion kann auf das letzte Element aus einem Array zugegriffen und entfernt werden:

```
@Sequenz = ('A', 'G', 'G', 'T', 'G', 'C', 'A');
$Position_Ende = pop @Sequenz; # $Position_Ende hat den Wert 'A'
# @Sequenz hat den Wert ('A', 'G', 'G', 'T', 'G', 'C')
```

Analog kann der `shift` Funktion kann auf das erste Element aus einem Array zugegriffen und entfernt werden:

```
@Sequenz = ('A', 'G', 'G', 'T', 'G', 'C', 'A');
$Position_1 = shift @Sequenz; # $Position_1 hat den Wert 'A'
# @Sequenz hat den Wert ('G', 'G', 'T', 'G', 'C', 'A')
```

Mit der `push` Funktion kann ein Element an das Ende eines Arrays angehängt werden:

```
@Sequenz = ('A', 'G', 'G', 'T', 'G', 'C', 'A');
$Neues_Element = 'T';
push (@Sequenz, $Neues_Element);
# @Sequenz hat den Wert ('A', 'G', 'G', 'T', 'G', 'C', 'A', 'T')
```

Analog kann dem mit der `unshift` Funktion ein Element an den Anfang eines Arrays angehängt werden:

```
@Sequenz = ('A', 'G', 'G', 'T', 'G', 'C', 'A');
$Neues_Element = 'T';
unshift (@Sequenz, $Neues_Element);
# @Sequenz hat den Wert ('T', 'A', 'G', 'G', 'T', 'G', 'C', 'A')
```

Mit der `reverse` Funktion kann der Inhalt eines Arrays *revertiert* werden:

```
@Sequenz = ('A', 'G', 'G', 'T', 'G', 'C', 'A');
@Reverse_Sequenz = reverse @Sequenz;
# @ Reverse_Sequenz hat den Wert
# ('A','C', 'G', 'T', 'G', 'G', 'A')
```

Mit der `scalar` Funktion kann die Länge eines Arrays ermittelt werden:

```
@Sequenz = ('A', 'G', 'G', 'T', 'G', 'C', 'A');
$Sequenz_Laenge = scalar @Sequenz;
# $Sequenz_Laenge hat den Wert 7
```

Mit der Funktion `split` kann eine Zeichenkette des Typs String in einen Array umgewandelt werden:

```
$Sequenz_String = 'AGGTGCA';
@Sequenz_Array = split(//, $Sequenz_String);
# @Sequenz_Array hat den Wert ('A', 'G', 'G', 'T', 'G', 'C', 'A')
```

Das erste Argument `//` ist das Muster für das die Funktion ausgeführt werden soll und besagt, dass die Trennung nach jedem Einzelzeichen erfolgen soll. Beachten sie die veränderte Trennung in folgendem Beispiel:

```
$Sequenz_String = 'AGGTGCA';
@Sequenz_Array = split(/TG/, $Sequenz_String);
# @Sequenz_Array hat den Wert ('AGG', 'CA')
```

Für den umgekehrten Fall können mit der `join` Funktion die einzelnen Elemente eines *Arrays* in einen *skalaren* Wert konvertiert werden:

```
@Sequenz_Array = ('A', 'G', 'G', 'T', 'G', 'C', 'A');
$Sequenz_String = join ('', @Sequenz_Array);
# $Sequenz_String den Wert 'AGGTGCA'
```

Das erste Argument `''` ist das Zeichen beim verbinden zwischen den Einzelzeichen eingefügt werden soll. Das Argument `''` oder `""` bewirkt, dass KEIN Zeichen eingefügt wird. Beachten sie die veränderte Verbindung in folgendem Beispiel:

```
@Sequenz_Array = ('A', 'G', 'G', 'T', 'G', 'C', 'A');
$Sequenz_String = join ('*', @Sequenz_Array);
# $Sequenz_String den Wert 'A*G*G*T*G*C*A'
```

4.2.1.3 Übung

4.2.1.3.1 Aufgabe

- Schreiben sie ein neues Programm `Sequenz_Array.pl`, das folgendes leisten soll:
    - Speichern von einer Sequenz beliebiger Länge über eine Eingabeaufforderung auf der Kommandozeile;
    - Konvertierung des Sequenz-Strings in einen Array;
    - Bestimmung der Sequenzlänge über Ermittlung der Array-Länge;
    - Ausgabe folgender Daten auf dem Bildschirm:
        - Sequenzlänge;
        - Das erste Zeichen der Sequenz;
        - Das letzte Zeichen der Sequenz;
        - Die komplette Sequenz als kontinuierliche Liste;

- Die komplette Sequenz als leerzeichen-separierte Liste;

4.2.1.3.2 Lösung

Programmtext von `Sequenz_Array.pl` (Programmelemente mit Array-Manipulationen sind gelb hervorgehoben):

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

# Programm-Name: Sequenz_Array.pl
# Autor: picker
# Datum: 11/03
# Aufruf: perl Sequenz_Array.pl

# Dieses Programm liest eine Nukleotid-Sequenz beliebiger Laenge von
# der Kommandozeile, konvertiert den Sequenz-String in einen Array,
# bestimmt die Laenge der Sequenz ueber die Anzahl der Elemente in
# diesem Array und gibt die Laenge der Sequenz, das erste und letzte
# Nukleotid, die Sequenz als kontinuierliche Liste und als leerzeichen-
# separierte Liste auf dem Bildschirm aus.

# Eingabe-Aufforderung:
print"Geben sie eine Nukleotid-Sequenz beliebiger Laenge ein: ";
$Sequenz_String = <STDIN>;
# Ohne den "chomp" Befehl waere das letzte Element in der Liste "\n"
chomp $Sequenz_String;
# Konvertierung des Eingabe-Strings in einen Array mit dem "split" Befehl:
@Sequenz_Liste = split(//, $Sequenz_String);
# Bestimmen der Laenge des Arrays @Sequenz_Liste:
$Sequenz_Laenge = @Sequenz_Liste;

# Ausgabe auf dem Bildschirm:
print "Laenge der Sequenz:\t$Sequenz_Laenge\n";
# Das erste Element in @Sequenz_Liste mit dem Index "[0]":
print "Erstes Element in Sequenz:\t$Sequenz_Liste[0]\n";
# Das letzte Element in @Sequenz_Liste mit dem Index "[$#Sequenz_Liste]":
print "Letztes Element in Sequenz:\t$Sequenz_Liste[$#Sequenz_Liste]\n";
print "Sequenz als kontinuierliche Liste:\n";
print @Sequenz_Liste;
print "\nSequenz als leerzeichen-separierte Liste:\n@Sequenz_Liste\n";
exit;
```

4.2.2 Verarbeitung von *Assoziativen Listen* mit Variablen des Typs Hash `%y`

Es gibt im wesentlichen drei Datentypen in `Perl`, zwei davon haben wir bereits kennengelernt: *Skalare* Variablen `$x` und *Array*-Variablen `@x`. Der dritte Datentyp sind *Hashes* vom Typ `%x`. Hashes werden auch als *assoziative Arrays* oder *assoziative Listen* bezeichnet.

Hashes bieten eine schnellen und effektiven Zugriff auf einen bestimmten *Wert* über einen definierten *Schlüssel*. Diese Assoziation `Schlüssel => Wert` wird in einem Hash gespeichert.

Betrachten wir unser Beispiel mit den Daten für Restriktionsenzyme in der Datei `Enzym_Master.tab`. Die Datei hat den Inhalt (verkürzte Fassung im Vergleich zum Original):

```
# Quelldatei fuer Restriktionsenzymdaten
# WWW Quelle: www.neb.com/neb/products/res_enzymes/re_update_frame.html
#  Datum: 11/03
# Tabellenformat: "Name-Tabulator-Erkennungssequenz"
# Name Erkennungssequenz
Acc16I TGCGCA
Acc36I ACCTGC
AccBSI CCGCTC
ApaI GGGCCC
BalI TGGCCA
BamHI GGATCC
BbeI GGCGCC
BciVI GTATCC
BmrI ACTGGG
```

In dem unteren Teil finden sie die Daten, die in Perl idealerweise als *assoziative Liste* oder *Hash*: dargestellt werden:

```
Acc16I TGCGCA
BamHI GGATCC
BciVI GTATCC
BmrI ACTGGG
```

Ein *Schlüssel* (der *Name* des Enzyms) ist mit einem *Wert* (der *Erkennungssequenz* des Enzyms) assoziiert, ähnlich wie in einem Nachschlagewerk / Wörterbuch. Also

```
Schlüssel => Wert
-----------------
Acc16I => TGCGCA
BamHI => GGATCC
BciVI => GTATCC
BmrI => ACTGGG
```

Wenn wir diese Daten in einem Perl *Hash* erfassen, können wir über den *Namen* (Schlüssel) eines beliebigen Enzyms seine *Erkennungssequenz* (Wert) abfragen.

4.2.2.1 Deklaration und Initialisierung eines Hashes

Die *Deklarierung* (*Auszeichnen*, *Zur-Verfügung-Stellen* innerhalb eines Programms) eines leeren Hash folgt der Notation:

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

Die *Initialisierung* (anfängliche Zuweisung von *Schlüssel-Wert-Paaren*) von Hashes erfolgt in Perl folgendermassen:

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

oder einfacher:

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

Damit ist jedem *Schlüssel* (auf der linken Seite) ein *Wert* (auf der rechten Seite zugewiesen).

4.2.2.2 Essentielle Befehle für die Verwendung von Hashes

Im letzten Abschnitt haben wir gelernt, dass auf die Elemente /Werte eines Arrays über die numerischen Index-Werte zugegriffen wird. Auf die Werte, die in einem Hash gespeichert sind, wird über den assoziierten Schlüssel zugegriffen.

Mit folgender Notation kann auf den *Wert* für einen bestimmten *Schlüssel* zugegriffen werden:

```
%Enzyme = (
        'Acc16I'   => 'TGCGCA',
        'BamHI'    => 'GGATCC',
        'BciVI'    => 'GTATCC',
        'BmrI'     => 'ACTGGG',);
$Erkennungssequenz_Acc16I = $Enzyme{Acc16I};
# $Erkennungssequenz_Acc16I hat jetzt den Wert 'TGCGCA'
```

Die geschweiften Klammern enthalten also den *Schlüssel*.

Mit folgenden Befehlen kann der Wert für einen bereits belegten Schlüssel überschrieben werden:

```
%Enzyme = (
        'Acc16I'   => 'TGCGCA',
        'BamHI'    => 'GGATCC',
        'BciVI'    => 'GTATCC',
```

```
        'BmrI'     => 'ACTGGG',);
$Erkennungssequenz_Acc16I = $Enzyme{Acc16I};
$Enzyme{Acc16I} = 'Keine gueltige Erkennungssequenz!';
$Erkennungssequenz_Acc16I = $Enzyme{Acc16I};
# $Erkennungssequenz_Acc16I hat jetzt den Wert
# 'Keine gueltige Erkennungssequenz!'
```

Mit folgenden Befehlen kann zu einem bereits bestehenden Hash ein neues *Schlüssel-Wert-Paar* hinzugefügt werden:

```
%Enzyme = (
        'Acc16I'   => 'TGCGCA',
        'BamHI'    => 'GGATCC',
        'BciVI'    => 'GTATCC',
        'BmrI'     => 'ACTGGG',);
$Enzyme{PvuII} = 'CAGCTG';
$Erkennungssequenz_ PvuII = $Enzyme{PvuII};
# Erkennungssequenz_ PvuII hat jetzt den Wert 'CAGCTG'
# Der Hash ist um ein Schluessel-Wert-Paar erweitert:
#
# %Enzyme = (
#       'Acc16I'   => 'TGCGCA',
#       'BamHI'    => 'GGATCC',
#       'BciVI'    => 'GTATCC',
#       'BmrI'     => 'ACTGGG',
#       'PvuII'    => 'CAGCTG',);
```

Einen Array aller *Schlüssel* eines Hashes erhält man mit:

```
%Enzyme = (
        'Acc16I'   => 'TGCGCA',
        'BamHI'    => 'GGATCC',
        'BciVI'    => 'GTATCC',
        'BmrI'     => 'ACTGGG',);
@Schluessel = keys %Enzyme;
# @Schluessel hat jetzt den Wert ('Acc16I', 'BamHI', 'BciVI', 'BmrI').
```

Einen Array aller *Werte* eines Hashes erhält man mit:

```
%Enzyme = (
        'Acc16I'   => 'TGCGCA',
        'BamHI'    => 'GGATCC',
        'BciVI'    => 'GTATCC',
        'BmrI'     => 'ACTGGG',);
@Werte = values %Enzyme;
# @Werte hat jetzt den Wert ('TGCGCA', 'GGATCC', 'GTATCC', 'ACTGGG').
```

4.2.2.3 Übung

4.2.2.3.1 Aufgabe

- Schreiben sie ein neues Programm `Enzym_hash.pl`, das folgendes leisten soll:
    - Initialisierung eines Hashes `%Enzyme` mit den drei Schlüssel-Wert-Paaren

Acc16I	XXXXXX
BamHI	GGATCC
BciVI	GTATCC

    - Überschreiben des Wertes für den Schlüssel `Acc16I` im Hash `%Enzyme` durch die korrekte Erkennungssequenz `TGCGCA` über eine Eingabe von der Kommandozeile;
    - Eingabe des *Namens* (Schlüssel) und der *Erkennungssequenz* (Wert) für ein neues Enzym über die Kommandozeile und Hinzufügen der Daten in den Hash `%Enzyme`. Bitte verwenden sie aus Gründen der Vergleichbarkeit das Schlüssel-Wert-Paar:

PvuII	CAGCTG

- Die Ausgabe währen des Programmablaufs auf dem Bildschirmsoll folgendermassen aussehen:

```
Die gespeicherte Erkennungssequenz fuer Acc16I ist:     XXXXXX
Geben sie die korregierte Erkennungssequenz fuer Acc16I ein: TGCGCA
Die korregierte Erkennungssequenz fuer Acc16I ist: TGCGCA
Name fuer neues Enzym: PvuII
Erkennungssequenz fuer PvuII: CAGCTG
Das neue Enzym PvuII hat die Erkennungssequenz CAGCTG
```

4.2.2.3.2 Lösung

Programmtext von `Enzym_hash.pl` (Programmelemente mit Hash-Manipulationen sind gelb hervorgehoben):

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

# Programm-Name: Enzym_hash.pl
# Autor: picker
# Datum: 11/03
# Aufruf: perl Enzym_hash.pl

# Dieses Programm enthaelt einen Hash mit Restriktionsenzym-Daten, fordert den
# Benutzer auf den fehlerhaften Wert fuer eines der vordefinierten Enzyme zu
# ueberschreiben und ein Schluessel-Wert-Paar fuer ein neues Enzym einzugeben.

# Deklaration und Initialisierung des Hashes %Enzyme
# ('Enzym' => 'Erkennungssequenz'):
%Enzyme = (
            'Acc16I'   => 'XXXXXX',
            'BamHI'    => 'GGATCC',
            'BciVI'    => 'GTATCC',);

# Ueberschreiben des Wertes fuer den Schluessel 'Acc16I':
# Ausgabe der momentanen Erkennungssequenz fuer Acc16I:
print "Die gespeicherte Erkennungssequenz fuer Acc16I ist:\t$Enzyme{Acc16I}\n";
# Eingabe der korregierten Erkennungssequenz fuer Acc16I:
print "Geben sie die korregierte Erkennungssequenz fuer Acc16I ein: ";
$Erkennungssequenz_Acc16I = <STDIN>;
chomp $Erkennungssequenz_Acc16I;
# Zuweisung der neuen Erkennungssequenz an den Schluessel Acc16I:
$Enzyme{Acc16I} = $Erkennungssequenz_Acc16I;
print "Die korregierte Erkennungssequenz fuer Acc16I ist: $Enzyme{Acc16I}\n";

# Eingabe der Daten fuer ein neues Enzym:
# Eingabe Name:
print "Name fuer neues Enzym: ";
$Enzym_Name_neu = <STDIN>;
chomp $Enzym_Name_neu;
# Eingabe Erkennungssequenz:
print "Erkennungssequenz fuer $Enzym_Name_neu: ";
$Enzym_Sequenz_neu = <STDIN>;
chomp $Enzym_Sequenz_neu;
# Anfuegen der Daten an den vordefinierten Hash %Enzyme:
$Enzyme{$Enzym_Name_neu} = $Enzym_Sequenz_neu;
print "Das neue Enzym $Enzym_Name_neu hat die Erkennungssequenz $Enzyme{$Enzym_Name_neu}\n";
exit;
```

4.2.2.4 Prüfen des Vorhandenseins eines Wertes in einem Hash

Mit folgender Bedingung kann überprüft werden ob für einen bestimmten Schlüssel ein Wert in einem Hash definiert ist:

```
%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;
}
```

Umschrieben würde der Programmabschnitt lauten:

> **Wenn im Hash `%Enzyme` ein *Wert* für den *Schlüssel* `$Auswahl` (in diesem Fall `BamHI`) definiert ist (entspricht dem Ausdruck `if (exists ($Enzyme{$Auswahl}))` dann tue was zwischen dem ersten Paar geschweifter Klammern `{}` steht. Sonst (entspricht `else`) tue was zwischen dem zweiten Paar geschweifter Klammern `{}` steht.**

Wir werden später im Detail auf programmatische Ausrücke von `Perl` wie `if...else` eingehen, die verwendet werden um den Programmfluss zu kontrollieren.

4.2.2.5 Übung

4.2.2.5.1 Aufgabe

- Schreiben sie auf der Basis von `Enzym_hash.pl` ein neues Programm `Enzym_hash_auswahl.pl`, das am Ende eine Block enthält der folgendes leisten soll (verwenden sie dazu die Information aus dem Abschnitt *Prüfen des Vorhandenseins eines Werte für einen bestimmten Schlüssel in einem Hash*):
    - Einmaliges Abfragen des Vorhandenseins eines *Wertes* (Erkennungssequenz) für einen bestimmten *Schlüssel* (Enzym-Name). Den Enzym-Namen soll der Benutzer über die Kommandozeile eingeben;
    - Wenn ein Wert vorhanden ist soll eine Ausgabe des *Schlüssel-Wert-Paares* auf dem Bildschirm gemacht werden. Wenn kein Wert vorhanden ist, soll eine Zeile ausgegeben werden, die den Benutzer darüber benachrichtigt;
    - Beispielhafte Ausgabe über den Programmfluss von `Enzym_hash_auswahl.pl`:

4.2.2.5.2 Lösung

Programmtext von `Enzym_hash_auswahl.pl` (der hinzugefügte Block im Vergleich zu `Enzym_hash.pl` ist gelb hervorgehoben):

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

# Programm-Name: Enzym_hash_auswahl.pl
# Autor: picker
# Datum: 11/03
# Aufruf: perl Enzym_hash_auswahl.pl

# Dieses Programm enthaelt einen Hash mit Restriktionsenzym-Daten, fordert den Benutzer
# auf den fehlerhaften Wert fuer eines der vordefinierten Enzyme zu ueberschreiben und
# Schluessel-Wert-Paar fuer ein neues Enzym einzugeben. Zusaetzlich prueft das Programm
# das Vorhandensein einer Erkennungssequenz fuer einen vom Benutzer ueber die Komman-
# dozeile eingegebenen Enzym-Namen.

# Deklaration und Initialisierung des Hashes %Enzyme ('Enzym' => 'Erkennungssequenz'):

%Enzyme = (
            'Acc16I'   => 'XXXXXX',
            'BamHI'    => 'GGATCC',
            'BciVI'    => 'GTATCC',);

# Ueberschreiben des Wertes fuer den Schluessel 'Acc16I':
# Ausgabe der momentanen Erkennungssequenz fuer Acc16I:
print "Die gespeicherte Erkennungssequenz fuer Acc16I ist:\t$Enzyme{Acc16I}\n";
# Eingabe der korregierten Erkennungssequenz fuer Acc16I:
print "Geben sie die korregierte Erkennungssequenz fuer Acc16I ein: ";
$Erkennungssequenz_Acc16I = <STDIN>;
chomp $Erkennungssequenz_Acc16I;
# Zuweisung der neuen Erkennungssequenz an den Schluessel Acc16I:
```

```
$Enzyme{Acc16I} = $Erkennungssequenz_Acc16I;
print "Die korregierte Erkennungssequenz fuer Acc16I ist: $Enzyme{Acc16I}\n";

# Eingabe der Daten fuer ein neues Enzym:
# Eingabe Name:
print "Name fuer neues Enzym: ";
$Enzym_Name_neu = <STDIN>;
chomp $Enzym_Name_neu;
# Eingabe Erkennungssequenz:
print "Erkennungssequenz fuer $Enzym_Name_neu: ";
$Enzym_Sequenz_neu = <STDIN>;
chomp $Enzym_Sequenz_neu;
# Anfuegen der Daten an den vordefinierten Hash %Enzyme:
$Enzyme{$Enzym_Name_neu} = $Enzym_Sequenz_neu;
print "Das neue Enzym $Enzym_Name_neu hat die Erkennungssequenz $Enzyme{$Enzym_Name_neu}\n";

# Pruefen der Erkennungssequenz fuer einen beliebigen Enzym-Namen:
print "Erkennungssequenz fuer welches Enzym suchen? ";
$Such_Enzym = <STDIN>;
chomp $Such_Enzym;
# wenn Wert fuer dem Schluessel $Such_Enzym in %Enzyme vorhanden ist:
if (exists $Enzyme{$Such_Enzym}) {
    print "Erkennungssequenz fuer $Such_Enzym: $Enzyme{$Such_Enzym}\n";
}
# wenn kein Wert fuer dem Schluessel $Such_Enzym in %Enzyme vorhanden ist:
else {
    print "Keine Erkennungssequenz fuer $Such_Enzym definiert. Programm wird beendet.\n";
    exit;
}
exit;
```