7 Block 7: Subroutinen, Funktionen und prozedurales Programmieren

7.1 Lernziele

Ziel dieses Blocks ist es zu verstehen,

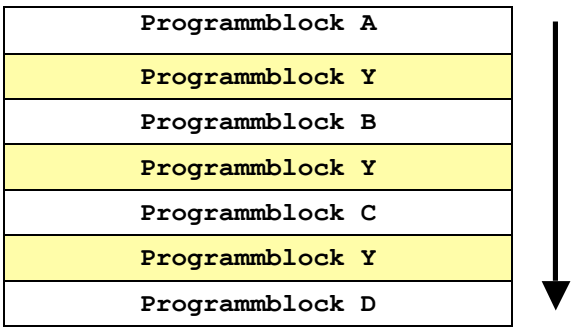
- was prozedurales Programmieren ist;
- was der Vorteil dieses Ansatzes zur Erstellung modularer Programmtexte ist;
- wie Unterprogramme / Subroutinen in **`Perl`** erstellt und verwendet werden;
- was der Unterschied zwischen Perl Standardfunktionen und benutzer-definierten Subroutinen ist;

Zusätzlich wird erläutert und erlernt wie mithilfe von **`Usage`** Blöcken, die korrekte Übergabe von Argumente beim Kommandozeilen-Aufruf eines **`Perl`** Programms kontrolliert werden kann.

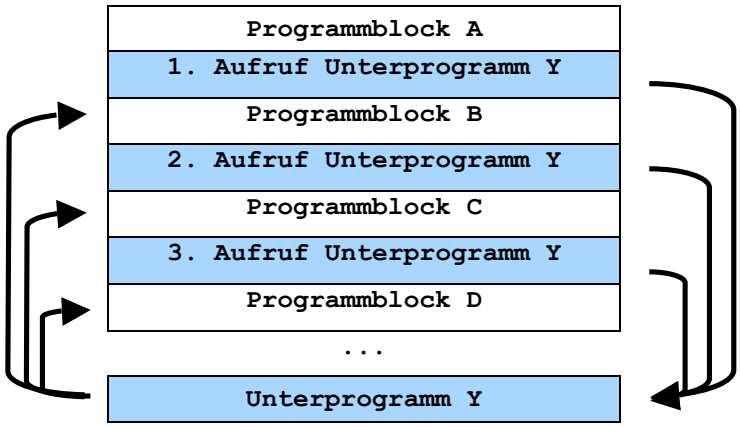
7.2 Theorie

Unterprogramme / Subroutinen werden in **Perl**, wie auch in anderen Programmiersprachen, dazu verwendet, Programmblöcke, die an mehreren Stellen eines Programms identisch verwendet werden, nur einmal zu erstellen und durch einen *Aufruf* wiederholt auszuführen:

Also statt



auszuschreiben um **Programmblock Y** wiederholt auszuführen



zu schreiben. Aus dem *Hauptprogramm* wird so ein *Unterprogramm* / *eine Subroutine* mehrmals aufgerufen – ähnlich einem Befehl, den man an mehreren Stellen verwendet. Bei jeden Aufruf *springt* der **Perl** Interpreter an das Ende des Hauptprogramms, wo das Unterprogramm definiert ist, führt es aus und kehrt in die nächste Zeile des Hauptprogramms zurück.

So kann der zu wiederholende Block mehrmals ausgeführt werden ohne jedes Mal den kompletten Programmtext zu im Hauptprogramm auszuschreiben. Der Konvention nach, werden Unterprogramme / Subroutinen an das Ende des Hauptprogramms gestellt.

Die Verwendung von Unterprogrammen hat folgende Vorteile bei der *Erstellung* und *Modifizierung* eines Programmtextes:

- Der Block muss nur einmal geschrieben, kann aber mehrmals verwendet werden: der Programmtext wird insgesamt kürzer.
- Das Unterprogramm kann leicht an einer Stelle modifiziert werden, mit Auswirkungen auf jeden Aufruf.
- Unterprogramme von *universeller* Bedeutung können auch in völlig separaten Dateien gespeichert werden, um sie in unabhängigen Programmen zu verwenden (mehr dazu in einem späteren Block).
- Unterprogramme können *rekursiv* andere Unterprogramme aufrufen, so lässt sich ein Programm sehr modular gestalten.

Die Verwendung von Unterprogrammen nennt man auch *prozedurales Programmieren* (in vielen anderen Programmiersprachen werden Subroutinen als *Prozeduren* bezeichnet) und dient dazu Programmtexte effizient, übersichtlich und strukturiert zu gestalten.

7.2.1 *Subroutinen & Funktionen*

Die Begriffe *Subroutine* und *Funktion* werden in **Perl** häufig synonym verwendet. Wir wollen trotzdem eine Unterscheidung in

1. *Benutzerdefinierte Funktionen* und
2. **Perl** *Standardfunktionen*

machen.

Benutzer-definierte Funktionen oder Subroutinen können vom Programmierer selbst geschrieben und verwendet werden. **Perl** *Standardfunktionen* sind vom Interpreter vorgegeben.

Für beide gilt:

Subroutinen / Funktionen dienen dazu Programmblöcke *zur Wiederverwendung zu isolieren.* Sie werden,

1. mit einem *Parameter* / *Argument* aufgerufen (über den vom Programmierer definierten oder von **Perl** vorgegebenen *Namen* der Subroutine),
2. dieser Parameter wird der Subroutine übergeben und *prozessiert*
3. und ein Wert an das Hauptprogramm zurückgegeben, der so genannte *Rückgabe-Wert.*

7.2.1.1 Standardfunktionen (*In Built* Functions)

In vorangegangenen Teilen des Kurses haben wir bereits an mehreren Stellen *Standardfunktionen* (engl. *in-built functions*) von **Perl** kennen gelernt. Zum Beispiel

abs

chomp

exit

int

keys

last

length

Diese Funktionen können ohne explizite Deklaration in jedem **Perl** Programm verwendet werden.

Betrachten wir ein Beispiel für eine *Standardfunktion* und eine *benutzer-definierte Funktion*, die dieselbe Arbeit leistet.

Mit

```
$Zeichen = 'AAATGCT';
$Zeichen_Laenge = length ($Zeichen);
```

können wir die Länge eines zusammengesetzten Zeichens bestimmen. In dem Beispiel wird die Standardfunktion **length** mit dem Argument **$Zeichen** aufgerufen, der Funktion wir dieser *Parameter übergeben*. Der Rückgabe-Wert wird der Variablen **$Zeichen_Laenge** zugewiesen.

Wäre die **length** Funktion in **Perl** nicht definiert, könnten wir mit dem bisher Gelernten eine Subroutine schreiben die gleiches leistet. Betrachten sie das Programmbeispiel **Funktion_Test.pl** (der Aufruf und Text der Subroutine sind gelb hervorgehoben):

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

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

$Zeichen = 'AAATGCT';
# Verwendung der Perl Standardfunktion "length":
$Zeichen_Laenge_1 = length ($Zeichen);
# Verwendung eine benutzter-definierten Subroutine:
$Zeichen_Laenge_2 = scalar_laenge($Zeichen);
#Ausgabe auf dem Bildschirm:
print "Untersuchte Zeichenkette:\t$Zeichen\n";
print "Zeichen-Laenge mit length Funktion:\t$Zeichen_Laenge_1\n";
print "Zeichen-Laenge mit benutzer-definierter Funktion:\t$Zeichen_Laenge_2\n";

# Subroutine zur Bestimmung der Laenge einer skalaren Variable
sub scalar_laenge {
    my $Zeichen = $_[0];
    my @Zeichen_Liste = split("", $Zeichen);
    return $#Zeichen_Liste+1;
}
```

Später werden wir die syntaktischen Regeln zum Erstellen und Verwenden benutzer-definierter Funktionen kennen lernen.

7.2.1.2 Alphabetische Übersicht `Perl` Standardfunktionen

```
-X

abs, accept, alarm, atan2

bind, binmode, bless

caller, chdir, chmod, chomp, chop, chown, chr, chroot, close, closedir, connect,
continue, cos, crypt

dbmclose, dbmopen, defined, delete, die, do, dump

each, endgrent, endhostent, endnetent, endprotoent, endpwent, endservent, eof, eval,
exec, exists, exit, exp

fcntl, fileno, flock, fork, format, formline

getc, getgrent, getgrgid, getgrnam, gethostbyaddr, gethostbyname, gethostent,
getlogin, getnetbyaddr, getnetbyname, getnetent, getpeername, getpgrp, getppid,
getpriority, getprotobyname, getprotobynumber, getprotoent, getpwent, getpwnam,
getpwuid, getservbyname, getservbyport, getservent, getsockname, getsockopt, glob,
gmtime, goto, grep

hex

import, index, int, ioctl

join

keys, kill

last, lc, lcfirst, length, link, listen, local, localtime, lock, log, lstat

m, map, mkdir, msgctl, msgget, msgrcv, msgsnd, my

next, no

oct, open, opendir, ord, our

pack, package, pipe, pop, pos, print, printf, prototype, push

q, qq, qr, quotemeta, qw, qx
```

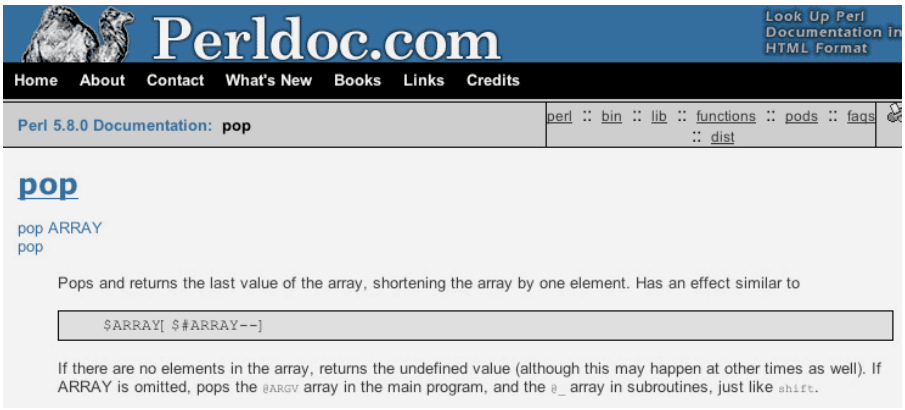
```
rand, read, readdir, readline, readlink, readpipe, recv, redo, ref, rename, require,
reset, return, reverse, rewinddir, rindex, rmdir

s, scalar, seek, seekdir, select, semctl, semget, semop, send, setgrent, sethostent,
setnetent, setpgrp, setpriority, setprotoent, setpwent, setservent, setsockopt,
shift, shmctl, shmget, shmread, shmwrite, shutdown, sin, sleep, socket, socketpair,
sort, splice, split, sprintf, sqrt, srand, stat, study, sub, substr, symlink,
syscall, sysopen, sysread, sysseek, system, syswrite

tell, telldir, tie, tied, time, times, tr, truncate

uc, ucfirst, umask, undef, unlink, unpack, unshift, untie, use, utime

values, vec

wait, waitpid, wantarray, warn, write

y
```

Für Details studieren Sie bitte das **Perl** Online Manual unter

http://www.perldoc.com/perl5.8.0/pod/perlfunc.html

7.2.1.3 Online-Dokumentation: www.perldoc.com

Ein Beispiel für die Darstellung in der Online-Dokumentation ist die **pop** Funktion:



7.3 Praxis

Als Daumenregeln sollten sie sich merken, dass Subroutinen

1. eine *klar definiertes programmatisches Problem* bearbeiten sollten
2. aus Gründen der Übersichtlichkeit *nicht länger als eine Textseite* im Programm sein sollten.

7.3.1 Standardsyntax zum Erstellen / Verwenden von Subroutinen

Betrachten wir nun das Beispiel **`Funktion_Test.pl`** genauer:

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

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

$Zeichen = 'AAATGCT';
# Verwendung der Perl Standardfunktion "length":
$Zeichen_Laenge_1 = length ($Zeichen);
# Verwendung eine benutzter-definierten Subroutine:
$Zeichen_Laenge_2 = scalar_laenge($Zeichen);
#Ausgabe auf dem Bildschirm:
print "Untersuchte Zeichenkette:\t$Zeichen\n";
print "Zeichen-Laenge mit length Funktion:\t$Zeichen_Laenge_1\n";
print "Zeichen-Laenge mit benutzer-definierter Funktion:\t$Zeichen_Laenge_2\n";

# Subroutine zur Bestimmung der Laenge einer skalaren Variable
sub scalar_laenge {
    my $Zeichen = $_[0];
    my @Zeichen_Liste = split("", $Zeichen);
    return $#Zeichen_Liste+1;
}
```

7.3.1.1 Deklaration

Subroutinen werden durch das *reservierten Schlüsselwort* **`sub`** definiert und durch geschweifte Klammern **`{...}`**, die den Text des Unterprogramms enthalten, abgetrennt; ähnlich wie bei Schleifen und Bedingungen zur Kontrolle des Programmflusses. Andere reservierte Schlüsselwort in **`Perl`** sind **`if`**, **`while`**, **`for`** etc.

```
sub scalar_laenge {
    my $Zeichen = $_[0];
    my @Zeichen_Liste = split("", $Zeichen);
    return $#Zeichen_Liste+1;
}
```

In diesem Fall hat die Subroutine den Namen **`scalar_laenge`**. Ähnlich Standardfunktionen sollten Namen von benutzer-definierten Subroutinen mit einem Kleinbuchstaben beginnen und die Funktion des Unterprogramms wiedergeben.

Subroutinen werden der Übersicht wegen am Ende eines **`Perl`** Programms deklariert und durch einen aussagekräftigen Kommentar klar vom Hauptprogramm abgesetzt.

```
# Subroutine zur Bestimmung der Laenge einer skalaren Variable
sub scalar_laenge {
    my $Zeichen = $_[0];
    my @Zeichen_Liste = split("", $Zeichen);
    return $#Zeichen_Liste+1;
}
```

7.3.1.2 Aufruf

Subroutinen werden über ihren Namen mit einem Argument / Parameter in runden Klammern **(...)** aufgerufen (analog z.B. zum Aufruf der **Perl** Standardfunktion **length** mit **length();**). In unserem Beispiel wird der Aufruf direkt in einem Zuweisungsbefehl gemacht:

```
$Zeichen_Laenge_2 = scalar_laenge($Zeichen);
```

D.h. der Aufruf der Subroutine beinhaltet

- das Schreiben des Namens

  scalar_laenge

- die Übergabe der Argumente / Parameter

  scalar_laenge($Zeichen);

- das Speichern des Rückgabe-Wertes durch einen Zuweisungsbefehl

  $Zeichen_Laenge_2 = scalar_laenge($Zeichen);

7.3.1.3 Parameterübergabe und Ausführung

Beim Aufruf

```
scalar_laenge($Zeichen);
```

wird ein Argument (in diesem Fall **$Zeichen**) an die Subroutine übergeben. Der Wert dieses Arguments (in diesem Fall die fest definierte Zeichenkette **AAATGCT**) wird der Subroutine über die **Perl** Standardvariable **@_** übergeben. Bitte beachten Sie, dass nicht der Name sondern nur der Wert der übergebenen Argumente in der Subroutine eine Rolle spielt.

In unserem Beispiel hat **@_** nach dem Aufruf den Wert **('AAATGCT')**, ist also ein Array mit einem Element.

Wird eine Subroutine mit mehreren Argumenten (durch Komma separiert in geschweiften Klammern) aufgerufen, z.B.

```
$Zeichen_1 = 'AAA';
$Zeichen_2 = 'GGG';
$Zeichen_3 = 'TTT';
neue_routine($Zeichen_1, $Zeichen_2, $Zeichen_3);
```

Dann hat der Array **@_** die entsprechende Anzahl Elemente also **('AAA', 'GGG', 'TTT')**. Also hat

- **$_[0]** den Wert **'AAA'**
- **$_[1]** den Wert **'GGG'**
- **$_[2]** den Wert **'TTT'**

im Kontext der Subroutine.

Betrachten wir nun was in der Subroutine von **Funktion_Test.pl** eigentlich berechnet wird.

```
sub scalar_laenge {
    my $Zeichen = $_[0];
    my @Zeichen_Liste = split("", $Zeichen);
    return $#Zeichen_Liste+1;
}
```

In der ersten Zeile der Subroutine

```
my $Zeichen = $_[0];
```

wird über die Indexposition 0 (**$_[0]**) auf den ersten (und in diesem Fall EINZIGEN) Wert im Array **@_** zugegriffen, also auf den String **AAATGCT**. Dieser Wert wird der Variablen **$Zeichen** Variablen zugewiesen.

7.3.1.3.1 Lokale und globale Variablen

Mit der Deklaration durch das Wort **my**

```
my $Zeichen = $_[0];
```

wird festgelegt, dass diese Variable nur im Kontext der Subroutine (allgemein: dem *Block*), in der sie deklariert ist, Gültigkeit hat: man sagt es ist eine *lokale Variable*. Obwohl im Hauptprogramm bereits eine gleichnamige *globale Variable* (Deklaration ohne **my**) verwendet wird

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

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

$Zeichen = 'AAATGCT';
# Verwendung der Perl Standardfunktion "length":
$Zeichen_Laenge_1 = length ($Zeichen);
# Verwendung eine benutzter-definierten Subroutine:
$Zeichen_Laenge_2 = scalar_laenge($Zeichen);
#Ausgabe auf dem Bildschirm:
print "Untersuchte Zeichenkette:\t$Zeichen\n";
print "Zeichen-Laenge mit length Funktion:\t$Zeichen_Laenge_1\n";
print "Zeichen-Laenge mit benutzer-definierter Funktion:\t$Zeichen_Laenge_2\n";

# Subroutine zur Bestimmung der Laenge einer skalaren Variable
sub scalar_laenge {
    my $Zeichen = $_[0];
    my @Zeichen_Liste = split("", $Zeichen);
    return $#Zeichen_Liste+1;
}
```

gibt es keine Verwechslungen. Alle Variablen in Subroutinen müssen lokal deklariert werden, um sicher zu stellen, dass nicht eine bereits deklarierte und initialisierte gleichnamige Variable aus dem Hauptprogramm verwendet wird.

In der nächsten Zeile wird ein **split** Befehl auf der Zeichenkette **$Zeichen** durchgeführt um die Einzelzeichen zu trennen. Der resultierende Array wird über

```
    my @Zeichen_Liste = split("", $Zeichen);
```

Dem lokal deklarierten Array **@Zeichen_Liste** zugewiesen.

7.3.1.4 Wert-Rückgabe

Die letzte Zeile der Subroutine enthält den **return** Befehl über den Werte in das Hauptprogramm zurückgegeben werden:

```
return $#Zeichen_Liste+1;
```

In diesem Fall ist der Wert, den **return** zurückgibt als ein numerischer Wert definiert, nämlich die letzte Indexposition des Array **@Zeichen_Liste** plus **1**. Da wir in der Subroutine die Länge eines Strings berechnen wollen, und der este Index Wert eines Arrays **0** und nicht **1** ist müssen wir **1** addieren um die Länge der Zeichenkette zu erhalten. Im Beispiel ist also der Rückgabe-Wert 7.

Dieser Rückgabe-Wert **7** aus der Subroutine **scalar_laenge** wird also der Variablen **$Zeichen_Laenge_2** im Hauptprogramm über

```
$Zeichen_Laenge_2 = scalar_laenge($Zeichen);
```

zugewiesen und kann nun weiter verarbeitet werden, z.B. als Ausgabe auf dem Bildschirm.

7.3.2 Übung

7.3.2.1 Aufgabe

- Führen sie das Programm **`Funktion_Test.pl`** auf ihrem Computer aus. Beobachten sie die Ausgabe und studieren sie den Programmtext (insbesondere die Syntax von Subroutinen);
- Schreiben sie ein neues Programm **`Lesen_Fasta.pl`**, das eine FASTA (definiert über ein Kommandozeilen-Argument) einliest und den Inhalt der Kopfzeile von der Sequenzinformation separiert. Verwenden sie die Datei **`Test_2.fasta`** aus ihrem aktuellen Arbeitsverzeichnis als zu lesende Datei.
    - Im Hauptprogramm
        - soll der Name der Datei in eine Variablen gespeichert werden;
        - der Wert dieser Variablen an eine Subroutine **`extrakt_Seq_Kopf_FASTA`** übergeben werden;
    - Die Subroutine soll den Inhalt der Kopfzeile von der Sequenzinformation separieren (siehe unten) und als Array mit zwei Elementen in das Hauptprogramm zurückgeben: **`return @resultat`**.
    - In der Subroutine **`extrakt_Seq_Kopf_FASTA`** soll(en)
        - die zu lesende Datei geöffnet, der zeilenweise Inhalt in einem lokal deklarierten Array gespeichert und die Dateien anschließend wieder geschlossen werden;
        - durch eine **`if...else`** Bedingung (in einer Schleife über den Array aller Zeilen der zu lesenden Datei) die Kopfzeile in einer lokalen Variablen **`$kopfzeile`** und die Sequenzinformation in einer lokalen Variablen **`$sequenz`** abgefangen werden;
        - diese Variablen als Array **`@resultat`** mit zwei Elementen **`($kopfzeile, $sequenz )`** an das Hauptprogramm zurückgegeben werden.
    - Das Hauptprogramm soll Kopfzeile und Sequenzinformation separat auf dem Bildschirm ausgeben (über den Zugriff auf die zwei Array-Elemente des Rückgabe-Wertes aus der Subroutine).

7.3.2.2 Lösungshinweise

- Studieren sie die Programmbeispiele **`Seq_In_Dat_raw.pl`** und **`Seq_In_foreach.pl`** aus Block 5 (Seite 7-8) zum *Einlesen* der FASTA Datei.
- Der Ansatz von **`Seq_In_Dat_raw.pl`** muss so modifiziert werden, dass in einer **`foreach`** Schleife über den Array mit dem zeilenweisen Datei-Inhalt (siehe **`Seq_In_foreach.pl`**) die Kopfzeile nicht ignoriert sondern in einer separaten Variablen aufgefangen wird.
- Verwenden sie für die Zeilen (**`$zeile`** in **`foreach`** Schleife über alle Zeilen) mit Sequenzinhalt
    - **`substr ($zeile, -1) = "";`** um den Zeilenumbruch zu entfernen (optional);
    - den Anhängen Operator **`$sequenz .= $zeile;`** um die Sequenzinformation schrittweise als String aufzubauen.

7.3.2.3 Lösung

Text von **`Lesen_Fasta.pl`** (Subroutine und Aufruf in gelb):

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

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

$Seq_Datei_Name = $ARGV[0];
chomp $Seq_Datei_Name;
# Aufruf der Subroutine
@Resultat = extrakt_Seq_Kopf_FASTA($Seq_Datei_Name);
# Ausgabe auf dem Bildschirm
print "Die Kopfzeile:\n$Resultat[0]";
print "Die Sequenz:\n$Resultat[1]\n";

# Subroutine zu Extraktion der Kopfzeile und der Sequenz
# aus einer FASTA-formatierten Datei
sub extrakt_Seq_Kopf_FASTA {
    # Zugriff auf das erste Element von @_
    my $fasta_datei_name = $_[0];
    # lokale Variable zum Speichern der Sequenzinformation
    my $sequenz = '';
    # lokale Variable zum Speichern der Kopfzeile
    my $kopfzeile = '';
    # lokale Variable zum Speichern beider Information
    my @resultat =();
    # Oeffnen der Datei
    open (FILE, "<$fasta_datei_name");
    # Zuweisung als Array
    my @datei_inhalt = <FILE>;
    # Schliessen der Datei
    close FILE;
    # Schleife ueber alle Zeilen der Datei
    foreach my $zeile (@datei_inhalt) {
        # Abfangen der Kopfzeile
        if ($zeile =~ /^>/) {
            $kopfzeile = $zeile;
        }
        # Abfangen der Sequenz-Zeilen
        else {
            substr ($zeile, -1) = "";
            $sequenz .= $zeile;
        }
    }
# Anhaengen beider Werte an den Rueckgabe-Array
push (@resultat, $kopfzeile);
push (@resultat, $sequenz);
# Rueckgabe des Arrays
return @resultat;
}
```

7.3.2.4 Aufgabe

- Führen sie das Programm **`Lesen_Fasta.pl`** auf ihrem Computer aus. Beobachten sie die Ausgabe und studieren sie den Programmtext ;
- Schreiben sie ein neues Programm **`ReverseComplement.pl`**. Übernehmen sie alle Elemente von **`Lesen_Fasta.pl`** und entwickeln sie eine zusätzliche Subroutine, die die revers-komplementäre Sequenz für die Eingabe Datei **`Test_2.fasta`** berechnet und auf dem Bildschirm ausgibt.
    - Im Hauptprogramm soll
        - die komplette Sequenzinformation (Resultat aus der Subroutine **`extrakt_Seq_Kopf_FASTA`**) als Variable an eine Subroutine **`revcom_seq`** übergeben werden. Beachten sie, dass nun keine Zeilenumbrüche mehr Im Sequenzinhalt stehen dürfen;
    - Die Subroutine soll

        - Auf der Basis des übergeben Wertes (der kompletten Sequenzinformation aus der FASTA Datei) deren Revers-Komplement berechnen und an das Hauptprogramm zurückgeben.
    - Im Hauptprogramm soll zusätzlich von der Ausgabe von auch noch die revers-komplementäre Sequenz ausgegeben werden.

7.3.2.5 Lösungshinweise

- Studieren sie die Programmfragmente in Block 3 (Seite 8-9) zu Informationen, wie man die revers-komplementäre Sequenz eines Strings berechnet.

7.3.2.6 Lösung

Text von **`ReverseComplement.pl`** (zusätzliche Zeilen in gelb):

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

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

$Seq_Datei_Name = $ARGV[0];
chomp $Seq_Datei_Name;
# Aufruf der Subroutine extrakt_Seq_Kopf_FASTA
@Resultat = extrakt_Seq_Kopf_FASTA($Seq_Datei_Name);
$Sequenz = $Resultat[1];
$RevCom_seq = revcom_seq($Sequenz);
# Ausgabe auf dem Bildschirm
print "Die Kopfzeile:\n$Resultat[0]";
print "Die Sequenz:\n5'-$Resultat[1]-3'\n";
print "Die revers-komplementaere Sequenz:\n5'-$RevCom_seq-3'\n";
exit;

# Subroutine zur Berechnung des Revers-Komplements einer
# Nukleotid-Sequenz
sub revcom_seq {
    my ($sequenz) = @_;
    my $revcom_seq = reverse $sequenz;
    $revcom_seq =~ tr/ACGTacgt/TGCAtgca/;
    return $revcom_seq;
}

# Subroutine zu Extraktion der Kopfzeile und der Sequenz
# aus einer FASTA-formatierten Datei
sub extrakt_Seq_Kopf_FASTA {
    # Zugriff auf das erste Element von @_
    my $fasta_datei_name = $_[0];
    # lokale Variable zum Speichern der Sequenzinformation
    my $sequenz = '';
    # lokale Variable zum Speichern der Kopfzeile
    my $kopfzeile = '';
    # lokale Variable zum Speichern beider Information
    my @resultat =();
    # Oeffnen der Datei
    open (FILE, "<$fasta_datei_name");
    # Zuweisung als Array
    my @datei_inhalt = <FILE>;
    # Schliessen der Datei
    close FILE;
    # Schleife ueber alle Zeilen der Datei
    foreach my $zeile (@datei_inhalt) {
        # Abfangen der Kopfzeile
        if ($zeile =~ /^>/) {
            $kopfzeile = $zeile;
        }
        # Abfangen der Sequenz-Zeilen
        else {
            substr ($zeile, -1) = "";
            $sequenz .= $zeile;
        }
    }
```

```
# Anhaengen beider Werte an den Rueckgabe-Array
    push (@resultat, $kopfzeile);
    push (@resultat, $sequenz);
# Rueckgabe des Arrays
    return @resultat;
}
```

7.3.2.7 Aufgabe

- Führen sie das Programm **`ReverseComplement.pl`** auf ihrem Computer aus. Beobachten sie die Ausgabe und studieren sie den Programmtext;
- Schreiben sie ein neues Programm **`Translate.pl`**. Übernehmen sie alle Elemente von **`ReverseComplement.pl`** bis auf die Funktionen zur Revers-Komplementierung.
- Erweitern sie es so, dass das neue Programm **`Translate.pl`**:
    - Als zweites Kommandozeilen-Argument den Namen der Datei **`Codon.tab`** aus ihrem aktuellen Arbeitsverzeichnis liest.
        - Diese Datei enthält den *Genetischen Code* in Form einer Tabelle des Typs:

```
# Genetischer Code Tabelle
# Triplet Aminosaeure (Einzelzeichen)
TCA S
TCC S
TCG S
TCT S
TTC F
TTT F
TTA L
TTG L
TAC Y
TAT Y
TAA -
TAG -
TGC C
TGT C
```

        - Mithilfe der Daten in dieser *leerzeichen-separierten* Tabelle können sie Trinukleotide (linke Spalte) in Aminosäuren (Einzelzeichen-Notation in der rechten Spalte) translatieren, also eine *Nukleotidsequenz* in eine *Peptidsequenz* umwandeln.
    - Entwickeln sie einen Ansatz, mit dem sie die Roh-Sequenz aus der FASTA Datei (zugänglich über den Rückgabe-Wert der Subroutine **`extrakt_Seq_Kopf_FASTA`**) elektronisch translatieren können:
        - Schreiben sie dazu eine erste Subroutine **`tri_nuc`**, welche mit der Roh-Sequenz als Parameter aufgerufen wird und einen Array von Trinukleotiden an das Hauptprogramm zurückgibt.
        - Schreiben sie eine zweite Subroutine **`trans_tri_nuc`**, die (1) mit dem Array von Trinukleotiden (der Rückgabe-Wert von **`tri_nuc`**) aus dem Hauptprogramm aufgerufen wird, (2) die Daten der Datei **`Codon.tab`** in einen Hash einließt, (3) den Aminosäure-Code für jedes Trinukleotid in dem Array über eine Abfrage des Hashes ermittelt und (4) das Ergebnis (die Peptidsequenz) in einer String Variablen speichert und an das Hauptprogramm zurückgibt.
    - Schliesslich sollen Kopfzeile, Nukleotidsequenz und Peptidsequenz auf dem Bildschirm ausgegeben werden.

7.3.2.8 Lösungshinweise

- Zur Subroutine **tri_nuc**, die die Nukleotidsequenz in einen Array von Trinukleotiden zerlegt:
    - Aufruf der Subroutine mit der Roh-Sequenz aus der FASTA Datei (gespeichert in einer String Variablen im Hauptprogramm);
    - Zugriff auf die Roh-Sequenz in der Subroutine über **$_[0]**;
    - Zerlegung des Strings in einen Array aus Einzelnukleotiden mithilfe der **split** Funktion auf einem leeren Muster **""** (siehe Block 4 Seite 5).
    - In einer **while** Schleife über alle Einzelnukleotide
        - Separieren von drei Einzelnukleotiden vom Anfang des Arrays mit

```
@ein_tri_nuc = splice (@volle_sequenz, 0, 3);
```

        - Verbinden der drei Einzelnukleotide mit

```
$tri_nuc = join ('', @ein_tri_nuc);
```

        - Anhängen der Trinukleotide (aus **$tri_nuc**) an eine Resultat-Array, der an das Hauptprogramm zurückgegeben wird;

      bis alle Array Element *konsumiert* sind.

- Zur Subroutine **trans_tri_nuc**, die einen Array von Trinukleotiden in eine Peptidsequenz translatiert:
    - Aufruf der Subroutine mit dem Array von Trinukleotiden (Rückgabe-Wert von **tri_nuc**, gespeichert in einer Array Variablen im Hauptprogramm);
    - Öffnen, Lesen und Schließen der Codon-Tabelle über Zugriff aus der Subroutine auf die globale Variable **$ARGV[1]**;
    - Speichern der Codo-Aminosäure-Information in einem Hash des Typs

```
%hash = ('TCA' => 'S',
            .
            .
            .
        'GGT' => 'G',);
```

      Dazu muss die Datei zeilenweise (in einer **foreach** Schleife analog zur Subroutine **extrakt_Seq_Kopf_FASTA**) gelesen werden. *Spalten* sie die Zeilen mit der Information (die Kopfzeilen müssen durch eine **if...else** Bedingung ignoriert werden) mit dem Befehl

```
split (" ", $zeile)
```

      und speichern sie die Information in einem Array der Form

```
(Trinukleotid_1, Aminosäure_1, Trinukleotid_2,
Aminosäure_2, ...)
```

    - Mit dem **shift** (1. shift Trinukleotid_1, 2. shift Aminosäure_1, 3. shift Trinukleotid_2 ...) Befehl können sie den Array schrittweise abbauen und die Element als *Schlüssel* (Trinukleotid) und *Werte* (Aminosäure) in einen Hash schreiben.
    - Als letztes müssen sie eine **foreach** Schleife über alle Trinukleotide bilden, sie im Codon-Hash suchen, die Aminosäure-Notation für alle auftretenden Trinukleotide abragen und eine String Variable **$as_sequenz**, die die gesamte Peptidsequenz speichert und ans Hauptprogramm zurückgibt, schrittweise durch

```
$as_sequenz .= $einzel_as;
```

(wobei **$einzel_as** die Aminosäure für das momentan betrachtete Trinukleotid ist) aufbauen.

7.3.2.9 Lösung

Text von **Tranlate.pl** (Subroutinen in gelb, Hauptprogramm in blau):

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

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

$Seq_Datei_Name = $ARGV[0];
$Codon_Datei_Name = $ARGV[1];
chomp $Seq_Datei_Name;
chomp $Codon_Datei_Name;
# Lesen der FASTA Datei
@FASTA_Resultat = extrakt_Seq_Kopf_FASTA($Seq_Datei_Name);
$Sequenz = $FASTA_Resultat[1];
# Aufspalten der Nukleotid-Sequenz in einen Array von Trinukleotiden
@Trinukleotide = tri_nuc($Sequenz);
# Translatierung der Trinukleotid-Sequenz
$AS_Sequenz = trans_tri_nuc(@Trinukleotide);
# Ausgabe auf dem Bildschirm
print "Die Kopfzeile:\n$FASTA_Resultat[0]";
print "Die Nukleotid-Sequenz:\n5' - $FASTA_Resultat[1] - 3'\n";
print "Die Peptid-Sequenz:\nH2N - $AS_Sequenz - COOH\n";
exit;

# Subroutine zur Translatierung eines Arrays von Trinukleotiden
sub trans_tri_nuc {
    my @tri_nuc = @_;
    my %codon_hash = ();
    my $as_sequenz = '';
    #print "Array: @tri_nuc";
    # Oeffnen der Datei mit der Codon Information
    open (FILE, "<$ARGV[1]");
    # Zuweisung als Array
    my @datei_inhalt = <FILE>;
    # Schliessen der Datei
    close FILE;
    # Schleife ueber alle Zeilen der Datei
    foreach my $zeile (@datei_inhalt) {
        # Abfangen der Kopfzeile
        if ($zeile =~ /^#/) {
            next;
        }
        # Abfangen der Sequenz-Zeilen
        else {
            my @codon_daten = split (" ", $zeile);
            my $triplet = shift @codon_daten;
            my $as = shift @codon_daten;
            # print"Triplet: $triplet AS: $as\n";
            $codon_hash{$triplet} = "$as";
        }
    }
    foreach my $element(@tri_nuc) {
        my $einzel_as = $codon_hash{$element};
        $as_sequenz .= $einzel_as;
    }
    return $as_sequenz;
}

# Subroutine zur Zerlegung einer Nukleotid-Sequenz in Trinukleotide
sub tri_nuc {
    my @nuc_seq = split ("", $_[0]);
    my @tri_nuc_full = ();
    while (@nuc_seq) {
        my @ein_tri_nuc = splice(@nuc_seq, 0, 3);
        my $tri_nuc = join ('', @ein_tri_nuc);
        push (@tri_nuc_full, $tri_nuc);
    }
    return @tri_nuc_full;
```

```
}

# Subroutine zu Extraktion der Kopfzeile und der Sequenz
# aus einer FASTA-formatierten Datei
sub extrakt_Seq_Kopf_FASTA {
    # Zugriff auf das erste Element von @_
    my $fasta_datei_name = $_[0];
    # lokale Variable zum Speichern der Sequenzinformation
    my $sequenz = '';
    # lokale Variable zum Speichern der Kopfzeile
    my $kopfzeile = '';
    # lokale Variable zum Speichern beider Information
    my @resultat =();
    # Oeffnen der Datei
    open (FILE, "<$fasta_datei_name");
    # Zuweisung als Array
    my @datei_inhalt = <FILE>;
    # Schliessen der Datei
    close FILE;
    # Schleife ueber alle Zeilen der Datei
    foreach my $zeile (@datei_inhalt) {
        # Abfangen der Kopfzeile
        if ($zeile =~ /^>/) {
            $kopfzeile = $zeile;
        }
        # Abfangen der Sequenz-Zeilen
        else {
            substr ($zeile, -1) = "";
            $sequenz .= $zeile;
        }
    }
    # Anhaengen beider Werte an den Rueckgabe-Array
    push (@resultat, $kopfzeile);
    push (@resultat, $sequenz);
    # Rueckgabe des Arrays
    return @resultat;
}
```

7.3.3 Programmaufruf mit Kommandozeilen-Argumenten und $USAGE

Wie wir bereits gelernt haben verwendet **Perl** für die Speicherung und den Zugriff auf Kommandozeilen-Argumente die Standard-Variable **@ARGV**. Damit können bereits beim Programmaufruf Argumente übergeben werden.

Im Programm **Translate.pl** wird über **$ARGV[0]** auf das erste und über **$ARGV[0]** auf das zweite Kommandozeilen-Argument zugegriffen,

```
$Seq_Datei_Name = $ARGV[0];
$Codon_Datei_Name = $ARGV[1];
```

um die Dateien **Test_2.fasta** und **Codon.tab** mit den Eingabe-Daten für die Berechnungen zu lesen.

Wenn das Programm ohne oder mit einer falschen Anzahl von Kommandozeilen-Argumenten aufgerufen wird, z.B. mit

```
perl Translate.pl
```

produziert der **Perl** Interpreter bei der Ausführung mehrere Fehlermeldungen (unten in gelb) und eine *sinnlose* Ausgabe (unten in blau), weil dem Programm die Quelldaten für die angestellten Berechnungen fehlen:

```
Use of uninitialized value in scalar chomp at Translate.pl line 10.
Use of uninitialized value in scalar chomp at Translate.pl line 11.
Use of uninitialized value in concatenation (.) at Translate.pl line 83.
readline() on closed filehandle main::FILE at Translate.pl line 85.
```

```
Use of uninitialized value in concatenation (.) at Translate.pl line 32.
readline() on closed filehandle main::FILE at Translate.pl line 34.
Die Kopfzeile:
Die Nukleotid-Sequenz:
5' -  - 3'
Die Peptid-Sequenz:
H2N -  - COOH
```

Mit einer einfachen **unless** Bedingung (siehe Block 6) kann man innerhalb eines **Perl** Programms eine Kontroll-Bedingung formulieren, die überprüft ob das Programm richtig aufgerufen wurde. So kann sichergestellt werden, dass z.B. die korrekte Anzahl von Kommandozeilen-Argumenten vom Benutzer beim Aufruf des Programms definiert wird oder ein kontrollierter Abbruch des Programms stattfindet.

Betrachten wir folgendes, einfaches Beispiel **Usage_test.pl**:

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

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

$USAGE = "Programmaufruf: $0 [Eingabe-Datei 1]!\n";
unless (@ARGV) {
    print $USAGE;
    exit;
}
print "Korrekter Programmaufruf:\n";
print "-------------------------\n";
print "Kommandozeilen-Argumente: @ARGV\n";
exit
```

In der Zeile

```
$USAGE = "Programmaufruf: $0 [Eingabe-Datei 1] [Eingabe-Datei 2]!\n";
```

wird die Variable **$USAGE** initialisiert. Sie enthält die Zeichenkette,

Programmaufruf: $0 [Eingabe-Datei 1] [Eingabe-Datei 2]!\n

die später auf dem Bildschirm ausgegeben werden soll, falls die Programmaufruf von der Kommandozeile fehlerhaft war. **$0** ist eine Standardvariable in **Perl**, über die aus dem Programm auf den Programm-Namen (in diesem Fall also **Usage_test.pl**) zugegriffen werden kann. Bitte beachten sie die Grosschreibung **$USAGE**, die konventionsgemäß für Variablen mit konstantem Wert verwendet wird.

In der nächsten Zeile steht die **unless** Bedingung, die dazu führt, dass entweder

- Der **unless** Block ausgeführt wird (wenn die Bedingung nicht erfüllt ist) oder
- Das Hauptprogramm weiter ausgeführt wird (wenn die Bedingung erfüllt ist).

Die **unless** Bedingung

```
unless (@ARGV) {
    print $USAGE;
    exit;
}
```

bedeutet:

> Wenn es *nicht* zutrifft (**unless**), dass ein oder mehrere Kommandozeilen-Argumente übergeben wurden, also der Array **@ARGV** ein oder mehrere Elemente enthält, dann führe den **unless** Block (Ausgabe von **$USAGE** auf dem Bildschirm und Programmabbruch mit **exit**) aus.

Bitte beachten sie, dass die **unless** Bedingung nur dann nicht erfüllt ist, also das Hauptprogramm weiter ausgeführt wird, wenn der Array **@ARGV** ÜBERHAUPT KEINE Argumente enthält! Sie prüfen also nicht die Anzahl der übergebenen Kommandozeilen-Argumente.

Im anschliessenden Block

```
print "Korrekter Programmaufruf:\n";
print "------------------------\n";
print "Kommandozeilen-Argumente: @ARGV\n";
exit;
```

werden alle übergebenen Kommandozeilen-Argumente durch Zugriff auf **@ARGV** auf dem Bildschirm ausgegeben.

7.3.3.1 Aufgabe

- Führen sie das Programm **Usage_test.pl** auf ihrem Computer aus. Beobachten sie die Ausgabe und studieren sie den Programmtext;
- Schreiben sie auf der Basis von **Translate.pl**. ein neues Programm **Translate_usage.pl** Übernehmen sie alle Elemente von **Translate.pl** und fügen sie einen Block ein, der den korrekten Programmaufruf mit GENAU ZWEI Kommandozeilen-Argumenten überprüft.

7.3.3.2 Lösungshinweise

- Beachten sie, dass sie nicht wie im Beispiel nur überprüfen sollen ob ÜBERHAUPT Kommandozeilen-Argumente übergeben werden, sondern ob genau zwei übergeben werden.
- Verwenden sie zur Implementierung, die Befehle zur Bestimmung der Anzahl von Elementen in einem Array (Block 4).

7.3.3.3 Lösung

Programmtext von **Translate_new.pl** (der **Usage** Block ist gelb hervorgehoben):

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

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

$USAGE = "Programmaufruf: $0 [FASTA-Sequenz-Datei] [Codon-Tabelle]!\n";
unless ($#ARGV == 1) {
    print $USAGE;
    exit;
}
$Seq_Datei_Name = $ARGV[0];
$Codon_Datei_Name = $ARGV[1];
chomp $Seq_Datei_Name;
chomp $Codon_Datei_Name;
# Lesen der FASTA Datei
@FASTA_Resultat = extrakt_Seq_Kopf_FASTA($Seq_Datei_Name);
$Sequenz = $FASTA_Resultat[1];
# Aufspalten der Nukleotid-Sequenz in einen Array von Trinukleotiden
@Trinukleotide = tri_nuc($Sequenz);
# Translatierung der Trinukleotid-Sequenz
$AS_Sequenz = trans_tri_nuc(@Trinukleotide);
# Ausgabe auf dem Bildschirm
print "Die Kopfzeile:\n$FASTA_Resultat[0]";
print "Die Nukleotid-Sequenz:\n5' - $FASTA_Resultat[1] - 3'\n";
print "Die Peptid-Sequenz:\nH2N - $AS_Sequenz - COOH\n";
exit;

# Subroutine zur Translatierung eines Arrays von Trinukleotiden
sub trans_tri_nuc {
    my @tri_nuc = @_;
    my %codon_hash = ();
    my $as_sequenz = '';
    #print "Array: @tri_nuc";
    # Oeffnen der Datei mit der Codon Information
    open (FILE, "<$ARGV[1]");
    # Zuweisung als Array
    my @datei_inhalt = <FILE>;
```

```
    # Schliessen der Datei
    close FILE;
    # Schleife ueber alle Zeilen der Datei
    foreach my $zeile (@datei_inhalt) {
        # Abfangen der Kopfzeile
        if ($zeile =~ /^#/) {
            next;
        }
        # Abfangen der Sequenz-Zeilen
        else {
            my @codon_daten = split (" ", $zeile);
            my $triplet = shift @codon_daten;
            my $as = shift @codon_daten;
            # print"Triplet: $triplet AS: $as\n";
            $codon_hash{$triplet} = "$as";
        }
    }
    foreach my $element(@tri_nuc) {
        my $einzel_as = $codon_hash{$element};
        $as_sequenz .= $einzel_as;
    }
    return $as_sequenz;
}

# Subroutine zur Zerlegung einer Nukleotid-Sequenz in Trinukleotide
sub tri_nuc {
    my @nuc_seq = split ("", $_[0]);
    my @tri_nuc_full = ();
    while (@nuc_seq) {
        my @ein_tri_nuc = splice(@nuc_seq, 0, 3);
        my $tri_nuc = join ('', @ein_tri_nuc);
        push (@tri_nuc_full, $tri_nuc);
    }
    return @tri_nuc_full;
}

# Subroutine zu Extraktion der Kopfzeile und der Sequenz
# aus einer FASTA-formatierten Datei
sub extrakt_Seq_Kopf_FASTA {
    # Zugriff auf das erste Element von @_
    my $fasta_datei_name = $_[0];
    # lokale Variable zum Speichern der Sequenzinformation
    my $sequenz = '';
    # lokale Variable zum Speichern der Kopfzeile
    my $kopfzeile = '';
    # lokale Variable zum Speichern beider Information
    my @resultat =();
    # Oeffnen der Datei
    open (FILE, "<$fasta_datei_name");
    # Zuweisung als Array
    my @datei_inhalt = <FILE>;
    # Schliessen der Datei
    close FILE;
    # Schleife ueber alle Zeilen der Datei
    foreach my $zeile (@datei_inhalt) {
        # Abfangen der Kopfzeile
        if ($zeile =~ /^>/) {
            $kopfzeile = $zeile;
        }
        # Abfangen der Sequenz-Zeilen
        else {
            substr ($zeile, -1) = "";
            $sequenz .= $zeile;
        }
    }
    # Anhaengen beider Werte an den Rueckgabe-Array
    push (@resultat, $kopfzeile);
    push (@resultat, $sequenz);
    # Rueckgabe des Arrays
    return @resultat;
}
```