10 Block 10: Verwendung von Modulen in Perl

10.1 Lernziele

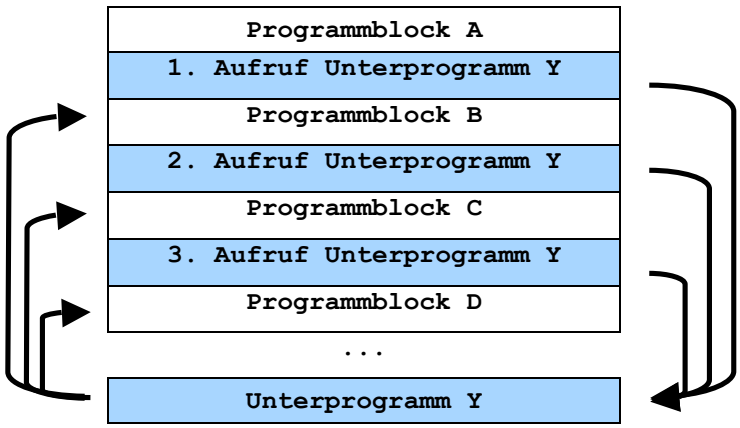
Ziel dieses Blocks ist es die Grundbegriffe *modularen* Programmierens in **`Perl`** zu erlernen. Behandelt wird

- wie Modul-Dateien **`*.pm`** erstellt und strukturiert werden;
- wie Code-Elemente in Modul-Dateien von **`Perl`** Programmen aufgerufen und importiert werden;

10.2 Theorie

10.2.1 Warum Module?

In Block 7 haben wir kennen gelernt, wie durch Verwendung benutzer-definierter Subroutinen **Perl**-Programme effizienter und übersichtlicher gestaltet werden können. Durch den wiederholten Aufruf einer Subroutine in ein und demselben Programm kann vermieden werden, dass ein identischer Block mehrmals explizit formuliert werden muss:



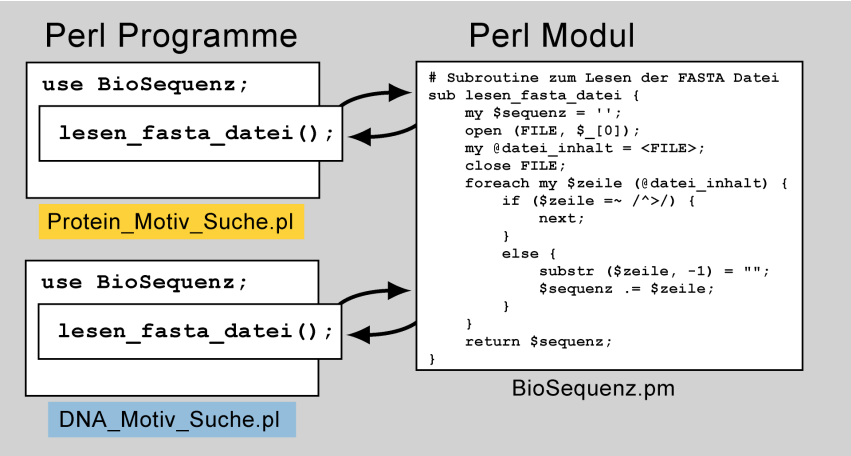
Viele der Subroutinen, die wir entwickelt haben, können aber nicht nur Verwendung in einem Programm finden, sondern sind so *generisch*, dass sie idealerweise mehreren unabhängigen **Perl**-Programmen zur Verfügung gestellt werden sollten.

Ein Beispiel für so eine Subroutine, die in vielen unsere Programme Anwendung findet ist **sub lesen_fasta_datei**

```
# Subroutine zum Lesen der FASTA Datei
sub lesen_fasta_datei {
    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;
}
```

Wie können wir erreichen, dass wir ein und dieselbe Subroutinen nicht immer wieder explizit in jedes Programm schreiben müssen, in dem wir sie benutzen wollen?

Subroutinen können in **`Perl`** in sogennanten *Modulen* gesammelt werden, um sie dann aus einem beliebigen Programm *extern* aufzurufen.



Module sind also nichts anderes als Sammlungen von Subroutinen *zur allgemeinen Benutzung*.

Die Vorteile liegen auf der Hand:

- Code-Fragmente müssen nicht unnötig dupliziert werden;
- Die Subroutinen können an einer zentralen Stelle verwaltet / geändert werden.

10.3 Praxis

10.3.1 Module erstellen

Um ein Modul zu erstellen müssen sie eine separate Textdatei erstellen, die alle Subroutinen aufnimmt, die sie in einem Modul ablegen wollen. Diese Datei muss die Endung **.pm** haben, um vom **Perl** Interpreter als Modul-Datei erkannt zu werden.

Betrachten wir ein Beispiel, das benutzer-definierte Modul **BioSequenz.pm**, das alle generischen Subroutinen aufnehmen soll, die zur Verwendung biologischer Sequenzen in der Bioinformatik nützlich sind.

Als erstes schreiben wir die Subroutine **sub lesen_fasta_datei** in unsere Modul-Datei. **BioSequenz.pm** sieht dann folgendermaßen aus:

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

Der einzige Unterschied zur Syntax: die letzte Zeile einer Modul-Datei ***.pm** muss **1;** sein. Wenn diese Zeile nicht enthalten ist produziert **Perl** eine Fehlermeldung und bricht die Ausführung des Programms ab.

Module können nicht nur Subroutinen aufnehmen sondern jegliche Art von Perl-Code. Wir können z.B. eine zweite Modul-Datei **BioData.pm** definieren, die wichtige biologische Konstanten definiert:

```
#Definition Triplet => AA-Einzelzeichen
%codon_hash = ();
open (FILE, "<Codon.tab");
my @datei_inhalt = <FILE>;
close FILE;
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;
            $codon_hash{$triplet} = "$as";
        }
}
1;
```

Analog zur der Subroutine **lesen_fasta_datei** aus **BioSequenz.pm**, steht nun die Variable **%codon_hash** **Perl** Programmen zur Verfügung, die das Modul **BioData.pm** aufrufen. Die Daten zur Translation von Nukleotid-Triplets in **%codon_hash**

```
('TGT' => 'C',
 'AAA' => 'K',
 'AAC' => 'N',
       .
       .
       .
 'TTT' => 'F',)
```

können nun von vielen **Perl** Programmen gleichzeitig als *biologische Konstante* verwendet werden.

Statt die Translations-Information *on-the-fly* in dem Modul **BioData.pm** aus der Datei **Codon.tab** im aktuellen Arbeitsverzeichnis zu berechnen, kann der hash natürlich auch direkt in **BioData.pm** deklariert werden (nur ein Teil der *assoziativen Liste* ist unten wiedergegeben):

```
%codon_hash =  ('TGT' => 'C',
                'AAA' => 'K',
                'AAC' => 'N',
                'ACA' => 'T',
                'AAG' => 'K',
                'ACC' => 'T',
                'ATA' => 'I',
                'CAA' => 'Q',
                'ATC' => 'I',
                'CAC' => 'H',
                'ACG' => 'T',
                'AGA' => 'R',
                'ATG' => 'M',
                'CAG' => 'Q',);
1;
```

Beim Schreiben von Modulen sollten sie beachten, dass die Code-Fragmente (Variablen, Subroutinen etc.) sinnvoll nach ihrer Verwendung gruppiert werden. Fertigen sie deshalb unterschiedliche Module für separate Funktionen und Daten an, also z.B.

BioSequenz.pm:	Methoden zur Manipulation von Sequenz-Strings
BioData.pm:	Definitionen wichtiger biologischer Konstanten
SequenzFormate.pm:	Methoden zum Lesen von Dateien in unterschiedlichen Formaten

10.3.2 Module aufrufen

Um ein Modul aus einem Programm aufzurufen, und so die im Modul definierten Subroutinen dem Programm zur Verfügung zu stellen, müssen sie die **Perl** Standard-Funktion **use** verwenden.

Betrachten wir das Beispiel **Modul_test.pl**:

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

use BioSequenz;
use BioData;

@Fasta_Data = lesen_fasta_datei($ARGV[0]);
print "Kopfzeile:\n$Fasta_Data[0]\nSequenz:\n$Fasta_Data[1]\n\n";
foreach $Schluessel (keys %codon_hash) {
    print "Triplet: $Schluessel => Aminosaeure: $codon_hash{$Schluessel}\n";
}
exit;
```

In den Zeilen

```
use BioSequenz;
use BioData;
```

werden die beiden Module **BioSequenz.pm** und **BioData.pm** aus dem Programm aufgerufen. Über die **use** Funktion wird dem Perl Interpreter mitgeteilt, dass alle Code-Fragment in den beiden Modul-Dateien dem Programm **Modul_test.pl** zur Verfügung gestellt werden müssen. Beachten sie, dass die Datei-Endung **.pm** nicht angegeben werden muss.

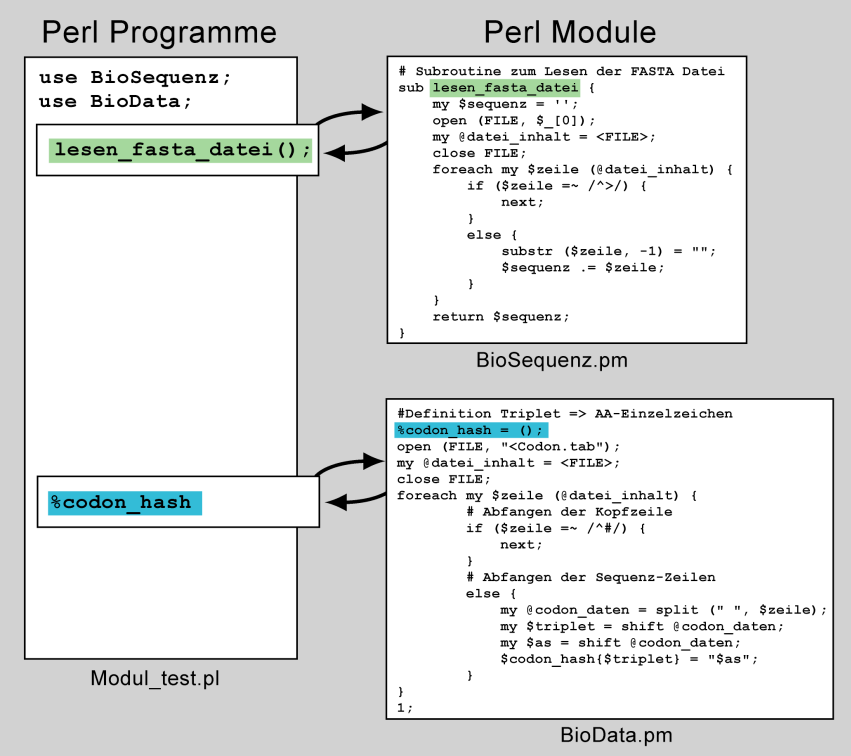
D.h. wir können in der Zeile

```
@Fasta_Data = lesen_fasta_datei($ARGV[0]);
```

auf die Subroutine **lesen_fasta_datei** aus **BioSequenz.pm** zugreifen, und in den Zeilen

```
foreach $Schluessel (keys %codon_hash) {
    print "Triplet: $Schluessel => Aminosaeure: $codon_hash{$Schluessel}\n";
}
```

auf den Hash **%codon_hash**, der im Module **BioData.pm** deklariert ist. Das Zusammenspiel zwischen dem Hauptprogramm **Modul_test.pl** und den beiden Modul-Dateien **BioSequenz.pm** und **BioData.pm** lässt sich also wie folgt veranschaulichen:



Wenn sie die **use** Funktion wie oben gezeigt aufrufen, sucht der **Perl** Interpreter im aktuellen Arbeitsverzeichnis und im *Standardpfad* nach den Modul-Dateien. Wenn sie den Pfad zu einem Modul explizit im Programm deklarieren wollen können sie folgende Notation verwenden:

Mit

```
use lib 'usr/picker/my_perl_lib';
use BioSequenz;
```

weisen sie den Interpreter an nur im Verzeichnis **usr/picker/my_perl_lib** nach Modul-Dateien zu suchen. Die Datei **BioSequenz.pm** muss also in diesem Verzeichnis stehen, damit der **use Biosequenz** Aufruf erfolgreich ist.

Mit

```
use usr::picker::my_perl_lib::BioSequenz;
```

weisen sie den Interpreter an, das Modul mit dem Pfad **usr::picker::my_perl_lib::BioSequenz** aufzurufen. D.h. in dieser Notation werden die Verzeichnisse nicht durch Schrägstrich **/** sondern durch doppelte Doppelpunkte **::** gekennzeichnet.

Die **use** Funktion kann auch verwendet werden um Subroutinen und Variablen zwischen Modulen zugänglich zu machen, d.h. sie können in einer Modul-Datei eine andere aufrufen, oder sogar spezifisch in einer Subroutine innerhalb einer Modul-Datei:

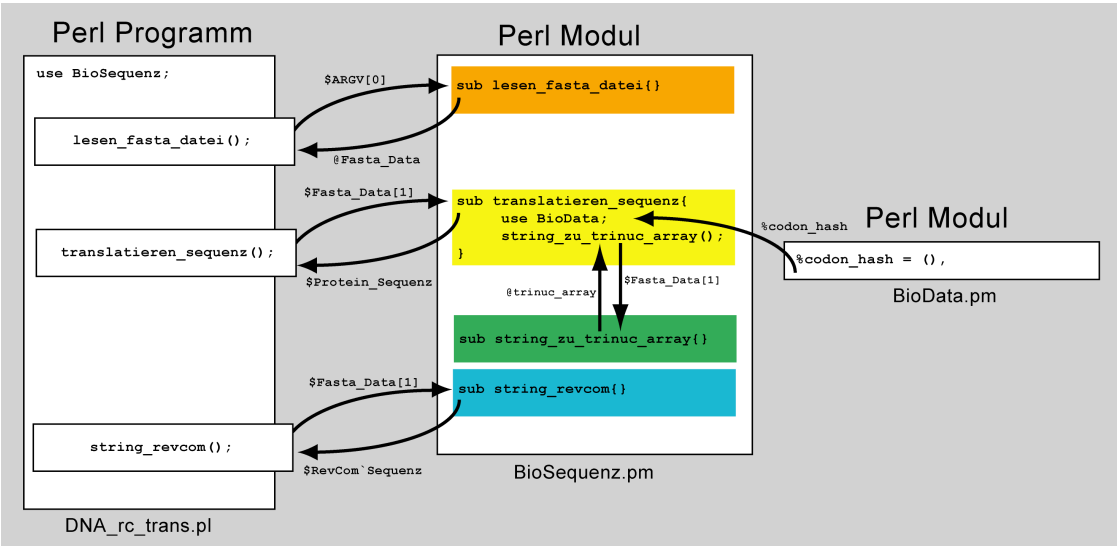
```
use modul_1;
sub subroutine_1 {
     use modul_2;
}
1;
```

10.3.3 Aufgabe

- Führen sie das Programm **Modul_test.pl** auf ihrem Computer aus. Beobachten sie die Ausgabe und studieren sie den Programmtext (insbesondere auch den Inhalt der assoziierten Modul-Dateien);
- Schreiben sie ein neues, modulares Programm **DNA_rc_trans.pl** das folgenden Spezifikationen genügen soll:
    - *Einlesen* einer FASTA-formatierten DNA-Sequenzdatei (z.B. **pZERO2.fasta** oder **Test.fasta**) aus dem aktuellen Arbeitsverzeichnis durch Übergabe des Dateinamens als Kommandozeilen-Argument. Separate Ausgabe des Dateinamens Kopfzeile und Rohsequenz auf dem Bildschirm;
    - *Revers-Komplementierung* der Sequenz und Ausgabe auf dem Bildschirm;
    - *Translatieren* der Nukleotid-Sequenz in die Aminosäure-Sequenz und Ausgabe auf dem Bildschirm.
- Einlesen, *Revers-Komplementierung* und *Translation* sollen durch Aufrufen von externen Subroutinen aus der dazu zu erweiternden Modul-Datei **BioSequenz.pm** gelöst werden;
- Die Subroutinen zum Lesen und der Revers-Komplementierung sollen direkt aus dem Hauptprogramm aufgerufen werden und den nötigen Rückgabe-Wert für die Bildschirm-Ausgabe liefern;
- Die Subroutine zum Translatieren soll eine weiter Subroutine zur Umwandlung eines Sequenz-Strings in einen Array von Trinukleotiden im Modul **BioSequenz.pm** intern aufrufen und den Hash mit der Translations-Information (Triplet => Aminosäure) aus dem Modul **BioData.pm** importieren.

10.3.4 Lösungshinweis

Die Gesamt-Architektur und der Datenfluss zwischen dem Hauptprogramm und den Modulen soll ungefähr wie folgt aussehen:



10.3.5 Lösung

Programmtext des Hauptprogramms **DNA_rc_trans.pl**:

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

use BioSequenz;

# Aufruf der Subroutine zum Lesen der FASTA Datei
# aus dem Modul BioSequenz.pm
@Fasta_Data = lesen_fasta_datei($ARGV[0]);

# Aufruf der Subroutine zur Revers-Komplementierung
# aus dem Modul BioSequenz.pm
$RevCom_Sequenz = string_revcom($Fasta_Data[1]);

# Aufruf der Subroutine zur Translatierung
# aus dem Modul BioSequenz.pm
$Protein_Sequenz = translatieren_sequenz($Fasta_Data[1]);

# Ausgabe auf dem Bildschirm
print "\nUntersuchte Sequenz:\t$ARGV[0]\n";
print "\nFASTA-Kopfzeile:\t$Fasta_Data[0]\n";
print "\nRoh-Sequenz:\n5' - $Fasta_Data[1] - 3'\n";
print "\nRevers-komplementaere Sequenz:\n5' - $RevCom_Sequenz - 3'\n";
print "\nTranslatierte Sequenz:\nH2N - $Protein_Sequenz - COOH\n";
exit;
```

Programmtext der Modul-Datei **BioSequenz.pm**:

```
# Subroutine zum Lesen einer FASTA Datei
# Uebergabe-Parameter: Name einer FASTA-
#                      formatierten Datei
# Rueckgabe-Wert: zwei-wertiger Array ([Kopfzeile], [Roh-Sequenz])
sub lesen_fasta_datei {
    my @data = ();
    my $sequenz = '';
    open (FILE, $_[0]);
    my @datei_inhalt = <FILE>;
    close FILE;
    foreach my $zeile (@datei_inhalt) {
        if ($zeile =~ /^(>.*)\n/) {
            push (@data, $1);
        }
        else {
            substr ($zeile, -1) = "";
            $sequenz .= $zeile;
        }
    }
    push (@data, $sequenz);
    return @data;
}

# Subroutine zum Translatieren eines DNA-Sequenz-String
# Uebergabe-Parameter: DNA-Sequenz-String
# Rueckgabe-Wert: Protein-Sequenz-String
# Bemerkungen:    (1) verwendet Modul-Datei BioData.pm
#                 (2) ruft Subroutine string_zu_trinuc_array
#                     aus BioSequenz.pm auf
sub translatieren_sequenz {
    use BioData;
    my $dna_sequenz = $_[0];
    my $protein_sequenz = '';
    my %codon_hash = %CODON_HASH;
    my $einzel_as = '';
    my $trinuc;
    my @trinuc_array = string_zu_trinuc_array($dna_sequenz);
    foreach $trinuc (@trinuc_array) {
        $einzel_as = $codon_hash{$trinuc};
        $protein_sequenz .= $einzel_as;
    }
    return $protein_sequenz;
}

# Subroutine zum Zerlegen eines DNA-Sequenz-String in Trinukleotide
```

```
# Uebergabe-Parameter: DNA-Sequenz-String
# Rueckgabe-Wert: Array mit Trinukleotiden
# Bemerkungen:    (1) wird aus  Subroutine translatieren_sequenz in
#                     BioSequenz.pm aufgerufen
sub string_zu_trinuc_array {
    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 zur Revers-Komplementierung eines DNA-Sequenz-String
# Uebergabe-Parameter: DNA-Sequenz-String
# Rueckgabe-Wert: DNA-Sequenz-String (revers-komplementaer)
sub string_revcom {
    my $sequenz = $_[0];
    my $revcom_seq = reverse $sequenz;
    $revcom_seq =~ tr/ACGTacgt/TGCAtgca/;
    return $revcom_seq;;
}
1;
```

Programmtext der Modul-Datei **`BioData.pm`**:

```
#Definition Triplet => AA-Einzelzeichen
%CODON_HASH = ();
open (FILE, "<Codon.tab");
my @datei_inhalt = <FILE>;
close FILE;
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;
            $CODON_HASH{$triplet} = "$as";
        }
}
1;
```