Stefan Kurtz,
Dirk Willrodt

Universität Hamburg
Zentrum für Bioinformatik

Programmierung in der Bioinformatik
Wintersemester 2013
Übungen zur Vorlesung: Ausgabe am 06.01.2014

Aufgabe 11.1 Eine typische Aufgabe in der Programmierung besteht in der Auswertung von Kommandozeilenoptionen .

Dafür gibt es ein Ruby-Standardmodul, das Sie dabei unterstützt. Das Modul heißt `OptionParser` und wird in der Ruby-Dokumentation im Internet beispielhaft erklärt. (http://ruby-doc.org/stdlib-1.9.3/libdoc/optparse/rdoc/OptionParser.html)

Ihre Aufgabe ist es nun, einen Parser für die Kommandozeilenoptionen eines Programmes zum parsen von GenBankfiles zu schreiben. Es sollen die folgende Optionen implementiert werden:

`--selecttop <entry>[,<entry> ... ]` Spezifiziert mindestens einen Genbank-Toplevel-Identifier, also z.B. `LOCUS`, `DEFINITION`, oder `ORIGIN`. Mehrere Bezeichner werden durch Kommata getrennt. [1]

`--echo` Spezifiziert, dass der echo-Modus verwendet werden soll. (Es wird erst in einer Aufgabe auf einem späteren Blatt klar, was das bedeutet.)

`--search <regexp>` Spezifiziert, dass der search-Modus mit dem angegebenen regulären Ausdruck verwendet weren soll. (Es wird erst in einer Aufgabe auf einem späteren Blatt klar, was das bedeutet.)

`--help` Gibt eine kurze Anleitung zur Benutzung des Skripts aus.

Außerdem soll folgendes für die Optionen gelten:

- setzen Sie mit dem accessor `.banner` die Usage-message auf wie folgt:

  ```
  Usage: #{$0} [options] inputfile
  ```

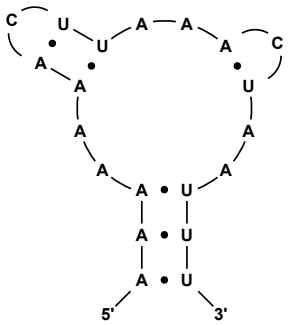
- `inputfile` ist ein notwendiger Parameter und soll der Name einer existierenden Datei sein.
- Keine der Optionen darf doppelt vorkommen.
- `--echo` und `--search` dürfen nicht zusammen benutzt werden.
- Bei der Benutzung von `--help` bricht das Skript nach der Ausgabe ab.
- Nach `--selecttop` dürfen nur Genbank-Bezeichner folgen.

Aufgabe 11.2 RNAs sind aus Ribonucleinsäuren aufgebaute einzelsträngige Moleküle, ähnlich wie DNA sind sie aus 4 Basen aufgebaut (`ACGU`). mRNAs sind in erster Linie über ihre Sequenz definiert, diese wird bei der Translation in eine Aminosäuresequenz übersetzt.

Es gibt aber auch RNAs, die andere Funktionen in einer Zelle übernehmen. Diese Funktionen werden nur indirekt über die Sequenz definiert. Stattdessen ist die Tertiärstruktur der RNA ausschlaggebend (ihre Faltung im dreidimensionalen Raum) relevant für die Funktion. Da diese Struktur aber schwer computergestützt vorhergesagt werden kann, vereinfacht man in der Bioinformatik die

[1] Für einen Hinweis zur Auswertung suchen Sie nach „List of arguments“ in der `OptionParser` Dokumentation.

Struktur von RNA auf ein zweidimensionales Abbild, welches in erster Linie die intermolekularen Basenpaarungen darstellt.



Dies könnte z.B. so aussehen:

Man kann eine solche Sekundärstruktur durch eine Sequenz von Klammern und Punkten darstellen, der sogenannten *dot-bracket*-Notation (auch Vienna-Notation) darstellen. Basen, die ein Watson-Crick Basenpaar bilden, sind durch zusammengehörige Klammern gekennzeichnet, ungepaarte Basen durch einen Punkt. Zusammen mit der RNA-Sequenz kann man daraus jederzeit die Sekundärstruktur rekonstruieren. Die zu obiger Graphik gehörende Vienna-Notation sieht z.B. so aus:

```
AAAAAAACTTAAACTAATTT
(((..((.))..(.)..)))
```

Ihre Aufgabe ist es, ein Ruby-Methode `validateDotBracket` zu schreiben, die eine RNA-Sekundärstruktur in dot-bracket-Notation validiert. D.h. sie überprüft, ob der übergebene Punkt/Klammer-String nach den folgenden Regeln gebildet werden kann:

- `.` ist eine gültige RNA-Struktur
- `()` ist eine gültige RNA-Struktur
- falls r eine gültige RNA-Struktur ist, dann ist es auch (r)
- falls r und s gültige RNA-Strukturen sind, dann ist auch rs auch eine gültige RNA-Struktur.

Wenn ein Fehler gefunden wird, soll die Methode einen Ausnahmfehler mit `raise` produzieren. Ist die Struktur valide soll die Methode alle Basenpaare in folgender Form ausgeben:

```
1-20
9-13
...
```

und `true` zurückgeben.

Verwenden Sie dazu den Stack, den Sie bereits entwickelt haben. (Oder die in Stine zur Verfügung gestellte Klasse.)

Verwenden Sie Ihre Methode in einem Skript, dass mehrere RNA-Sekundärstrukturen aus einer Datei einliest und validiert.

Das Dateiformat für diese Aufgabe ist wie folgt: Eine Struktur pro Zeile ohne Leerzeichen. Nach einem Leerzeichen folgt `valid` oder `invalid`. Damit können Sie in Ihrem Skript testen, ob ihre Validierungsmethode richtig arbeitet.

In STiNE finden Sie die Datei `bracketTest.txt`. Diese enthält mehrere Einträge mit Angaben ob diese valide sind oder nicht.

Aufgabe 11.3 (In dieser Aufgabe geht es darum einen Iterator für Multi-Fasta-Dateien zu schreiben. Zu Iteratoren siehe Skript: Iterators.

Sie haben schon in früheren Aufgaben Iteratoren verwendet, wenn Sie z.B. die Methode `each` auf ein Array angewendet haben:

```
my_array.each do |element| puts element end
```

Dabei wird der Methode `each` der Block **`do...end`** übergeben und mit jedem einzelnen Element des Arrays aufgerufen.

In dieser Aufgabe sollen Sie einen Iterator für das Parsen von Multi-Fasta- Dateien entwickeln. Schreiben Sie eine Klasse, deren Instanzen mit einem Dateinamen initialisiert werden. Implementieren Sie für diese Klasse eine `each`-Methode, in der Sie mit **`yield`** einen übergebenen Block für jeden Fasta-Eintrag ausführen.

Der Fasta-Eintrag soll dabei als Array aus 2 Elementen an den Block übergeben werden. Der Header als erstes Element und die Sequenz ohne Leerzeichen oder Zeilenumbrüche als zweites.

Ein Beispiel soll dies verdeutlichen:

```
my_fasta = FastaIterator.new("filename.fasta")
my_fasta.each do |header, sequence|
  #do sth with header or sequence
end
```

Schreiben Sie dann ein Skript, welches diese Klasse verwendet und zweimal `each` benutzt um

- die Fasta-Einträge formatiert auszugeben.
- eine Längenverteilung der Sequenzen zu berechnen. Teilen Sie dazu die Längen in 10er-Bins ein (1-10,11-20,...)

Die Lösungen zu diesen Aufgaben werden am 20.01.2014 besprochen.