Ruby-Einarbeitungsaufgaben

1) Thema „Warshall-Algorithmus“ (4 Personen)

Zweck: Zweistellige Relationen auf einer Menge M (von Strings?) modellieren und nützlichen Operationen darauf bereitstellen, mit denen man Relationen erzeugen, einlesen, drucken, speichern, laden und vor allem die transitive Hülle bilden kann.

Teilaufgaben:

1. Eine Klasse „Menge“ zur Modellierung von Mengen (über M) implementieren:
   - Daten in einem Ruby-Array halten
   - Konstruktor erhält als Parameter einen Array
   - Array-Methoden „uniq“ und „sort“ verwenden, um interne Darstellung frei von Doubletten und aufsteigend sortiert zu halten.
   - Methoden: to_s, add(elem), enthaelt?(x), speichern, laden, ...
   - Einen „each“-Iterator auf „Menge“ bereitstellen
2. Die Klasse „Relation“ implementieren:
   - Die Basismenge im Attribut @basis halten.
   - Zu jedem Element aus @basis zwei „Menge-Objekte“ halten: eines für die unmittelbaren Vorgänger des Elements, eines für die unmittelbaren Nachfolger. Dazu dienen zwei Ruby-Hash-wertige Attribute @vor und @nach von „Relation“. Schlüssel ist jeweils das Element aus @basis, Wert die Vorgänger- bzw. Nachfolger-Menge.
   - Konstruktor erhält Array von Paaren (Paar ist Array mit 2 Elementen)
   - Grundmethoden: to_s, addPaar(x,y), speichern, laden, drucken, ...
   - Relationenspezifische Methoden: Vereinigung zweier Relationen, das Relationenprodukt, die **transitive Hülle nach Warshall** (vgl. dazu Buch S. 283-285 Mitte und S. 289, Ada-Version in Einf. II - Folien).
   - Abfragemethoden: enthaeltPaar?(x,y), Mengen-wertige Abfragen wie vorgaenger(x), nachfolger(x), Iteratoren auf Basismenge.
3. **Doku:** Für alle Klassen ausführliche Inline-Doku mit RDoc erstellen und für alle Methoden Beispielaufrufe als UnitTests ausarbeiten. Vorher in UnitTests und RDoc einarbeiten.

Beachte: Operationen außer Konstruktoren und add...s „funktional“ implementieren, d.h. so, daß sie wie in Haskell jeweils neue Objekte zurückgeben und nicht das angesprochene Objekt modifizieren!

2) Thema „Grammatiken“ (5 Personen)

Zweck: Grammatiken modellieren und nützlichen Operationen darauf bereitstellen. Eine Grammatik wird durch folgende Klassen dargestellt:

- „Symbol“ mit String-wertigen Attribut @name (sowohl für terminale als auch für nichtterminale Symbole verwenden).
- „Alternative“ hat eine Folge (d.h. ein Ruby-Array) @symbs von Symbolen.
- „Regel“ hat eine @linkeSeite (Symbol) und eine Folge @alts von Alternativen (sowohl die Alternativen einer Regel als auch die Symbole einer Alternative sind also durchnummeriert).
- „Grammatik“ hat ein Startsymbol @start, eine Menge (siehe Aufgabe 1.1) @term von terminalen Symbolen, eine Menge @nonterm von nichtterminalen Symbolen sowie eine Menge @regeln von Regeln. Um den Zugriff zu vereinfachen, ist @regeln ein Ruby-Hash, in dem jede Regel unter ihrer linken Seite als Schlüssel zu finden ist (redundant, aber nützlich).

Teilaufgaben:

1. Die Klassen implementieren und mit geeigneten Konstruktoren versehen. Der Konstruktor von „Grammatik“ erhält einen Text im gleichen Format wie beim Rules-Abschnitt von SIC-Grammatiken:
   - Der Text ist eine Folge von Regeln, die jeweils durch genau eine Leerzeile voneinander getrennt sind.
   - Eine Regel beginnt mit dem Symbol auf der linken Seite; darauf folgt in der gleichen Zeile der Pfeil „->“ und die erste Alternative.
   - Die Alternativen sind jeweils durch „Zeilenwechsel“, beliebige Einrückung und „|“ voneinander getrennt.
   - Nichtterminale sind die Symbole, die als linke Seite vorkommen; alle anderen Symbole sind terminal.
   - Startsymbol ist die linke Seite der ersten Regel (also, das Symbol, mit dem der Text beginnt).
   - Die Konstruktoren der Grammatik-Teile („Regel“, „Alternative“ und „Symbol“) erhalten als Parameter die jeweiligen Text-Anteile.
2. Methoden zum Zugriff auf Grammatiken und all ihre Bestandteile implementieren:
   - getStartsymbol und die Array-wertigen getRegeln, getNichtterminale, getterminale für Grammatiken
   - anzahlAlternativen und [i] (Redefinition von „[...]“ siehe Pickaxe-Buch!) für „Regel“
   - bei „Alternative“ analog

3. Methoden zum Drucken, Laden und Speichern von Grammatiken und all ihren Bestandteilen implementieren sowie eine Grundfunktion:
   - laden und speichern (mit YAML).
   - to_s erzeugt das vom Konstruktor erwartete Format (in allen Klassen).
   - to_SIC erzeugt komplette SIC-Grammatik-Beschreibung (neben den Regeln das Startsymbol, die Terminalmenge, der Grammatiknamme, ...).
   - epsilon-Symbole bestimmt die Menge (Ruby-Array) der Nichtterminale, die epsilon produzieren können (siehe Buch S 63f); auch hier wirken Methoden in verschiedenen Klassen zusammen: istEpsilon in „Alternative“ und „Regel“, epsilon-Symbole in Grammatik.

4. Ausdrucks-Grammatiken aus Kurzangaben erzeugen (Beispiele siehe unten):
   - Die Kurzangaben sind eine Folge von Zeilen, die jeweils ein Operatortoken (z.B. addOp) in seinen Eigenschaften beschreiben.
   - Bei binären Operatortoken hat die Zeile den Aufbau:

     <Assoc> <Operator>

     <Assoc> steht für left, right oder none und legt die Assoziativität des Operators fest; none besagt, daß keine Assoziativität vorliegt und daher bei mehr als einem Operator voll geklammert werden muß.
   - Bei unären Operatortoken hat die Zeile den Aufbau:

     <Pos> <Operator>
   - <Pos> steht für pre oder post und legt fest, ob der Operator vor oder nach dem Operanden steht.
   - Die Operatorzeilen sind nach wachsender Präzedenz angeordnet, d.h. die erste Zeile gehört zu dem Operatortoken, das am schwächsten bindet.
   - Die letzte Zeile hat die Form:

     atom <token1>, ..., <tokenN>

     und führt die atomaren Bestandteile des Ausdrucks ein.
   - Die erste Zeile hat die Form:

     name <Bezeichner>

     und führt den Namen der Ausdrucksgrammatik ein, mit dem alle erzeugten Nonterminals beginnen.
   - Dieses Verfahren ist als Klassenmethode from_Yacc in der Klasse Regeln zu implementieren (vgl. auch Buch, S. 52 ff.).

- Beispiel „arithmetische Ausdrücke“: Aus den Angaben

  ```
  name Arith
  left addOp
  left mulOp
  right expOp
  pre addOp
  atom zahl, name
  ```

  soll erzeugt werden:

  ```
  Arith -> Arith1

  Arith1 -> Arith1 addOp Arith2
          | Arith2

  Arith2 -> Arith2 mulOp Arith3
          | Arith3

  Arith3 -> Arith4 expOp Arith3
          | Arith3

  Arith4 -> addOp Arith4
          | Arith5

  Arith5 -> (Arith1 )
          | zahl
          | name
  ```

- Beispiel „einfache logische Ausdrücke“: Aus den Angaben

  ```
  name L
  none logBinOp
  pre negOp
  atom true, false, name
  ```

  soll erzeugt werden:

  ```
  L -> L1

  L1 -> L2 logBinOp L2
          | L2

  L2 -> negOp L2
          | L3

  L3 -> ( L1 )
          | true
          | false
          | name
  ```

5. **Doku:** Für alle Klassen ausführliche Inline-Doku mit RDoc erstellen und für alle Methoden Beispielaufrufe als UnitTests ausarbeiten. Vorher in UnitTests und RDoc einarbeiten.

Beachte: Operationen außer Konstruktoren und add...s „funktional“ implementieren, d.h. so, daß sie wie in Haskell jeweils neue Objekte zurückgeben und nicht das angesprochene Objekt modifizieren!

3) Thema „Scanner“ (4 Personen)

Zweck: Als Vorübung zur Entwicklung eines Scannergenerators einige Scanner „von Hand“ in Ruby programmieren.

- In die regulären Ausdrücke **(„Patterns“)** von Ruby einarbeiten.
- Gemeinsame Vorübung: Das Telegramm-Problem in Ruby implementieren (siehe Buch S. 135).

Teilaufgaben:

1. Scanner für die aus der Einführungsvorlesung I bekannte while-Sprache:
   - Beschreiben Sie die folgenden Tokenklassen der while-Sprache durch Ruby-Patterns:

     name (Bezeichner beginnt mit Buchstabe, dahinter beliebig viele weitere Buchstaben und Ziffern gemischt)
     zahl (ganze Zahl ohne Vorzeichen)
     addOp (+ oder -)
     mulOp (*, div oder mod)
     auf (öffnende Klammer)
     zu (schließende Klammer)
     ergibt (Zuweisungssymbol :=)
     strichpunkt (ein ;)
   - Erstellen Sie eine Ruby-Methode „zerlege“, die den Text eines while-Programms gemäß den Regeln (R1) und (R2) aus dem Buch, S. 125f in die den Token entsprechenden Teile zerlegt, ohne aber die Zugehörigkeit zu den Tokenklassen zu ermitteln (Ergebnis ist ein Array von Strings). Hinweis: die Methode „scan“ von Ruby verwenden!
   - Erstellen Sie eine Ruby-Methode „bestimme“, die das Ergebnis der Methode „zerlege“ überführt in das Ergebnis eines SIC-Scannerlaufs.
2. Erstellen Sie analog einen Scanner zum Buch-Beispiel von S. 126.
3. Im Buch ist auf den Seiten 258-261 ein Markov-Toolset beschrieben, das i.w. aus einem in sed implementierten Markov-Generator besteht, der in sed implementierte Markov-Interpreter erzeugt (dazu ein paar Kommandoprozeduren).
   - Reimplementieren Sie den Interpreter aus Abbildung 8.1.3 „von Hand“ in Ruby.
   - Programmieren Sie in Ruby einen Generator, der solche (in Ruby geschriebenen) Interpreter aus dem Text eines Markov-Algorithmus erzeugt. Der Generator liefert also den Text eines Ruby-Programms als String. Anders gesagt: Übertragen Sie den Markov-Generator von sed nach Ruby!

- Schreiben Sie ein Ruby-Rahmenprogramm, welches die Kommandoprozeduren ersetzt. Dieses Programm soll nacheinander:
    - den Namen <m> einer Datei mit dem Text des Markov-Algorithmus (in der Pfeil-Darstellung gegeben) erfragt;
    - daraus den entsprechenden Ruby-Interpreter erzeugt;
    - den Namen einer Datei mit dem Text der Markov-Eingabe erfragt;
    - mit Hilfe des Interpreters zu der Eingabe einen Trace in der Datei <m>Trace.txt erzeugt (Hinweis: „eval“-Methode von Ruby nutzen!).

4. **Doku:** Für alle Klassen ausführliche Inline-Doku mit RDoc erstellen und für alle Methoden Beispielaufrufe als UnitTests ausarbeiten. Vorher in UnitTests und RDoc einarbeiten.

4) Thema „Plotter“ (3 Personen)

Zweck: Die Möglichkeiten von Ruby in Verbindung mit der FX-API (Fox für Ruby) erkunden. Dazu zuerst in diese API anhand der mitgelieferten Beispiele und der Kurzeinführung einarbeiten.

Teilaufgaben:

1. Eines der Beispiel-Programme zeigt ein „Scribble-Fenster“. Ergänzen Sie dieses Programm um folgende Möglichkeiten, die per Menü und per Hotkey zur Verfügung gestellt werden sollen:
   - Farbe des Zeichenstifts wählen
   - Zeichenfläche löschen
   - Speichern und Laden von Bildern in Datei (mit Marshal-dump und –load)
   - PDF-Fassung des Bilds erzeugen: Siehe dazu pdf-Writer etc. unter den unten angegebenen Links.
2. Auf der Basis von „Scribble“ ein „Plotter-Fenster“ implementieren:
   - Im „Plotter-Fenster“ sind die Freihand-Zeichenmöglichkeiten abgeschaltet, die in der Teilaufgabe 1 implementierten Menüpunkte aber weiter verfügbar.
   - Ein weiterer Menüpunkt (oder drei Menüpunkte) öffnet einen Dialog, in dem dreierlei unabhängig voneinander eingestellt werden kann (zu allen drei Angaben gibt es Default-Werte, die bei der Initialisierung des Fensters bereits verwendet werden; Veränderungen an den Werten führen zum Neuzeichnen des Fensterinhalts):
     - Die Definition einer Funktion als Text, also ein String mit einem gültigen Ruby-Ausdruck in x, z.B. “0.3 * x**3 – x**2“.
     - Die x- und y-Koordinaten, mit denen der untere, obere, linke und rechte Rand des Fensters identifiziert werden soll (xmin, xmax, ymin, ymax).
     - Position des Mittelpunkts des Koordinatenkreuzes (muß nicht (0,0) sein, muß aber im Fenster sichtbar sein).
3. **Doku:** Für alles ausführliche Inline-Doku mit RDoc erstellen. Dazu in RDoc einarbeiten. Außerdem einen Handzettel (Word-Datei) erstellen, der anderen eine möglichst schnelle Einarbeitung in FX erlaubt. Dabei dürfen vorhandene Quellen beliebig geplündert werden, müssen aber angegeben werden (mit den entsprechenden Links).

Einige Links:

http://www.fxruby.org/ (beachte die Beispiele in Kap. 6 des Tutorials!)

http://rubyforge.org/projects/ruby-pdf/

http://openfacts.berlios.de/index.phtml?title=PDF_Dateien_mit_Ruby_erstellen