Christian-Albrechts-Universität zu Kiel
Institut für Informatik
Lehrstuhl für Programmiersprachen und Übersetzerkonstruktion

Priv.-Doz. Dr. Frank Huch, Christoph Daniel Schulze, Sandra Dylus

Institut für Informatik

7. Klausur zur Vorlesung „Informatik für Nebenfächler“

WS 14/15

Hinweise zur Klausur:

- Sie dürfen jegliches Material, allerdings keine mitgebrachten elektronischen Geräte verwenden. Mobiltelefone bitte ausschalten.
- Zur Bearbeitung der Aufgaben sollen Sie Ihren iLearn-Account verwenden, allerdings nicht mit Ihrem normalen Passwort, sondern dem iLearn-Passwort auf Ihrem Klausurdeckblatt.
- Bei technischen Problemen kontaktieren Sie uns bitte frühzeitig. Gegebenenfalls werden wir zum Ausgleich technischer Probleme die Bearbeitungszeit verlängern, also keine Panik.
- Die Einlesezeit beträgt 10 Minuten. Diese Zeit soll ausschließlich dazu dienen, die Klausuraufgaben zu lesen. Sie dürfen in dieser Zeit nicht schreiben.
- Während der Einlesezeit werden wir Ihre Identität anhand Ihres Lichtbild- und Ihres Studienausweises überprüfen. Bitte legen Sie diese gut sichtbar auf Ihrem Tisch bereit. Unterschreiben Sie das Deckblatt erst, wenn wir vorbeikommen.
- Die anschließende Bearbeitungszeit beträgt 150 Minuten.
- Bitte notieren Sie auf jedem Blatt, das Sie abgeben möchten, Ihren vollständigen Namen und Ihre Matrikelnummer. Bei Abgaben im iLearn ist dies nicht erforderlich.
- Da die Klausur im Anschluss direkt noch einmal geschrieben wird, können Sie leider nicht früher abgeben und gehen. Sollte dies Probleme ergeben, kontaktieren Sie uns bitte frühzeitig.
- Am Ende der Bearbeitungszeit werden wir die Klausur an Ihrem Platz zusammenheften und einsammeln. Ihre elektronischen Abgaben werden wir zur Korrektur ausdrucken.
- Die Bekanntgabe der Noten und die Einsicht in die Korrekturen findet am 17. April 11 bis 12 Uhr in Raum 715 im Hochhaus statt.
- Sie benötigen 35 Punkte um die Klausur zu bestehen. Die maximal erreichbare Punktzahl beträgt 80 Punkte. Sie können also eine 1,0 erreichen, auch wenn Sie nur 70 Punkte bearbeiten. Werden alle Aufgaben bearbeitet, so können fehlende Punkte aus anderen Aufgaben ausgeglichen werden.
- Teilen Sie sich Ihre Zeit gut ein. Beißen Sie sich nicht an (noch) fehlerhaften Programmen fest.

Aufgabe 1 - Rekursion

8 Punkte

Eine Funktion zum Summieren aller Elemente eines Arrays kann in Ruby iterativ recht einfach wie folgt programmiert werden:

```
def sum(a)
  sum = 0
  for i in 0..a.size-1 do
    sum = sum + a[i]
  end
  return sum
end
```

Definieren Sie eine *rekursive* Variante `sum_rec(a)`, welche ebenfalls die Summe aller Elemente eines Arrays berechnet. Sie dürfen für die Rekursion auch eine zusätzliche Hilfsfunktion einführen.

Aufgabe 2 - EBNF und Reguläre Ausdrücke 18 Punkte

Markup-Sprachen werden verwendet um (meistens im Web-Kontext) einfach Textformatierungen vornehmen zu können. Beispiele, bei denen zur Texteingabe Markup-Sprachen verwendet werden, sind die Wikipedia oder aber die Wiki-Eingabe im iLearn. Wir wollen hier nicht den vollen Sprachumfang von Markup-Sprachen betrachten, sondern uns lediglich Verweise (auch *Link* genannt) in Markup-Sprachen näher anschauen. Die folgende EBNF definiert die Sprache aller Markup-Texte mit Verweisen:

```
ML   ::= (Text | '[' Text ']' '(' Text ')') [' ' ML]
Text ::= {Sym}
Sym  ::= 'a' | .. | 'z' | 'A' | ... | 'Z' | ' ' | '.' | ...
```

Die möglichen Symbole (`Sym`) sollen hierbei zur Vereinfachung beliebige Zeichen, aber keine eckigen oder runden Klammern sein. Ein möglicher Text ist der folgende:

```
Im [Internet](http://de.wikipedia.org/wiki/Internet) können viele interessante
[Informationen](http://de.wikipedia.org/wiki/Information) abgerufen werden.
Besonders beliebt für eine erste Orientierung ist die [Wikipedia](http://de.wikipedia.org).
```

(a) Überführen Sie die angegebene EBNF in eine äquivalente BNF.

(b) Implementieren Sie ein Ruby-Programm, welches eine Datei `text.ml` aus dem Dateisystem einliest, alle vorkommenden Verweise in die HTML-Notation umwandelt und das Resultat in einer Datei `text.html` speichert. Verweise in HTML haben den folgenden Aufbau:

```
<a href="http://...">Text</a>
```

Die Übersetzung des obigen Textes sollte also den folgenden Text ergeben:

```
Im <a href="http://de.wikipedia.org/wiki/Internet">Internet</a> können viele
interessante <a href="http://de.wikipedia.org/wiki/Internet">Informationen</a>
abgerufen werden. Besonders beliebt für eine erste Orientierung ist die
<a href="de.wikipedia.org">Wikipedia</a>.
```

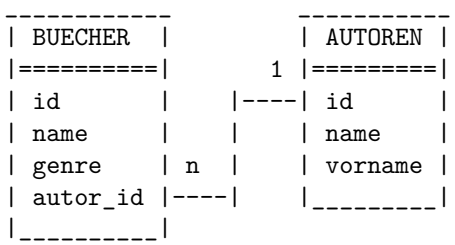
Gehen Sie davon aus, dass sowohl die URL als auch der Beschriftungstext des Verweises beliebige Zeichenfolgen sind, die mit der Regel `Text` ableitbar sind.

Hinweis: Der Reguläre Ausdruck `/[^\])]/` matcht jedes Zeichen außer der schließenden runden Klammer und der schließenden eckigen Klammer.

(c) In Markup-Sprachen können Teile des Textes üblicherweise auch mit zwei Sternen (`**`) umgeben werden, was dafür sorgt, dass der eingeschlossene Text fett gedruckt wird. Erweitern Sie die oben angegebene EBNF um diese Möglichkeit. Achten Sie dabei darauf, dass fett gedruckter Text im Text-Teil von Verweisen (das heißt, zwischen eckigen Klammern) zwar erlaubt ist, in der URL (das heißt, zwischen runden Klammern) allerdings nicht.

Aufgabe 3 - Manuelle Inner Joins 10 Punkte

Zur Verwaltung von Autoren und Büchern ist eine Datenbank mit folgendem Schema gegeben:



Schreiben Sie in Ruby eine Funktion `join_books_and_authors(books, authors)`, welche Bücher-Datensätze mit ihren Autoren verknüpft. Der Inhalt der Büchertabelle wird dazu im zweidimensionalen Array `books` übergeben, der Inhalt der Autorentabelle im zweidimensionalen Array `authors`. Ihr Funktion soll ein neues zweidimensionales Array zurückgeben, welches die folgenden Spalten enthält:

```
BUECHER.id
BUECHER.name
BUECHER.genre
AUTOREN.name
AUTOREN.vorname
```

Beispiel:

```ruby
books = [[13, "1984", "Science Fiction", 13], [24, "Die Farbe der Magie", "Fantasy", 1]]
authors = [[1, "Pratchett", "Terry"], [13, "Orwell", "George"],
           [5, "Tolkien", "John Ronald Reuel"]]
p(join_books_and_authors(books, authors))
# Ausgabe: [[13, "1984", "Science Fiction", "Orwell", "George"],
#           [24, "Die Farbe der Magie", "Fantasy", "Pratchett", "Terry"]]
```

Aufgabe 4 - Erreichbarkeit in Graphen 12 Punkte

Graphen bestehen aus Elementen, genannt *Knoten*, und Verbindungen zwischen den Elementen, genannt *Kanten*. Kanten haben eine Richtung, so dass man sie nur in einer Richtung "entlanglaufen" darf. In Abbildung 1 findet sich ein Beispiel.

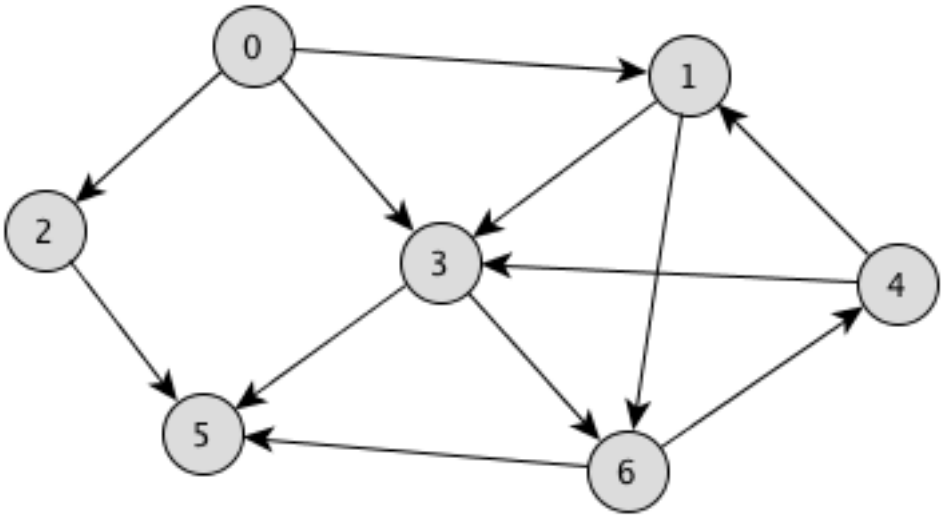


Abbildung 1: Ein Beispielgraph.

In Ruby kann man Graphen über ein zweidimensionales Array `graph` darstellen. Dabei ist `graph[i]` ein eindimensionales Array, welches diejenigen Knoten enthält, die man von Knoten `i` aus direkt über eine gerichtete Kante erreichen kann. Für den Beispielgraphen würde das Array folgendermaßen aussehen:

```
graph = [[1,2,3], [3,6], [5], [5,6], [1,3], [], [4,5]]
```

In diesem Graphen können wir sehen, dass es vom Knoten 6 einen Weg zu Knoten 1 gibt (über den Knoten 4). Um allgemein zu prüfen, ob man von einem gegebenen Startknoten `start` aus einen gegebenen Endknoten `target` erreicht, kann man zum Beispiel folgenden Algorithmus verwenden, den wir hier in Form von Ableitungsregeln angeben:

- Startkonfiguration: $(start, \emptyset)$
- Endkonfiguration: $(target, visited)$, wobei $visited$ eine beliebige Knotenmenge ist
- Ableitungsregel: $(q, visited) \rightarrow (q', visited \cup \{q\})$ falls $q \neq target$ und es eine Kante von q nach q' gibt und $q' \notin visited$

(a) Zeichnen Sie den kompletten Ableitungsbaum zu obigem Graphen für die Startkonfiguration $3, \emptyset$ und $target = 1$.

(b) Programmieren Sie eine Funktion `is_reachable(graph, start, target)`, welche `true` zurückliefert, falls es in dem durch das zweidimensionale Array `graph` definierten Graphen einen Weg vom Knoten mit dem Index `start` hin zu dem Knoten mit dem Index `target` gibt. Falls es keinen Weg zwischen den Knoten gibt, soll die Funktion `false` zurückliefern.

Beispiele:

```
puts(is_reachable(graph, 0, 4))  # -> true
puts(is_reachable(graph, 2, 3))  # -> false
```

Aufgabe 5 - Methodenschreibweise 18 Punkte

(a) Geben Sie folgende Ausdrücke in Methodenschreibweise an:

1. `a + b**2 * 3`
2. `c[3] + c[b - a]`
3. `c[2, 2] = [Math.sqrt(b + c[4])]`

(b) Geben Sie zu den folgenden Aussagen an, ob sie wahr oder falsch sind. Begründen Sie ihre Antwort und geben Sie, wenn möglich, hierzu auch geeignete Belegungen der vorkommenden Variablen an.

1. Objekte der Klasse FixNum können mutiert werden.
2. `str[2,1] = "42"` ist keine Zuweisung.
3. `a[1][3] = 73` besteht aus zwei Methodenaufrufen, von denen einer nicht mutierend und einer mutierend ist.
4. `x = 42` kann auch in der Methodenschreibweise notiert werden.
5. `b[1]["hallo"]` kann den Wert 42 als Ergebnis haben.
6. In Wirklichkeit unterscheidet Ruby gar nicht zwischen Anweisungen und Ausdrücken. Jede Anweisung hat also insbesondere auch einen Rückgabewert.

(c) Wir betrachten das folgende Ruby-Programm:

```
a = [42,42]          #1
a[0] = a             #2

puts(a[0][0][1])     #3
```

Was gibt dieses Programm aus? Erklären Sie, wieso es zu dieser Ausgabe kommt. Skizzieren Sie insbesondere, wie die Objektstruktur im Speicher während der Ausführung des Programms aussieht und verändert wird. Erläutern Sie, was in den einzelnen Schritten passiert, z.B. Methodenaufruf, Zuweisung.

Aufgabe 6 - Geld 14 Punkte

In diese Aufgabe sollen Sie eine Klasse `Geld` zur Repräsentation von Geldbeträgen einer beliebigen Währung definieren.

Wir gehen davon aus, dass der Wert jedes Geldbetrags genau zwei Nachkommastellen hat (zum Beispiel 4,56 Euro oder 0,03 Dollar). Weitere Nachkommastellen sollen nicht möglich sein. Der ganzzahlige Betrag, die Nachkommastelle und die Währung (ein beliebiger String) sollen bei der Konstruktion eines Geldbetrags angegeben werden.

Ihre Klasse soll mindestens die folgenden Methoden zur Verfügung stellen:

- `+` und `-` zum Rechnen mit Geldbeträgen der gleichen Währung. Geldbeträge unterschiedlicher Währung sollen nicht adiert beziehungsweise subtrahiert werden können; in diesen Fällen soll `nil` als Ergebnis zurückgeliefert werden.
- `to_s` soll einen Geldbetrag als String darstellen. Hierbei soll Ihre Klasse gewährleisten, dass tatsächlich exakt zwei Nachkommasstellen dargestellt werden und die Währung mit einem Leerzeichen separiert hinter dem Wert steht. Zwischen ganzzahligem Anteil und den Nachkommastellen setzen Sie bitte ein Komma.

Geben Sie für jede Methode an, ob sie mutierend oder nicht mutierend ist und begründen Sie, weshalb Sie die Methode entsprechend implementiert haben.

Beispiel:

```
e1 = Geld.new(73, 7, "Euro")
e2 = Geld.new(42, 73, "Euro")
d  = Geld.new(100, 0, "Dollar")

puts(e1 - e2)   # 30,34 Euro
puts(d  - e1)   # liefert keine Ausgabe, da das Ergebnis von
                # d - e1 der Wert `nil` ist und dieser mittels
                # to_s zum leeren String umgewandelt wird.
```

Hinweis: Eine interne Darstellung des Betrags als `Float`-Wert ist nicht sinnvoll, da es schwierig ist, hieraus die gewünschte `String`-Darstellung zu erzeugen und auch Rundungsfehler Probleme machen würden. Überlegen Sie sich eine andere Darstellung eines Geldbetrags.

Definieren Sie außerdem außerhalb Ihrer Klasse eine Funktion `to_geld(str)`, welche einen `String` in einen Geldbetrag umwandelt. Gehen Sie davon aus, dass der String die folgende Form hat: beliebige Zahl, gefolgt von einem Komma, exakt zwei Ziffern, gefolgt von einem Leerzeichen. Der restliche String hinter dem Leerzeichen wird dann als Währung interpretiert. Mit Hilfe dieser Prozedur können Sie Geldbeträge einfacher konstruieren, zum Beispiel für `e1` und `d` von oben:

```
e1 = to_geld("73,42 Euro")
d  = to_geld("100,00 Dollar")
```