Algorithmieren und Programmieren

Sommersemester 2023

Lehrstuhl Programmiersprachen und Compilerbau
Prof. Dr. rer. nat. habil. Petra Hofstedt
Ilja Becker
Andreas Eberle, Jan Robert Meyer, Oliver Pflug

Brandenburgische
Technische Universität
Cottbus - Senftenberg

Übungsblatt 11 - Zusatzaufgaben Graphen

Abgabedatum: 16.07.2023

Hinweise

- Beachten Sie, dass die Tutoren unkommentierte Programme nicht als Lösung akzeptieren, selbst wenn die Programme richtig funktionieren. Zu einer richtigen Lösung gehören immer sinnvolle Kommentare, deren Umfang der Komplexität des Programms angemessen ist.
- **Halten Sie sich an die in den Übungsblättern vorgegebenen Namen von Funktionen und Dateien, Funktionstypen (Typsignaturen), Reihenfolge der Parameter und verwenden Sie — sofern vorhanden — die Vorgaben!**
- Auf den Übungsblättern finden Sie einige Haskell-Quelltextfragmente. Diese sind, der besseren Lesbarkeit wegen, unter Nutzung einiger mathematischer Sonderzeichen wiedergegeben.
- Für diese Veranstaltung wird die Verwendung des ***Glasgow Haskell Compiler (GHC)*** (`https://www.haskell.org/`) empfohlen. Weitere Informationen finden Sie im Software-Reiter auf Moodle.
- Als **Tutoriumsaufgabe** markierte (Teil-)aufgaben werden in den Übungen ausführlicher besprochen. Die schriftlichen bzw. elektronischen Lösungen müssen jedoch trotzdem mit abgegeben werden. Bitte schauen Sie Sich diese Aufgaben im Vorfeld der Übung an und bereiten Sie Sich darauf vor.
- Die Abgabe Ihrer Lösungen erfolgt vor Ablauf der Abgabefrist digital über die Moodle-Plattform an Ihren Tutor. Erstellen Sie dazu ein PDF-Dokument, das die Lösungen Ihrer schriftlichen Aufgaben enthält. Laden Sie dieses PDF-Dokument und die erzeugten Hakell-Dateien, mit den in den Aufgaben vorgegebenen Namen, bei Moodle hoch.

Informationsquellen

- Sie finden unter `http://haskell.org/` sehr viele Informationen über die Programmiersprache Haskell. Von besonderem Interesse sind dabei sicherlich die Übersicht über zahlreiche online verfügbare Haskell-Tutorials (`https://wiki.haskell.org/Tutorials`) sowie die Suchmaschine Hoogle (`http://hoogle.haskell.org/`) für die Haskell-API, die Ihnen mit zunehmender Haskell-Erfahrung wertvolle Dienste leisten wird.

Sie können maximal **(5 Zusatzpunkte)** mit diesen zusätzlichen Aufgaben erreichen.

Bei Zusatzaufgaben gibt es keine Möglichkeit zur Nachbearbeitung.

Aufgabe 1 (Implementierung von Graphen mittels Adjazenzmatrizen) 2 Punkte

Bitte die Vorgabe beachten!

Verwenden Sie für die Abgabe den Dateinamen: `AdjMatrixGraph.hs`

In der Vorlesung haben Sie die Implementierung eines abstrakten Datentyps für Graphen basierend auf einem Array von Adjazenzlisten kennen gelernt. In dieser Aufgabe sollen Graphen auf Basis von Adjazenzmatrizen implementiert werden. Dabei legen wir vereinfachend fest, dass nur gerichtete Graphen dargestellt werden sollen.

Der Datentyp `Graph` ist folgendermaßen definiert:

```
data Graph knoten gewicht = Graph (Array (knoten → knoten) (Maybe gewicht))
```

Ein Graph wird also als zweidimensionales Array von Werten des Typs `Maybe gewicht` dargestellt. Steht an einem Index `(Knoten1, Knoten2)` der Wert `Nothing`, so geht keine Kante von `Knoten1` nach `Knoten2`. Steht dort dagegen ein Wert `Just gewicht`, so gibt es eine Kante von `Knoten1` nach `Knoten2` mit dem angegebenen Gewicht.

1. Informieren Sie sich in der Haskell-API-Dokumentation über die Operationen der Typklasse **`Ix`** und des Datentyps **`Array`**.
2. Ersetzen Sie in dem vorgegebenen Haskell-Modul `AdjMatrixGraph` die Definitionen der Form $<$ *Bezeichner* $>=$ *undefined* durch die korrekten Definitionen. Beachten Sie, dass sich die in `AdjMatrixGraph` verwendete Schnittstelle geringfügig von der in der Vorlesung definierten unterscheidet.

Aufgabe 2 (Graphtraversierung) 3 Punkte

Verwenden Sie für die Abgabe den Dateinamen: `Travers.hs`

In der Vorlesung haben Sie zwei Verfahren zur Traversierung von Graphen kennen gelernt. Beide Verfahren verfolgen dieselbe prinzipielle Strategie, unterscheiden sich aber in der Datenstruktur, welche für die zu besuchenden Knoten verwendet wird. Diese Datenstruktur ist bei der Tiefensuche ein Stack, bei der Breitensuche eine Queue. Die Typklasse `Sequenz` bezeichnet eine allgemeine Schnittstelle für beide Datenstrukturen. Sie ist wie folgt deklariert:

```
class Sequenz sequenz where
    leer :: sequenz el
    einfuegen :: el → sequenz el → sequenz el
    loeschen :: sequenz el → sequenz el
    naechstesElement :: sequenz el → el
    istLeer :: sequenz el → Bool
```

1. **(1 Pkt.) (Tutoriumsaufgabe)** Implementieren Sie den Datentyp `Stack` mit den üblichen Operationen. Setzen Sie dabei die Stack-Operationen nicht als normale Funktionen um, sondern machen Sie `Stack` zu einer Instanz von `Sequenz`. Dabei soll `einfuegen` die Operation `push`, `loeschen` die Operation `pop` und `naechstesElement` die Operation `top` realisieren.
2. **(1 Pkt.)** Implementieren Sie den Datentyp `Queue` mit den üblichen Operationen. Setzen Sie dabei die Queue-Operationen nicht als normale Funktionen um, sondern machen Sie `Queue` zu einer Instanz von `Sequenz`. Dabei soll `einfuegen` die Operation `enqueue`, `loeschen` die Operation `dequeue` und `naechstesElement` die Operation `from` realisieren.

3. **(1 Pkt.)** Implementieren Sie basierend auf Ihrer Graphimplementierung aus Aufgabe 2 eine Funktion `traversieren` vom Typ

```
traversieren :: (Sequenz sequenz, Ix knoten) ⇒
                        sequenz knoten →Graph knoten gewicht →knoten →[knoten]
```

welche eine leere Sequenz, einen Graphen und einen Startknoten erwartet, den Graphen traversiert und die Liste der besuchten Knoten zurück gibt.

Die Kandidatensequenz soll aus der übergebenen leeren Sequenz konstruiert werden und den gleichen Typ wie diese besitzen. Die Funktion `traversieren` kann damit sowohl eine Tiefen- als auch eine Breitensuche umsetzen, nämlich, indem sie mit einem leeren Stack bzw. einer leeren Queue aufgerufen wird. Das führt zu folgenden Funktionsdefinitionen:

```
tiefensuche :: (Ix knoten) ⇒ Graph knoten gewicht → knoten → [knoten]
tiefensuche = traversieren (leer :: Stack knoten)
breitensuche :: (Ix knoten) ⇒ Graph knoten gewicht → knoten → [knoten]
breitensuche = traversieren (leer :: Queue knoten)
```