Übungsaufgaben zu Funktionale Programmierung vom 13.10.2004, fällig: 20.10.2004
Themen: *Hugs kennen lernen, erste Schritte in Haskell und erste weiterführende Programme*

Allgemeine Hinweise

Sie können die Programmieraufgaben im Labor im Erdgeschoss des Gebäudes Argentinierstraße 8 mit den dort befindlichen Rechnern (im Rahmen der Kapazitäten) bearbeiten und lösen. Sie erreichen dieses Labor über den kleinen Innenhof. Einen Lageplan finden Sie unter der URL
`http://www.complang.tuwien.ac.at/ulrich/p-1851-E.html`.
Um die Aufgaben zu lösen, rufen Sie bitte den Hugs 98-Interpretierer durch Eingabe von `hugs` in der Kommandozeile einer Shell auf. Falls Sie die Übungsaufgaben auf Ihrem eigenen Rechner bearbeiten möchten, müssen Sie zunächst Hugs 98 installieren. Hugs 98 ist beispielsweise unter `http://www.haskell.org/hugs/` für verschiedene Plattformen verfügbar. Der Aufruf der jeweiligen Interpretierervariante ist dann vom Betriebssystem abhängig.

On-line Tutorien zu Haskell und Hugs

Unter `http://www.cse.ogi.edu/PacSoft/projects/Hugs/pages/hugsman/index.html` finden Sie ein Online-Manual für Hugs 98. Verschiedene ausdruckbare Versionen des Manuals finden Sie auch unter `http://www.cse.ogi.edu/PacSoft/projects/Hugs/pages/downloading.htm`. Lesen Sie die ersten Seiten des Manuals. In jedem Fall sollten Sie mindestens Kapitel 3 zum Thema „Hugs for beginners“ lesen und die darin beschriebenen Beispiele ausprobieren. Machen Sie sich so weit mit dem Haskell-Interpretierer vertraut, dass Sie problemlos einfache Ausdrücke auswerten lassen können.

Ein on-line Tutorial zu Haskell selbst finden Sie hier: `http://haskell.org/tutorial/`.

Abgabe und erreichbare Punkte

Zum Zeitpunkt der Abgabe (Mi, 20.10.2004, 12 Uhr pünktlich) **und** der nachträglichen Abgabe (Di, 27.10.2004, 12 Uhr pünktlich) soll eine Datei namens `Aufgabe1.hs` mit Ihren Lösungen der Aufgaben in Ihrem Home-Verzeichnis (**keinesfalls** in einem Unterverzeichnis) auf dem Übungsrechner (b1) stehen. Von dort aus wird sie zu den genannten Zeitpunkten automatisch kopiert.

Für das erste Aufgabenblatt sind insgesamt 100 Punkte zu erreichen.

Vorsicht: Klippen!

Die Syntax von Haskell birgt im Großen und Ganzen keine besonderen Fallstricke und ist zumeist intuitiv, wenn auch im Vergleich zu anderen Sprachen gelegentlich “gewöhnungsbedürftig”. Eine Hürde für Programmierer, die neu mit Haskell beginnen, sind Einrückungen. Einrückungen tragen Bedeutung in Haskell für die Bindungsbereiche und müssen daher unbedingt eingehalten werden. Alles, was zum selben Bindungsbereich gehört, muss in derselben Spalte beginnen. Diese in ähnlicher Form auch in anderen Sprachen wie etwa `Occam` vorkommende Konvention erlaubt es, Strichpunkte und Klammern einzusparen. Ein Anwendungsbeispiel in Haskell: Wenn eine Funktion mehrere Zeilen umfasst, muss alles, was nach dem „=“ steht, in derselben Spalte beginnen, oder noch tiefer eingerückt sein als in der ersten Zeile. Anderenfalls liefert Hugs dem Haskell-Programmierbeginner (scheinbar) unverständliche Fehlermeldungen wegen fehlender Strichpunkte (obwohl Strichpunkte, soweit wir Haskell bis jetzt kennen gelernt haben, gar nicht vorkommen). Weiterhin sollen in Haskellprogrammen alle Funktionsdefinitionen und Typdeklarationen in derselben Spalte (also ganz links) beginnen. Verwenden Sie keine Tabulatoren, oder stellen Sie die Tabulatorgröße auf acht Zeichen ein. Achten Sie auf richtige Klammerung. Haskell erfordert oft keine Klammern, da sie gemäß geltender Prioritätsregeln automatisch ergänzt werden können. Im Zweifelsfall ist es gute Praxis ggf. überflüssig zu klammern, um “Überraschungen” zu vermeiden. Beachten Sie, dass außer „−“ (Minus) alle Folgen von Sonderzeichen als Infix- oder Postfix-Operatoren interpretiert werden; Minus wird als Infix- oder Prefix-Operator interpretiert. Achtung: Der Funktionsaufruf „`potenz 2 -1`“ entspricht „`potenz 2 - 1`“, also „`(potenz 2) - (1)`“ und nicht, wie man erwarten könnte, „`potenz 2 (-1)`“. Operatoren haben immer eine niedrigere Priorität als das Hintereinanderschreiben von Ausdrücken (Funktionsanwendungen). Ein “Underline” (_) zählt zu den Kleinbuchstaben. Wie gesagt, wenn Sie unsicher sind, verwenden Sie lieber mehr Klammern als nötig. Funktionsdefinitionen und Typdeklarationen können Sie nicht direkt im Haskell-Interpretierer schreiben, sondern nur von Dateien laden. Genauere Hinweise zur Syntax finden Sie unter `http://haskell.org/tutorial/`.

Aufgaben

Für dieses Aufgabenblatt sollen Sie die unten angegebenen Aufgabenstellungen in Form eines "traditional Haskell script" lösen und in einer Datei mit Namen `Aufgabe1.hs` in Ihrem home-Verzeichnis ablegen. Kommentieren Sie Ihre Programme aussagekräftig und machen Sie sich so auch mit den unterschiedlichen Möglichkeiten vertraut, ihre Entwurfsentscheidungen in Haskell-Programmen durch zweckmäßige und (Problem-) angemessene Kommentare zu dokumentieren. Benutzen Sie, wo sinnvoll, Hilfsfunktionen und Konstanten.

Versehen Sie insbesondere alle Funktionen, die Sie zur Lösung der Aufgaben brauchen, auch mit ihren Typdeklarationen, d.h. geben Sie deren Signatur explizit an. Laden Sie Ihre Datei anschließend mittels „`:load Aufgabe1`" (oder kurz „`:l Aufgabe1`") in das Hugs-System und prüfen Sie, ob die Funktionen sich wie erwartet verhalten. Nach dem ersten erfolgreichen Laden können Sie Änderungen der Datei mittels `:reload` oder `:r` aktualisieren. Im einzelnen sollen Sie folgende Problemstellungen durch geeignete Haskell-Rechenvorschriften lösen. Sie brauchen keine Fehlerbehandlung vorzusehen. Sie können davon ausgehen, dass die Funktionen nur mit "passenden" Argumenten getestet werden.

1. Eine natürliche Zahl n, $n \geq 1$, heißt *Primzahlzwilling*, wenn mit der Zahl n auch die Zahl $n+2$ oder die Zahl $n-2$ eine Primzahl ist. 1 ist keine Primzahl. Schreiben Sie eine Wahrheitswertfunktion (oder kürzer: Prädikat) `isPrimeTwin` mit der Signatur `isPrimeTwin :: Integer -> Bool`, die angewendet auf ein ganzzahliges Argument `n` den Wahrheitswert `True` liefert, wenn `n` ein Primzahlzwilling ist, `False` sonst.

2. Eine natürliche Zahl n, $n \geq 1$, heißt *vollkommen*, wenn sie gleich der Summe ihrer von sich selbst verschiedenen (positiven) Teiler ist. So ist z.B. die natürliche Zahl 6 vollkommen, da sich 6 als Summe ihrer Teiler 1, 2 und 3 ergibt. 4 andererseits ist nicht vollkommen, da die Summe von 1 und 2 on 4 verschieden ist. Schreiben Sie ein Prädikat `isPerfect :: Integer -> Bool`, das angewendet auf ein ganzzahliges Argument `n` den Wahrheitswert `True` liefert, wenn `n` vollkommen ist, `False` sonst.

3. Die Zahlen $F_k =_{df} 2^{2^k} + 1$, $k \in I\!N$, $k \geq 0$, heißen *Fermatsche Zahlen*. Schreiben Sie eine Haskell-Rechenvorschrift `fermat` zur Berechnung der i-ten Fermatschen Zahl. Dabei ist F_0 die 1. Fermatsche Zahl, F_1 die zweite, F_2 die dritte usw. Benutzen Sie als Datentyp für Argument und Resultat der Funktion `fermat` `Integer`. Das Symbol "$=_{df}$" steht in dieser Aufgabe für *definitionsgemäß gleich*. Wir verwenden es hier (und auch im weiteren Verlauf der Lehrveranstaltung) anstelle des in der Mathematik gebräuchlichen Symbols "`:=`", da letzteres in Programmiersprachen mit Zuweisungen oft den Zuweisungsoperator bezeichnet.

4. Der Wert der Exponentialfunktion an der Stelle x lässt sich als Wert der folgenden unendlichen Reihe darstellen:

$$e^x = 1 + \frac{x}{1\,!} + \frac{x^2}{2\,!} + \ldots + \frac{x^k}{k\,!} + \ldots$$

   Schreiben Sie eine Haskell-Rechenvorschrift `expApprox :: Float -> Integer -> Float` zur näherungsweisen Berechnung des Funktionswerts der Exponentialfunktion an der Stelle x. Dabei ist durch das erste Argument die Stelle gegeben, durch das zweite Argument die Anzahl der für die Approximation aufzusummierenden Summenglieder. Ist das zweite Argument i kleiner oder gleich 0, soll die Funktion `expApprox` als Resultat 0 liefern, ansonsten die Summe der ersten i Summenglieder.

Der Aufruf `expApprox 2 (-5)` liefert somit den Wert 0, der Aufruf `expApprox 2 1` den Wert 1 und der Aufruf `expApprox 2 2` den Wert 3.