## Einführung in Haskell - Hugs

Höhere Programmiersprachen lassen sich in in deklarative (funktional, logisch), imperative (Pascal, Modula ) und objektorientierte Sprachen (Java, Python .. ) einteilen.
Imperative Sprachen spiegeln dabei aufgrund der Verwendung des Variablenkonzeptes die Architektur des Von - Neumann - Rechners wider. Ein Programm stellt hier eine Menge von Zustandsübergängen dar. Ein solcher Übergang wird durch eine Anweisung (Befehl) erreicht. Demgegenüber dient ein funktionales Programm im Wesentlichen der Berechnung des Ergebnisses einer oder mehrerer Funktionen. Damit stellt ein Programm hier eine Menge von Funktionsdefinitionen und Funktionsaufrufen, die in sich verschachtelt sein können, dar. Logische Programmierung basiert auf prädiktiver Logik und enthält als Grundlage für logische Schlußfolgerungen Fakten und Regeln. In jüngerer Zeit werden zunehmend objektorientierte Sprachen für Implementierungen verwendet. Trotzdem es mit Modula - 2, C ++ hybride Sprachen gibt, die neben dem prozeduralen (imperativen Ansatz ) auch die Möglichkeit der Objektorientierung beinhalten, werden rein objektorientierte Sprachen wie Java zunehmend verwendet. Die grundlegenden Konzepte der Informatik lassen sich mit allen Ansätzen erfassen.
Haskell (Hugs) zählt zu den funktionalen Programmiersprachen und ist plattformunabhängig. Das Programm ist sowohl unter unixoiden (Linux), als auch in der Windowswelt einsetzbar. Während `winhugs` typischerweise über ein Icon gestartet werden kann, ist die Arbeit auf Konsolenebene für z.B. Linux typisch. In beiden Fällen meldet sich Hugs mit der Versionsnummer des Programms und es erscheint der Haskellpromt. Im Folgenden das Erscheinungsbild bei einer Linuxsitzung.

```
Hugs session for:
/usr/lib/hugs/lib/Prelude.hs
Type :? for help
Prelude >
```

Damit wird die Standardbibliothek geladen, die mit

```
Prelude > :e
```

im entsprechenden Editor (Windows : Notepad, Linux : vi [1]) eingesehen werden kann. Eine Änderung der Defaulteinstellungen sollte in jedem Fall unterbleiben. Um mit Haskell zu arbeiten, reichen folgende Befehle völlig aus.Dabei ist zu beachten, dass Haskellprogramme die Extension `.hs` haben und unter dieser auch abgespeichert werden.

```
Prelude > :e Test.hs -- öffnet den Editor für die Arbeit mit Test.hs
Prelude > :l Test.hs -- lädt das Programm und compiliert es
Prelude > :r Test.hs -- reload der Datei
Prelude > :l ‘‘Verzeichnis/Test.hs‘‘
Prelude > :v -- liefert die Versionsnummer
Prelude > :? -- liefert die Liste aller Befehle
Prelude > :q -- beendet Hugs
```

## Ein erstes Haskellprogramm

Nachdem in das Arbeitsverzeichnis für Haskell gewechselt wurde, wird das Programm im Editor geöffnet, editiert und abgespeichert. Als Beispiel dient hier die Datei `hello.hs`

```
1 -- Testprogramm Hugs
2
3 add a b = a + b
4 lies a = putStr a
5 and a b = a && b
```

Die Angabe des Programmierers und das Datum der Erstellung sind als Kommentar auszuweisen.

[1] Ein Defaulteinstellung des Editors ist unter unixoiden Systemen eventuell nachzuholen.
Dazu ist mit `hugs -E‘‘vi +%d %s ‘‘` zu starten oder der Flag in die `.bashrc` zu integrieren.

Nach dem Start von Hugs und dem Lesen des Programms

```
Prelude > :l hello.hs
```

meldet sich nach erfolgreicher Compilierung das Programm.

```
Hugs session for:
/usr/lib/hugs/lib/Prelude.hs
Type :? for help
Prelude > :l hello.hs
Reading file ``hello.hs'':

Hugs session for:
/usr/lib/hugs/lib/Prelude.hs
hello.hs
Main >
```

Wird am Haskellpromt `Main` ausgewiesen, war die Compilierung erfolgreich . Die enthaltenen Ausdrücke können nun getestet werden. Die als Argumente übergebenen, durch Leerzeichen getrennten Werte werden ausgewertet und das Ergebnis der Auswertung zurückgeliefert. Jeder dieser Werte ist von einem bestimmten Datentyp.

```
Main > add 3 4
7
Main > lies ``Hallo Welt''
Hallo Welt
Main > and True True
True
```

Die Angabe des Datentyps im Programm ist nicht obligat, aber für die bessere Lesbarkeit wünschenswert. Ein Abfrage des jeweiligen Typs ist möglich.

```
Main > :t add
add :: Num a => a -> a -> a
Main > :t lies
lies :: [Char] -> IO ()
Main > :t and
and :: Bool -> Bool -> Bool
```

Ein schon etwas komplexeres Programm hätte nun folgendes Aussehen.

```
1  -- Testprogramm 2
2
3  add :: Num a => a -> a -> a
4  add a b = a + b
5
6  lies :: [Char] -> IO ()
7  lies a = putStr a
8
9  and :: Bool -> Bool -> Bool
10 and a b = a && b
```

**Übung 1**

```
1  -- Umgang mit elementaren Datentypen
2  -- Numerischer Typ
3
4  prod a b = a * b
5  rest a b = mod a b
6  sub a b = a - b
7
8  -- Zeichen und Zeichenketten (Bsp.: 'a', "affe")
9
10 konk a b    = a ++ b
11 kopf a      = head a
12 schwanz a   = tail a
13 umkehr a    = reverse a
14
15 -- Boolsche Werte
16
17 or a b       = a || b
18 neq a b      = a /= b
19
20
21 -- Tupel : Zusammenfassung einer festen Zahl
22 -- unterschiedlichen Typs
23
24 fst (a,b) = a  -- Paare
25 snd (a,b,c) = b -- Tripel
```

**Aufgabe**

(a) Bestimmen Sie die Typen der verwendeten Funktionen von Hand und überprüfen Sie ihre Ergebnisse im Anschluss.

(b) Spezifizieren Sie die im Programm enthaltenen Funktionen.

(c) Geben Sie einen Überblick über die wichtigsten Datentypen und Operatoren.

## Analyse eines Programms

Bei der Berechnung der Nullstellen quadratischer Funktionen der Form $y = ax^2 + bx + c$ verwendet man die sogenannte Normalform.

$$x_{1,2} = -\frac{b}{2a} \pm \sqrt{\frac{b^2}{4a} - \frac{c}{a}} \quad = -\frac{b}{2a} \pm \frac{\sqrt{b^2-4ac}}{2a} \quad \text{wobei} \qquad D = b^2 - 4ac$$

Die Beschaffenheit der Diskriminante D gibt dabei die Anzahl der potenziellen Lösungen der quadratischen Gleichung an. Daraus ergibt sich das folgende Haskellprogramm.

```
1  -- Testprogramm 3 (lokale Definitionen, where Klausel)
2
3  quad (a,b,c) =
4      if dis == 0.0 then (x0,x0) else
5      if dis > 0.0 then (x0+r, x0-r) else error "keine␣Loesung"
6
7      where dis  = b*b - 4.0*a*c
8            r    = (sqrt dis)/(2.0*a)
9            x0   = - b/(2.0*a)
```

In diesem Programm werden elementare Fälle über if-then-else Konstrukte abgeprüft. Da jedes Mal die boolschen Bedingungen die Evaluation gleicher Ausdrücke erwarten, führt man eine `lokale Definition` dieser Ausdrücke ein, welche durch `where` eingeleitet werden.

Die Sequenz kann aber noch klarer durch die Verwendung von sogenannten Guards (Guarded Equations) gestaltet werden. Die einzelnen Fälle der Sequenz werden durch | eingeleitet und bestehen aus dem `Guard - Wächter`,welcher die boolsche Bedingung beinhaltet, und dem auszuwertenden Ausdruck der durch = von der Bedingung getrennt ist. Wird keiner der `Guards` zu True [2] ausgewertet, werden alle anderen potenziellen Fälle durch den `otherwise` Fall abgefangen, welcher stets zu True auswertet. Daraus ergibt sich die folgende veränderte Implementierung.

```
1  -- Testprogramm 4 (Verwendung von Guards)
2
3  quad :: (Ord a, Floating a) => (a,a,a) -> (a,a)
4
5  quad (a,b,c)
6      | dis == 0.0 = (x,x)
7      | dis > 0.0 = (x+r, x-r)
8      | otherwise = error "keine␣Loesung"
9
10           where dis = b*b - 4.0*a*c
11                 r   = (sqrt dis)/(2.0*a)
12                 x   = - b/(2.0*a)
```

**Übung 2**

(a) Berechnen Sie den Rückgabewert von `Testprogramm 4` für den Aufruf `quad (1,5,1)` von Hand. Notieren Sie die einzelnen Berechnungsschritte.

(b) Schreiben Sie eine Funktion welches testet, ob es sich bei dem übergebenen Argument um ein Schaltjahr handelt.
(Aufruf `schaltjahr 2000 -> True`)

(c) Schreiben Sie eine Funktion `maximum`, welche mittels Guards das Maximum 3er Zahlem liefert.
(Aufruf `maximum 2 5 6 -> 6`)

(d) Bei Skatspielen findet man als Beiblatt oft folgende Auflistung.

| Teilnehmer | A | B | C | D |
|---|---|---|---|---|
| Endzahlen der Liste | +196 | +32 | -12 | +80 |
| Vergleichsrechnung | +164 | -164 | -208 | -116 |
| | +208 | +44 | -44 | +48 |
| | +116 | -48 | -92 | +92 |
| Cent | +488 | -168 | -344 | +24 |

Erläutern Sie das Abrechnungsprinzip und erstellen Sie ein Haskellprogramm welches die Abrechnungsfunktionalität leistet. Die Implementierung einer Showfunktion ist dabei optional.

[2] True - boolscher Ausdruck für wahr

## Erweiterte Datenstrukturen

Die Zusammenfassung von Daten, die inhaltlich zusammengehören werden in objektorientierten Sprachen über das Klassenkonzept realisiert. Ein Beispiel hierfür wäre ein Punkt, die durch 2 Gleitkommazahlen eindeutig erfasst wird. In Haskell ist die Vorgehensweise konzeptionell ähnlich.

## Typdefinitionen

- `Typsynonyme`

  Typsynonyme erhöhen die Lesbarkeit von Programmen. So könnte zum Beispiel der der Typ auto als Typel aufgefasst werden, welches Alter, Name und gefahrene Kilometer des Porsche enthält. Um nicht jedesmal das vollständige Typel in der Typdeklaration auszuweisen, könnte man nachstehendes Synonym verwenden.

  Beispiel : `type auto = (Int, String, Int)`

- `Algebraische Datentypen`

  Algebraische Datentypen finden unter z.B Anwendung bei Aufzählungen (`data Bool = True| False`, Verbindungen (`data Punkt = P Float Float`) gleicher oder differierender Basistypen. Rekursive Datentypen wie Listen oder Bäume sind ebenso algebraisch erfassbar. Die Grundstruktur eines algebraischen Datentyps stellt sich vereinfach wie folgt dar.

  Beispiel (Verbundtyp) : `data` $\underbrace{Punkt}_{Datentyp}$ `=` $\underbrace{P}_{Konstruktor}$ `Float Float`

Die Datei geometrie.hs als Beispiel für die Arbeit mit algebraischen Datentypen.

```
1  -- Testprogramm 5 (Algebraische Datentypen)
2
3  data Punkt = P Float Float
4  data Kreis = K Punkt Punkt
5
6  quad :: Float -> Float -- Hilfsfunktion
7  quad x = x * x
8
9  entfernung :: Punkt -> Punkt -> Float
10 radius     :: Kreis -> Float
11 flaeche    :: Kreis -> Float
12
13 entfernung (P x0 y0) (P x1 y1) = sqrt (quad (x1-x0)+(quad (y1-y0)))
14 radius (K (P xm ym) (P xk yk)) = entfernung (P xm ym) (P xk yk)
15 flaeche (K (P xm ym) (P xk yk)) = pi * quad (radius (K (P xm ym) (P xk yk)))
16
17 kreis1 = K (P 0 0) (P 5 0) -- Testbelegung
```

Die Sitzung liefert folgende Ausgabe.

```
Hugs session for:
/usr/lib/hugs/lib/Prelude.hs
geometrie.hs
Main > radius kreis1
5.0
Main > flaeche kreis1
78.5398
```

**Übung 3**

(a) Erweitern Sie die Datei `geometrie.hs` um die Methode Umfang.

(b) Entwerfen Sie einen algebraischen Datentyp `Dreieck`, spezifizieren und implementieren Sie die Methoden `hoehe` und `flaeche` in der Datei `geometrie.hs` .

(c) Entwerfen Sie ein Modul Kalender auf der Grundlage des Datentyps

```
data Datum = Date Int String Int.
```

Die enthaltenen Methoden sollen wenigstens `monat`, `tag`, `jahr` und `schaltjahr` umfassen.

## Integrierte Datenstrukturen

### Listen

Die wohl wichtigste in Haskell integrierte Datenstruktur stellt die Liste dar. In einer Liste sind Elemente gleichen Datentyps enthalten, damit erinnert die Liste stark an den mathematischen Mengenbegriff . In einer Liste können aber im Gegensatz zu einer Menge gleiche Elemente mehrfach vorkommen. Darüber hinaus unterliegen die Elemente einer Liste einer Ordnungsrelation.

Beispiele :

```
zahlenliste = [1,2,3,4]
zeichenliste = ['a','b','c']
Listenliste = [[1,9],[2,6],[3,4]] ..
```

Die allgemeine Darstellung für nichtleere Listen lautet (x:xs). (:) stellt dabei einen Operator (cons) dar, der ein Element an das vordere Ende der Liste anfügt.

Beispiele :

"hallo "= 'h': "allo "
$[1, 4, 7, 6] = 1 : [4, 7, 6]$

Im Standardprelude sind verschiedene Listenfunktionen definiert. Um auf Teile einer Liste zuzugreifen, ist deren Traversierung bzw. teilweise Traversierung in den meisten Fällen erforderlich. Der erforderliche Durchlauf erfolgt rekursiv. Dazu folgende Beispiele.

**Länge einer Liste**

```
1 length' []  = 0
2 length' (x:xs) = 1 + length' xs
```

**Auswertung**

rekursiver Abstieg
`length` $[1, 2, 3] \rightsquigarrow$ `1 + length` $[2, 3] \rightsquigarrow$ `1+1+length` $[3] \rightsquigarrow$ `1+1+1+ length` $[] \rightsquigarrow$

Terminationsfall : `length` $[]$ `= 0`

rekursiver Aufstieg
`0 + 1 + 1 + 1 = 3`

**Konkatenation zweier Listen**

```
1 concat' :: [a] -> [a] -> [a]
2 concat' [] ys  = ys
3 concat' (x:xs) ys = x : concat' xs ys
```

**Auswertung**

rekursiver Abstieg
`concat'` $[1,2][3,4] \rightsquigarrow$ `1: concat'`$[2][3,4] \rightsquigarrow$ `1:2: concat'`$[][3,4]$

Terminationsfall : `concat'` $[][3,4]$ `=` $[3,4]$

rekursiver Aufstieg
$[3,4] \rightsquigarrow [2,3,4] \rightsquigarrow [1,2,3,4]$

**Übung 4**

(a) Implementieren Sie die Funktionen take', drop' und werten Sie die Methoden für entsprechende Beispiele aus. Verwenden Sie als Hilfe das Prelude.

(b) Implementieren Sie folgende Listenfunktionen einschließlich der Spezifikation:

- `zensur` : ersetzt alle 'r' durch ein 'X'
  Aufruf $\rightsquigarrow$ zensur "riesenerind "$\mapsto$ "XiesenXind "
- `doppelt`: verdoppelt alle Buchstaben
  Aufruf $\rightsquigarrow$ doppelt "wasser "$\mapsto$ "wwaasssseerr "
- `loesche`: loescht übergebenes Element aus der Liste
  Aufruf $\rightsquigarrow$ loesche 'k' "kuemel "$\mapsto$ "uemel "

## Abstrakte Datentypen

Mitunter ist es erforderlich, eigene Module vor allem im Zusammenhang mit der Erstellung größerer Softwareprojekte zu schaffen. Dadurch wird eine arbeitsteilige Vorgehensweise ermöglicht. Hauptaugenmerk liegt bei solchen Entwicklungsaufgaben auf der `Datenabstraktion`.

Die grundlegenden Prinzipien der Datenabstraktion sind folgende.

- `Geheimnisprinzip`

  Nach `Parnas (1972)` sollten die nicht benötigten Teile eines Programms unsichtbar sein, die eigentliche Implementierung des Datentyps wird also versteckt. Es ist nur möglich von außen über Operationen auf diesen Datentyp zuzugreifen. Dieses Prinzip findet in allen Softwareprojekten auch heute Verwendung und wird als `Geheimnisprinzip (information hiding principle)` bezeichnet.

- `Kapselung`

  Die Datentypen werden nur über ihre Schnittstelle benutzt, wobei die Schnittstelle die bereitgestellten Funktionen beinhaltet. Dabei ist es durchaus denkbar, Hilfsfunktionen zu verdecken.

Die Vorteile dieser Vorgehensweise sind

- `Sicherheit` Eine Veränderung des Objektes ist unmöglich, da ein Zugriff nur über Operationen möglich ist.
- `Flexibilität` Die Implementierung kann unter der Oberfläche weiterentwickelt oder geändert werden, ohne die Funktionalität zu beeinflussen.

Das folgende Beispiel verdeutlicht die Arbeit mit abstrakten Datentypen. In einem ersten Schritt wird das Modul Punkt1.hs erstellt, welches die Schittstelle, den Datentyp `Punkt` und die entsprechende Implementierung beinhaltet. Es werden nur die Operatioen exportiert, die Repräsentation derselben erfolgt über die Schnittstelle, der Datentyp bleibt verdeckt. Ersichtlich ist dies an der Exportliste, welche den Datentyp Punkt nicht enthält. Damit handelt es sich um einen, wenn auch überschaubaren ADT (Abstrakten DatenTyp). Dieser kann nun importiert werden und ist über seine Operationen erreichbar.

```
1  module Punkt1 (setpoint, getx, gety)
2  where
3
4        -- Schnittstelle (Signatur und Spezifikation der Operatoren)
5
6        setpoint :: Float -> Float -> Punkt
7        getx   :: Punkt -> Float
8        gety   :: Punkt -> Float
9
10       -- Datentyp (Modell)
11
12       data Punkt = P Float Float
13               -- algebraischer Verbundtyp Punkt
14
15
16       -- Implementierung
17       setpoint x y = P x y
18       getx (P x y) = x
19       gety (P x y) = y
20
21
22       instance Show Punkt where
23              show (P x y)="("++show x++","++show y++")"
```

Die Verwendung des ADT's erfolgt durch einen Import, im Beispiel die Datei geo1.hs

```
1  import Punkt1
2
3  neuer_Punkt a b = setpoint a b
```

welche folgende Ausgabe hervorruft

```
Hugs session for:
/usr/lib/hugs/lib/Prelude.hs
Punkt1.hs
geo1.hs
Main > neuer_Punkt 1 2
(1.0, 2.0)
```

Es ist möglich den Datentyp zu exportieren und damit in einem weiteren Modul auf diesen zuzugreifen. Das Modul Kreis.hs erbt in diesem Fall gewissermaßen sowohl den Datentyp Punkt als auch die darauf basierenden Operationen. Kreis wiederum exportiert den Datentyp nicht und ist damit von außen nur über die Operation erreichbar. Zusätzlich ist hier erkennbar, dass die Hilfsoperation `quad` nicht exportiert wird, da sie für die Schnittstelle sekundär ist und für Funktionalität des Programms nur eine untergeordnete Bedeutung hat. Das Prinzip der Kapselung und eine schmale Datenkopplung sind dadurch erreicht.

Folgendes Bild ergibt sich insgesamt.

Modul Punkt

```
1 module Punkt
2 where
3
4       -- Schnittstelle (Signatur und Spezifikation der Operatoren)
5
6       setpoint :: Float -> Float -> Punkt
7       getx   :: Punkt -> Float
8       gety   :: Punkt -> Float
9
10      -- Datentyp (Modell)
11
12      data Punkt = P Float Float
13              -- algebraischer Verbundtyp Punkt
```

Modul Kreis

```
1 module Kreis (entfernung)
2 where
3
4 import Punkt
5
6 entfernung :: (Float,Float) -> (Float,Float) -> Float
7 quad     :: Float -> Float
8
9 data Kreis = K Punkt Punkt
10
```

Hauptprogramm

```
1 import Kreis
2
3 dist (x0,y0) (x1,y1) = entfernung (x0,y0) (x1,y1)
```

**Übung 4**

(a) Entwerfen Sie arbeitsteilig einen ADT Stack und einen ADT Queue. Implementieren Sie je eine Anwendung auf diesen ADT's.

(b) (Zusatz)

Die vorstehende Vorgehensweise der Erstellung eines ADT's bezeichnet man als `modellierende Spezifikation`.
Informieren Sie sich über das Vorgehen, um eine algebraische Spezifikation zu erstellen. Spezifizieren Sie den ADT Stack auf diese Weise.