Kapitel 6
Abstrakte Datentypen

6.1 Die natürlichen Zahlen

6.2 Lineare Datenstrukturen: Keller, Schlangen, Reihungen, Dateien

6.3 Binärbäume

6.4 Mengen und Mehrfachmengen

Funktionsabstraktion:

- Eine Funktion `f m = a` wird durch einen Aufruf `f x` angesprochen
  - mit einer Zusicherung {P} a {Q} kann man das Ergebnis des Aufrufs bestimmen, ohne die Einzelheiten des Funktionsrumpfs a zu kennen:
    - es gilt {Pσ} aσ {Qσ} mit der Substitution σ = [m / x], Qσ beschreibt das Ergebnis
- wenn die Funktionsdefinition von Dritten stammt oder in einer anderen Programmiersprache oder nur binär codiert vorliegt, kann man sie ohne Kenntnis des Rumpfs benutzen, wenn die Zusicherung bekannt ist

Datenabstraktion:

- Gegeben eine Datenstruktur DS, z.B. eine Liste, ein Keller, ein gerichteter Graph, eine Menge:
  - muß man die rechnerinterne Repräsentation von DS und die Realisierung der Grundoperationen auf DS kennen, um DS zu benutzen?
  - nein! Es genügt, wenn man das Ergebnis einer Operation auf DS aufgrund der vorangegangenen Operationen vorhersagen kann

Beispiel:

- wenn man ein Element x in einen Keller steckt (mit `push k x`), ist es gleichgültig, wo es sich befindet, wichtig ist allein,
  - daß eine anschließende top-Operation (Projektion) x zurückliefert
  - daß eine anschließende Abfrage isEmpty (ebenfalls eine Projektion) den Wert False liefert

Beispiel:

- wir benutzen Datenbanken von Universitäten, IBM, Oracle, ..., ohne zu wissen, wie sie realisiert sind
- wichtig ist allein, wie die Definition von Relationen und Tupeln solcher Relationen das Ergebnis zukünftiger SQL-Abfragen beeinflußt
  - auch die SQL-Abfragen sind Projektionen

6 Datenabstraktion: Prinzipien

- Grundlage der Datenabstraktion ist der Begriff **abstrakter Datentyp (ADT)** aus Kap. 3: Quotient einer freien Termalgebra nach einer Äquivalenzrelation $\mathcal{A} / \equiv$
  - mit **Konstruktoren** steckt man Werte x in die Datenstruktur
    - ihre Realisierung ist uninteressant, wichtig ist ihre Wirkung auf zukünftige Projektionsoperationen
  - **Hilfskonstruktoren** erlauben die Konstruktion weiterer Werte *ds* der Datenstruktur, die aber gemäß der Äquivalenzrelation gleichwertig zu Werten sind, die nur Konstruktoren benutzen
  - **Projektionen** erteilen Auskunft über Eigenschaften der Datenstruktur und über Werte in der Datenstruktur
    - das Ergebnis der Projektionen **muß** aufgrund vorangegangener (Hilfs-)Konstruktoroperationen vorhersagbar sein
- Die Äquivalenzrelation eines ADT ist durch die Gesetze des ADT, d.h. durch Termersetzungsregeln, bestimmt
- **Datenabstraktion: die Signaturen und Gesetze des ADT genügen, um Ergebnisse zu bestimmen, die Kenntnis der Implementierung ist nicht nötig**

6 Geheimnisprinzip, Module

Parnas 1972

- Ein Programmstück heißt ein Modul (oder eine Komponente), wenn
  - es eine logisch und funktional zusammenhängende Aufgabe löst,
  - wenn es eine Schnittstelle, bestehend aus Funktionssignaturen und Zusicherungen über diese Funktionen besitzt, so daß
    - der Modul von Dritten nur unter Kenntnis der Schnittstelle benutzt werden kann, ohne Kenntnis der Implementierung
    - der Modul implementiert werden kann nur unter Kenntnis der Schnittstelle, ohne Kenntnis seiner Verwendung(en)
- Der Modulbegriff ist grundlegend für die arbeitsteilige Konstruktion größerer Systeme, nicht nur in Software, sondern auch in Hardware und anderen Bereichen des Ingenieurwesens
- Die strikte Trennung von Implementierung und Verwendung heißt Geheimnisprinzip (die Implementierung geheimhalten) oder Kapselung (der Implementierung gegen die Benutzung)

  information hiding
- Das Geheimnisprinzip erlaubt es, Implementierungen auszutauschen, ohne die Verwendung funktional zu beeinflussen (aber Leistungsänderung möglich)

6 Abstraktion und Geheimnisprinzip

- Funktionsabstraktion: eine einzelne Funktion ist ein Modul, wenn ihr korrektes (zusicherungsgemäßes) Funktionieren nicht durch die Verwendung beeinflußt werden kann
  - in FP immer erfüllt, im imperativen Programmieren wegen potentieller Nebenwirkungen der Anwendung häufig nicht
- Datenabstraktion: ein ADT ist immer ein Modul, die Schnittstelle besteht aus Signaturen und Gesetzen
  - In imperativen Sprachen (C++, Java, C#, ...) können Objektattribute zur Schnittstelle gehören. Sie werden als Funktionspaar (lies_Wert, ändere_Wert) behandelt
- Umgekehrt sind im sequentiellen Programmieren nahezu alle Module in Wahrheit ADTs oder Funktionsabstraktionen; allerdings ist die Angabe der Gesetze oft sehr schwierig und aufwendig
- Viele Organisationsmodelle in der realen Welt sind ADTs!

6 Behälterdatentypen

- Die am häufigsten vorkommenden nicht-elementaren ADTs definieren Datenstrukturen wie Listen, Keller, Schlangen, Sequenzen, Dateien, Bäume, Graphen, Reihungen, Mengen, ..., die Elemente unterschiedlichen Typs α enthalten können
  - data Typ α = ... Konstruktor | Konstr... oder
  - type Typ α = ... Implementierung
- Solche ADTs heißen **Behälterdatentypen** (Behälter für Objekte des Typs α)
  - Beispiel in C++: die STL (*standard template library*) besteht ausschließlich aus Behältertypen

- Hauptziel des Kapitels: Einführung von Behältertypen (Auswahl) mit jeweils unterschiedlichen Implementierungen

6 Die häufigsten Fehler in der Softwarekonstruktion

- Fehler in der Anforderungsanalyse (**> 50%** !):
  - unzureichende oder fehlerhafte Erfassung und Analyse der Vorgaben, der verlangten Ergebnisse und deren Zusammenhänge
- Verstöße gegen das Geheimnisprinzip:
  - Schnittstellen passen nicht zusammen, z.B. weil unzureichend dokumentiert,
  - Anwendung nutzt Eigenschaften eines Moduls, die nicht zur Schnittstelle gehören; Modulimplementierung daher nicht mehr unabhängig änderbar,
    - schwarze Schnittstellen (nicht öffentlich bekannte Schnittstellen)
  - ...
- Programmierfehler
  - Fehler wegen unzureichender Aufgabenanalyse, mangelhafter Kenntnis der Schnittstellen von (Hilfs-)Moduln
  - Flüchtigkeitsfehler,
  - mangelhafte Beherrschung der Programmiersprache (nicht bei Profis)

Definition abstrakter Datentypen:

```
data Keller t = CreateStack | Push (Keller t) t
```

- definiert Konstruktoren CreateStack und Push
- Konstruktoren haben keine weitere Bedeutung„ d.h. sie spannen eine freie Termalgebra auf
- insbesondere gibt es keine Gesetze

☛ Konstruktoren definieren eine initiale Grundtermalgebra ohne Gesetze

- Werte sind also z.B. :

```
Main> Push CreateStack 1
Push CreateStack 1
Main> Push (Push CreateStack 1) 2
Push (Push CreateStack 1) 2
```

6 Wiederholung: Abstrakte Datentypen

- Werte abstrakter Datentypen sind Terme aus Konstruktoren
  - Konvention in FP: Bezeichner für Datentypen und Konstruktoren beginnen mit Großbuchstaben, alles andere klein geschrieben
- Die Signatur der Konstruktoren ist implizit definiert:
  - Ergebnistyp ist der definierte Datentyp
  - Argumenttypen sind explizit erwähnt

  Beispiele:

    - `CreateStack :: → Keller t`
    - `Push :: (Keller t) → t → Keller t`
- Behälterdatentypen haben einen Parameter, den Typ der Elemente im Behälter
  - alle Elemente haben gleichen Typ
  - solche Typparameter heißen generische (Typ-)Parameter

6.1 Natürliche Zahlen als ADT

Definition natürlicher Zahlen als Grundtermalgebra:

```
data Nat = Null | Succ Nat
```

Null, Succ Null, Succ Succ Null, . . .

- keine Gesetze! (freie Grundtermalgebra)
- weitere Funktionen z.B.

```
pre:: Nat → Nat
pre (Succ n) = n -- Vorgänger von Null nicht definiert
pre Null = ⊥
```

Alternative: Rückführung auf bekannte Typen (Angabe einer konkreten Implementierung)

- ```
  type Nat = Integer; null‘ :: Nat; succ‘ :: Nat → Nat
  null‘ = 0
  succ‘ n = n+1
  ```

[ ] 0
[T] 1
[F,T] 2
[F, T, T]

oder

- ```
  type Nat = [Bool]; null‘ :: Nat; succ‘ :: Nat → Nat
  null‘       = [ ]
  succ‘ [ ]   = [ True ]
  succ‘ (x:xs)      | x == True   = False : (succ‘ xs)
                    | otherwise   = True : xs
  ```

Lineare Datenstruktur:

Ein abstrakter Datentyp $\mathcal{A}$ heißt linear, falls er durch lineare Listen, d.h. Listen ohne Unterlisten, realisiert werden kann.

Eigenschaften:

- generischer Typparameter α für die Elemente
- heterogen, die lineare Datenstruktur benutzt außer $\mathcal{A}$ und α noch weitere Sorten, z.B.:
  - den Typ `Bool`, zur Prüfung, ob die Datenstruktur leer ist
  - `Int`, zum Abfragen des Umfanges der Datenstruktur

Beispiele:

Listen, Reihungen, Keller, Schlangen, Sequenzen (Dateien)

6.2 Listen

- In funktionalen Sprachen fest vorgegebener Datentyp
  `data Liste α = leere_Liste | (:) α Liste α`   [ ]
  statt `Liste α` schreibt man `[α]`
- Konstruktoren:
  - Leere Liste: `[]`
  - `(:)` α Liste α, d.h. Element vorne an Liste anfügen: `x:xs`
  - einelementige Listen: `x:[] == [x]`
- keine Hilfskonstruktoren
- zahlreiche weitere Operationen, wie bekannt
- in FP keine alternative Implementierung, grundlegender Datentyp
  - in imperativen Sprachen zahlreiche verschiedene Implementierungen, Hauptunterscheidungsmerkmale: wie schnell kommt man von der Position eines Elements zur Position eines anderen (Vorgänger, Nachfolger)? Wie effizient ist Einfügen und Streichen von Elementen sowie Anhängen von Listen? Sind zirkuläre Listen möglich?
- keine weiteren Neuigkeiten

Keller:

Der Keller arbeitet nach der letzter-zuerst-Strategie (engl. *last-in-first-out*, LIFO):

- Auslesen liefert das zuletzt eingefügte Element
- Beim Entfernen wird das zuletzt eingefügte Element entfernt.

LIFO-Strategie:

- Eltern, die ein Kind, das ein Fach, das Informatik heißt, studiert, haben, freuen sich
  - kein Wunder, daß das Kellerprinzip in Deutschland erfunden wurde (F.L. Bauer, K. Samelson, 1956/57)

6.2 Keller als ADT

- Datentyp mit Konstruktoren:

```
data Keller α = CreateStack | Push (Keller α) α
```

- Signatur der anderen Operationen:

```
top :: (Keller α) → α
pop :: (Keller α) → (Keller α)
```

- Axiome für LIFO-Verhalten:

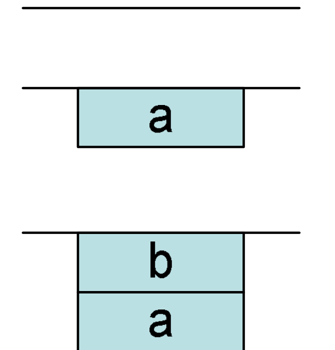
```
top (Push s x) = x
pop (Push s x) = s
```

- Abfrage, ob Keller leer:

```
isEmpty :: Keller α → Bool
isEmpty ( CreateStack )      = True
isEmpty ( Push k x )         = False
```

Beispiele:

- Aufbau eines Kellers:
  - `CreateStack`
  - `Push( CreateStack, 'a')`
  - `Push( Push( CreateStack, 'a'), 'b')`



- merke:
  - CreateStack usw. sind Kellerwerte
  - Push und pop liefern neue Kellerwerte
  - Einen Keller, der seinen Zustand ändert, gibt es in FP nicht

- Arbeiten auf dem Keller:
  - `top Push( Push( CreateStack, 'b'), 'a') == 'a'`
  - `pop (pop Push( Push( CreateStack, 'b'), 'a')) == CreateStack`

6.2 Kellerimplementierung mit Listen


```
type Keller α   = [α]

createStack     = []
push k x        = x:k
top (x:xs)      = x
pop (x:xs)      = xs
isEmpty []      = True
isEmpty _       = False
```

top (push k x) ==
top (x:k) == x
pop (push k x) ==
pop (x:k) == k

- Beweis der ADT-Eigenschaften:
  - Nachweis der Gesetze: elementar

6.2 Kellermaschine

grundlegende Anwendung von Kellern:

- Berechnung arithmetischer Ausdrücke in Postfixform (Kellermaschine)

```
berechne :: String → Int
berechne as = top (fst (eval CreateStack as))
eval :: (Keller Int) → String → (Keller Int, String)
eval k [] = (k, [])
eval k (a:as)
      |zahl a = eval (Push k (wandle a)) as
      |a=="+" = eval (bin (+) k) as
      |a=="-" = eval (bin (-) k) as
      |a=="*" = eval (bin (*) k) as
      |a=="/" = eval (bin (/) k) as
bin :: (Int → Int → Int) → (Keller Int) → (Keller Int)
bin f k = Push k2 (f (top k1) (top k)
            where {k1 = pop k; k2 = pop k1}
```

Ausdruck: 5 * (7 – 3)
Postfix: 5 7 3 - *

berechne "5 7 3 - *"

5
7
3

hier nicht angegeben:

`zahl :: Char → Bool` – teste, ob Ziffer vorliegt

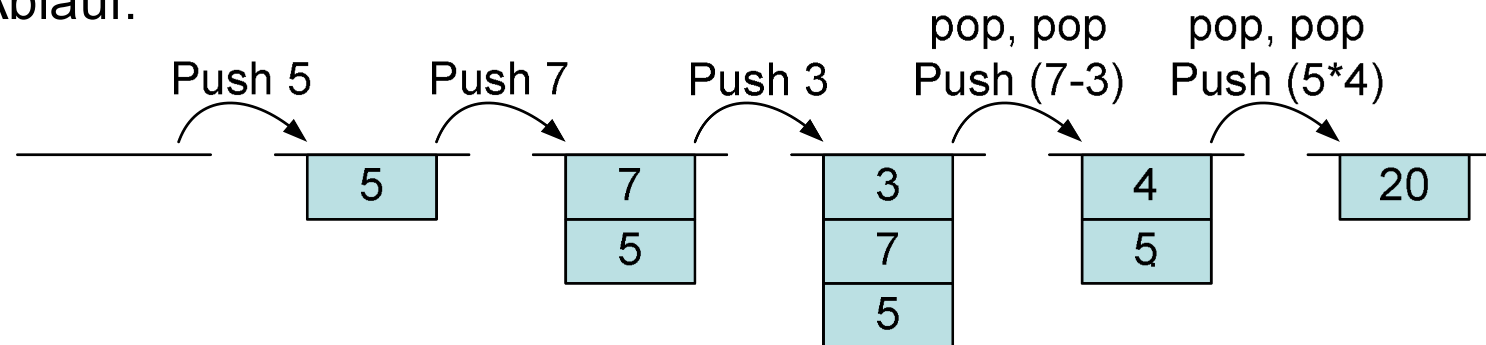
`wandle :: Char → Int` – wandle Ziffer in Zahl

6.2 Kellermaschine II

- Der Ausdruck muß in Postfixform vorliegen
- Ablauf:
  - Operanden auf Keller legen, bis Operator erreicht
  - Dann, zwei Operanden vom Keller nehmen, Ergebnis berechnen
  - Ergebnis auf Keller legen
  - usw.

Beispiel:

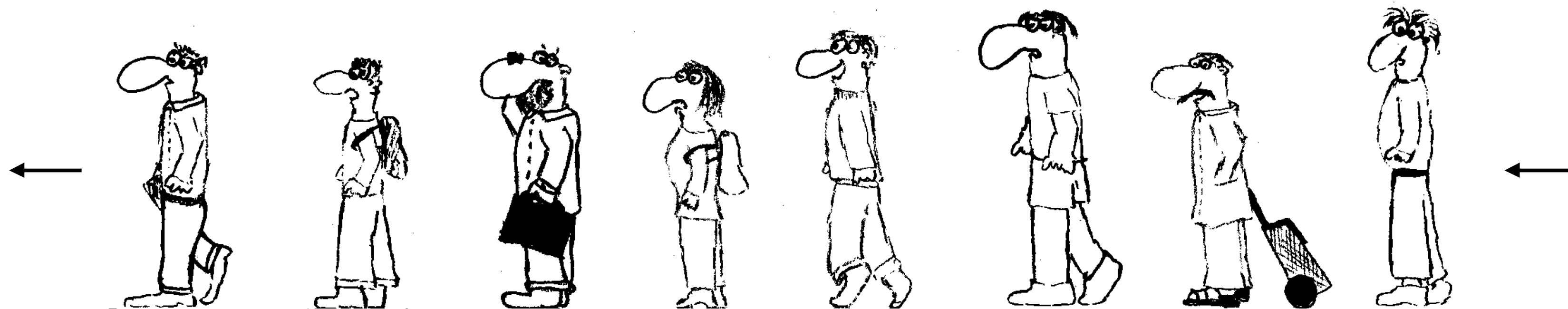
- Ausdruck 5 * (7 - 3) in Postfixform: 5 7 3 - *
- Ablauf:



Schlange (engl. *queue*):

Die Schlange arbeitet nach der **erster-zuerst-Strategie** (engl. ***first-in-first-out***, FIFO):

- Auslesen liefert das Element, das am längsten in der Schlange ist.
- Entfernen beseitigt dieses Element.



6.2 Schlangen als ADT

- **Datentyp mit Konstruktoren:**

```
data Schlange α = CreateQueue | Enqueue (Schlange α) α
```

≙ Push

- Signatur der anderen Operationen:

```
front   :: (Schlange α) → α
dequeue :: (Schlange α) → (Schlange α)
```

top ≙ front, pop ≙ dequeue

- Axiome für FIFO-Verhalten:

```
front   (Enqueue CreateQueue x) = x
front   (Enqueue s x)           = front s  -- s ≠ CreateQueue
dequeue (Enqueue CreateQueue x) = CreateQueue
dequeue (Enqueue s x)           = Enqueue (dequeue s) x
```

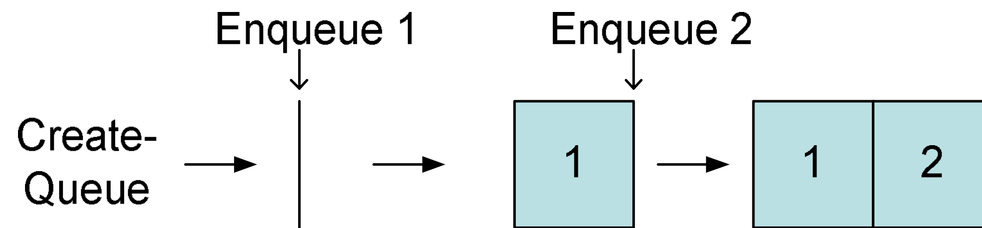
Reihenfolge!

- Abfrage, ob Schlange leer:

```
isEmpty:: Schlange α       → Bool
isEmpty ( CreateQueue )    = True
isEmpty ( Enqueue s x )    = False
```

Beispiel:

- Aufbau einer Schlange:
  - `Enqueue (Enqueue CreateQueue 1) 2`



- Arbeiten auf der Schlange:
  - `dequeue (Enqueue (Enqueue CreateQueue 1) 2)`

    `→ Enqueue CreateQueue 2`

    `front (Enqueue (Enqueue CreateQueue 1) 2)`

    `→ 1`

6.2 Schlangenimplementierung durch Listen

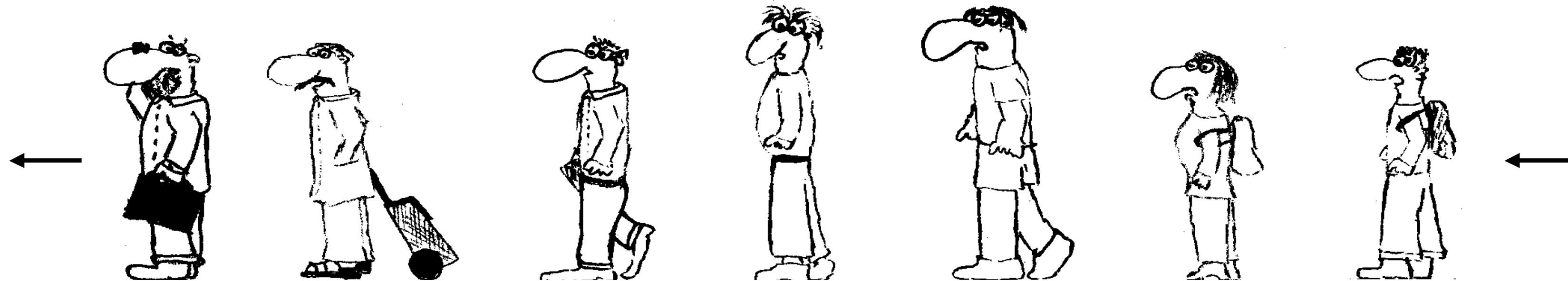
- Genauso wie einen `Keller` kann man auch den Datentyp `Schlange` mit Hilfe von Listen implementieren:

```
type Schlange t          = [t]

createQueue              = []
dequeue  (a:as)          = as
enqueue  []  x           = [x]
enqueue  (a:as)  x       = a:(enqueue as x)
front    (a:as)          = a
empty    []              = True
empty    (a:_)           = False
```

- „Erster-zuerst" Eigenschaft realisiert durch Anhängen neuer Elemente am Listenende, vorderstes Element damit stets das zuerst eingefügte.

Beispiele von Prioritäten:

- Anordnung von Leuten in einer Schlange nach Alter
- Anordnung von Kreditwünschen nach Höhe, usw.



Prioritätswarteschlangen:

Behälter mit folgender Strategie:

- Auslesen liefert das Element höchster Priorität, das am längsten in der Schlange war.
- Beim Entfernen wird dieses Element entfernt.

Voraussetzung:

- Der Datentyp der Prioritäten muß mit < total geordnet sein
- Eingetragen werden Paare (element, prio)

6.2 Prioritätsschlangen als ADT

Problem: Finde Element maximaler Priorität, das am längsten in der Schlange ist

☛ Hilfsfunktion zum Berechnen der maximalen Priorität einer Schlange

- Datentyp:

```
data Schlange α = CreateQueue | Enqueue (Schlange α ) (α, prio)
```

- Signatur:

```
front    :: (Schlange α)     → α
dequeue  :: (Schlange α)     → (Schlange α)
priority :: (Schlange α)     → prio
```

liefert höchste Prio
priority sollte privat sein

- Axiome:

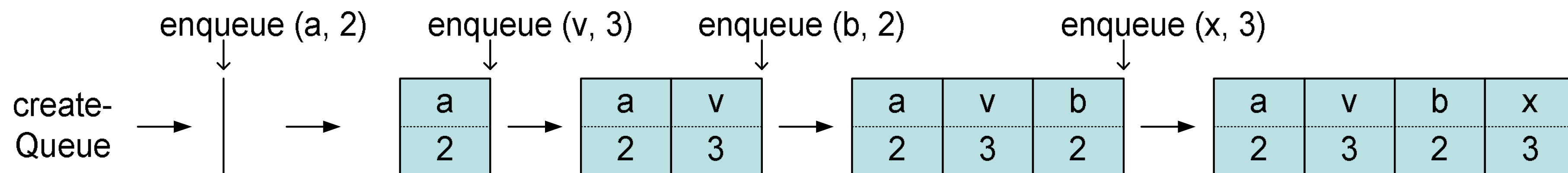
```
front (Enqueue CreateQueue (x, p)) = x
front (Enqueue s (x, p))     | priority s < p = x
                             | otherwise      = front s
dequeue (Enqueue CreateQueue (x, p))          = CreateQueue
dequeue (Enqueue s (x, p))   | priority s < p = s
                             | otherwise
                                 = Enqueue (dequeue s) (x, p)
priority (Enqueue CreateQueue (x, p)) = p
priority (Enqueue s (x, p))           = max p (priority s)
```

6.2 Öffentliche und private Funktionen

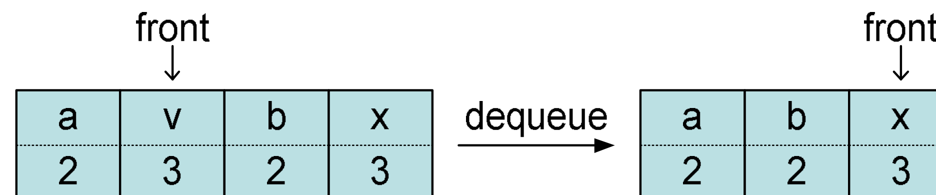
- Soll die Hilfsfunktion `priority` auch den Anwendern des ADT zugänglich sein? eigentlich nein
- Methodik allgemein:
  - Sichtbarkeit: Unterscheide öffentliche und private Funktionen eines ADTs (oder Moduls)
    - Dazwischen gibt es in manchen Sprachen noch limitierte Sichtbarkeit (Java: protected)
  - Verschiedene Sprachen haben unterschiedliche Grundannahmen:
    - Grundannahme öffentlich: alle Funktionen sind öffentlich, es sei denn sie sind explizit als privat gekennzeichnet
    - Grundannahme privat: alle Funktionen sind privat, es sei denn sie sind explizit als öffentlich gekennzeichnet
- Lösung in Haskell:
  - Unterscheidung auf Modulebene:
    - **`module`** `Modulname` **`where`** `...` : alle Fkt. des Moduls öffentlich
    - **`module`** `Modulname` $(f_1, f_2, ...)$ **`where`** `...` : nur $f_1, f_2, ...$ öffentlich

Beispiel:

- Aufbau einer Prioritätsschlange:
  - `Enqueue (Enqueue (Enqueue (Enqueue CreateQueue ('a', 2)) ('v', 3)) ('b',2)) ('x',3)`



- Operationen auf der Schlange:
  - `front (dequeue (Enqueue (Enqueue (Enqueue (Enqueue CreateQueue ('a', 2)) ('v', 3)) ('b',2)) ('x',3))) → 'x'`



6.2 Prioritätssschlangen implementiert mit Listen

Idee: sortiere die Schlange nach Prioritäten, benutze Sortieren durch Einfügen (die ältesten Leute kommen zuerst in der Schlange)

Implementierung durch Listen: (keine Hilfsfunktion `priority`!)

```
type Pr = Int
type Schlange α = [(α, Pr)]

createQueue:: Schlange α
enqueue    :: Schlange α → (α , Pr) → Schlange α
front      :: Schlange α → α
dequeue    :: Schlange α → Schlange α

createQueue              = []
dequeue (a:as)           = as
front (a:as)             = fst a
enqueue [] (x, p)        = [(x, p)]
enqueue (a:as)(x, p) | snd a < p = (x, p):a:as
                     | otherwise = a:(enqueue as (x, p))
```

Satz: Die Implementierung ist korrekt

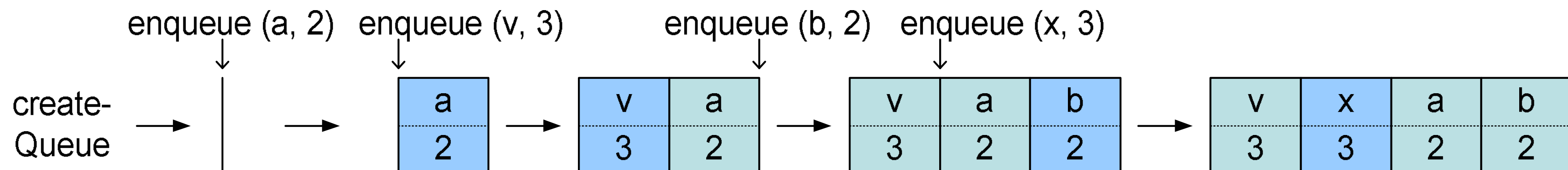
Beweis:

- durch Nachweis der Gleichungen oder
- durch Angabe eines Algebra-Homomorphismus

Beispiel:

- `enqueue (enqueue (enqueue (enqueue createQueue ('a', 2)) ('v', 3)) ('b', 2)) ('x', 3)`

  → `[('v', 3), ('x', 3), ('a', 2), ('b', 2)]`



Reihung: (engl. *array*) Liste fester Länge n mit folgenden Operationen:

- auslesen des i-ten Elements
- ändern des i-ten Elements ≙ liefere neue Reihung mit abgeänderten i-ten Element

Beispiel:

- auslesen des Elements mit Index 3 ergibt 'd'

0	1	2	3	4	5	6	7
'a'	'b'	'c'	'd'	'e'	'f'	'g'	'h'

- ändern des Elements mit Index 5 zu 'x' ergibt

0	1	2	3	4	5	6	7
'a'	'b'	'c'	'd'	'e'	'x'	'g'	'h'

6.2 Der Datentyp Reihung

n: Länge der Reihung/ des Arrays

statische Reihung: **n** ist im Programm **fest.**

☛ in Sprachen Pascal, C, Fortran, Cobol, Java

dynamische Reihung: **n** ist **nach Erzeugen** der Reihung **fest.**

☛ z.B. in Sprachen Algol60, Fortran, Ada, Java

flexible Reihung: **n** kann auch **während der Existenz** der Reihung **verändert werden.**

☛ z.B. in Algol68, C (mit Zeigern)

Reihungen sind in funktionalen Sprachen i. a. nicht verfügbar (Ändern eines Werts ist kein Konzept in FP)

Die Funktionen `createArray`, `assign` und `getElem`:

```
type Array α = [α]
```

[[α]] 2-stufig, usw.

1) CreateArray:

```
createArray :: Int → α → (Array α)
createArray 0_ = []
createArray (n+1) def = def : (createArray n def)
```

Vorbesetzung

Beispiel:

```
Main> createArray 5 0
[0,0,0,0,0]
```

2) assign:

```
assign ::(Array α) → Int → α → (Array α)
assign a i x | i >= 0 && i <(length a) =
                         (take i a) ++ [x] ++(drop (i +1) a)
             | i < 0     = error "assign: negative index"
             | otherwise = error "assign: index too large"
```

Beispiele:

```
Main> assign (create array 5 0) 3 1
[0,0,0,1,0]
Main> assign (assign (create array 5 0) 3 3) 3 4
[0,0,0,4,0]
Main> assign (create array 5 0) (-1) 1
Program error: assign: negative index
Main> assign (create array 5 0) 7 1
Program error: assign: index too large
```

3) getElem:

```
getElem ::(Array α) → Int → α
getElem a i = a !! i
```

Beispiele:

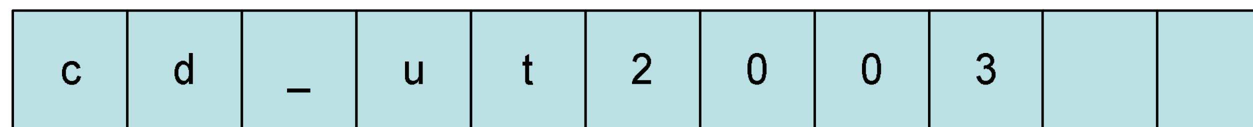
```
Main> getElem (assign (assign (createArray 5 0) 3 3) 2 2) 2
2
Main> getElem (create array 5 0) (-1)
Program error: Prelude.!!: negative index
```

Sequentielle Datei (engl. *File*): Datei ⊇ Files

Behälter mit Operationen zum

- Erzeugen von Dateien
- Schreiben von Dateien (`write`)
- Rückspulen der Datei (`reset`)
- Fortschalten der aktuellen Position (`skip`)
- Lesen von Daten an aktueller Position (`get`)
- Prüfung, ob aktuelle Position das Dateiende ist (`eof`)
- Beim Schreiben wird der nachfolgende Teil der Datei zerstört

Beispiel: Magnetband als Speichermedium mit Schreib-/Lesekopf zur Ein- und Ausgabe



Verbund (auch: Datensatz, engl. *record*): Elemente einer Datei

Hintergrundspeicher: Behälter für Dateien

Folgen von Datensätze, die nicht in den Arbeitsspeicher eines Rechners passen, werden als Dateien auf dem Hintergrundspeicher gespeichert.

Konventionen für die Ein-/Ausgabe von Programmen mit Textdateien: Benutzung von 3 Dateien (weitere Dateien möglich)

- Standardeingabe: Das Programm liest Eingaben aus dieser Textdatei
  - Die Datei kann nicht beschrieben werden (Eingabedatei)
  - Oft ist dies die Tastatur
- Standardausgabe: Das Programm schreibt Ausgaben in die Textdatei.
  - Die Datei kann nicht gelesen werden (Ausgabedatei)
  - Oft ist dies ein Drucker oder der Bildschirm.
- Standardfehlerausgabe: Das Programm schreibt Fehlermeldungen in diese Textdatei (Ausgabedatei).

Problem: Repräsentation des Schreib-/Lesekopfes

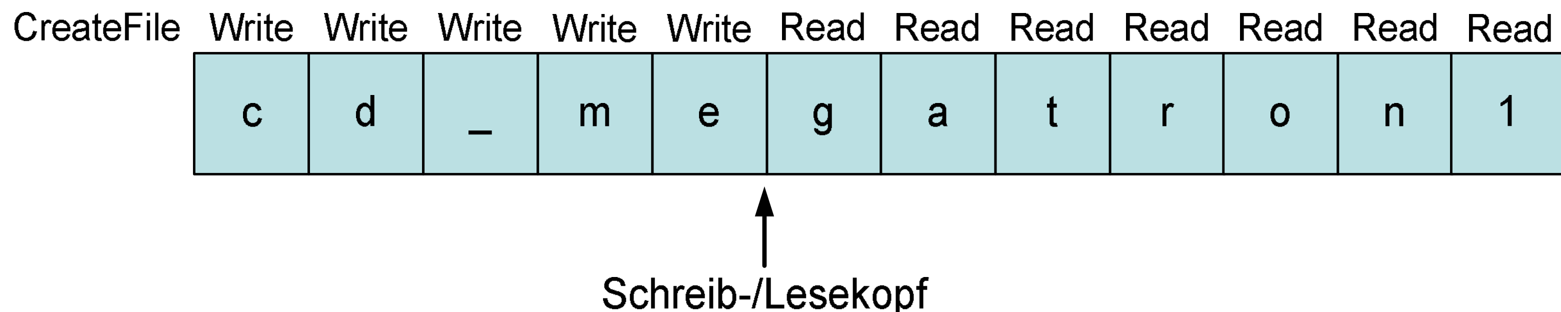
Idee:

- Elemente vor dem Schreib-/Lesekopf sind geschrieben, derzeit nicht lesbar: Konstruktor `Write`
- Elemente danach lesbar (mit Fortschalten des Schreib-/Lesekopfes): Konstruktor `Read`

⇒ Schreib-/Lesekopf am Übergang von `Write` zu `Read`

- sonst wird `Read` nicht benötigt (privater Konstruktor)

⇒ `Read` soll außerhalb des Datentyps nicht benutzt werden.



6.2 sequentielle Dateien als ADT II

- Datentyp:

```
data Datei α = CreateFile | Write (Datei α) α | Read (Datei α) α
```

- Signatur:

```
reset :: Datei α → Datei α
skip  :: Datei α → Datei α
get   :: Datei α → α
eof   :: Datei α → Bool
```

- Axiome:

```
skip  CreateFile               = error "empty file"
skip  (Write f a)              = error "skip at eof"
skip  (Read CreateFile a)      = Write CreateFile a
skip  (Read (Write f a) b      = Write (Write f a) b
skip  (Read (Read f a) b)      = Read (skip (Read f a)) b
reset CreateFile               = CreateFile
reset (Write f a)              = Read (reset f) a
reset (Read f a)               = Read (reset f) a
get   CreateFile               = error "empty file"
get   (Write f a)              = error "get at eof"
get   (Read CreateFile a)      = a
get   (Read (Write f a) b)     = b
get   (Read (Read f a) b)      = get (Read f a)
eof   CreateFile               = True
eof   (Write f a)              = True
eof   (Read f a)               = False
Write (Read f a) b             = Write f b
```

Fkt.def. für Konstruktor ???

6.2 Probleme

Das Gesetz

```
Write (Read f a) b = Write f b
```

repräsentiert die Regel
„Schreiben zerstört alle nachfolgenden Datensätze"

- korrekt in der abstrakten Algebra
- aber unzulässig in FP
  (keine Funktionsdefinition für Konstruktoren erlaubt)

Abhilfe: neue Funktion `put`:

```
put :: Datei α → α → Datei α
put CreateFile a     = Write CreateFile a
put (Write f a) b    = Write (Write f a) b
put (Read f a) b     = put f b
```

- `put` ist dem Benutzer zugänglich anstelle von `Write`, beide Konstruktoren `Write` und `Read` dürfen von außen nicht benutzt werden

Kapselung:

Operationen `Read` und `Write` sollten von außen nicht benutzt werden.

Abhilfe: Module

- fassen Typen und Funktionen zusammen
- erlauben es, nur relevante Typen und Daten zu exportieren
- nur diese sind von außen benutzbar

Schreibweise (am Beispiel Datentyp Datei):

```
module File (Datei, CreateFile, reset, skip,get, put, eof)
where
      data Datei = CreateFile | ...
```

Signaturen und Axiome wie auf den vorangehenden Folien

- Der Modul hat den Namen `File`.
- Die von außen sichtbaren Typen, Konstruktoren und Funktionen folgen in Klammern.

Benutzung:
Zur Verwendung der in einem Modul definierten Datentypen und Funktionen, müssen diese importiert werden.

Mehrere Varianten:

- **Komplettes Importieren:** `import File`
  importiert alle sichtbaren Typen, Konstruktoren und Funktionen
- **Partielles Importieren:** `import File (Datei, put, eof)`
  nur die in Klammern angebenen sichtbaren Typen, Konstruktoren und Funktionen werden importiert
- **Qualifiziertes Importieren:** `import qualified File`
  importiert alle sichtbaren Typen, Konstruktoren und Funktionen, erlaubt aber die Benutzung nur mit Qualifikation,
  Beispiel: `File.Datei, File.skip`
- Qualifiziertes und partielles Importieren können kombiniert werden.

Beispiele:

- Direktes Importieren:

```
import File
leer :: Datei Char
leer = CreateFile
```

Anwendung:
```
put leer 'a'
→ Write CreateFile 'a'
```

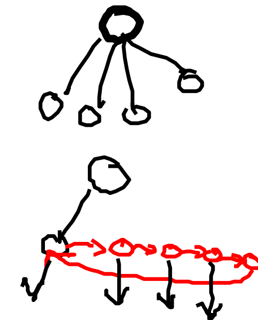
- Qualifiziertes Importieren:

```
import qualified File
leer :: File.Datei Char
leer = File.CreateFile
```

Anwendung:
```
File.eof File.put leer 'a'
→ True
```

Ziel:

Spezifikation eines Datentyps zur Verwaltung von Binärbäumen

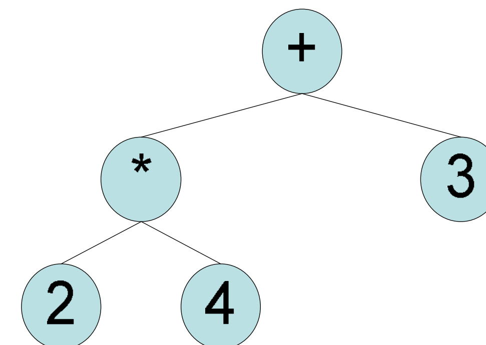


Binärbaum:

- Spezialfall eines Baumes mit Bedingung $|e^{\bullet}| \leq 2$, d.h. jede Ecke hat maximal zwei Nachfolger
- Jede Ecke erhält einen Wert (Alternative: **Blattbäume**, nur die Blätter haben einen Wert)
- Jede Ecke hat einen linken und rechten Nachfolger (Kind, Sohn)
  ☛ Binärbäume sind geordnet

Anwendungen:

- Kantorowitsch-Bäume
- Berechnungsformulare
- Suchbäume
- LISP-Listen (Blattbäume)



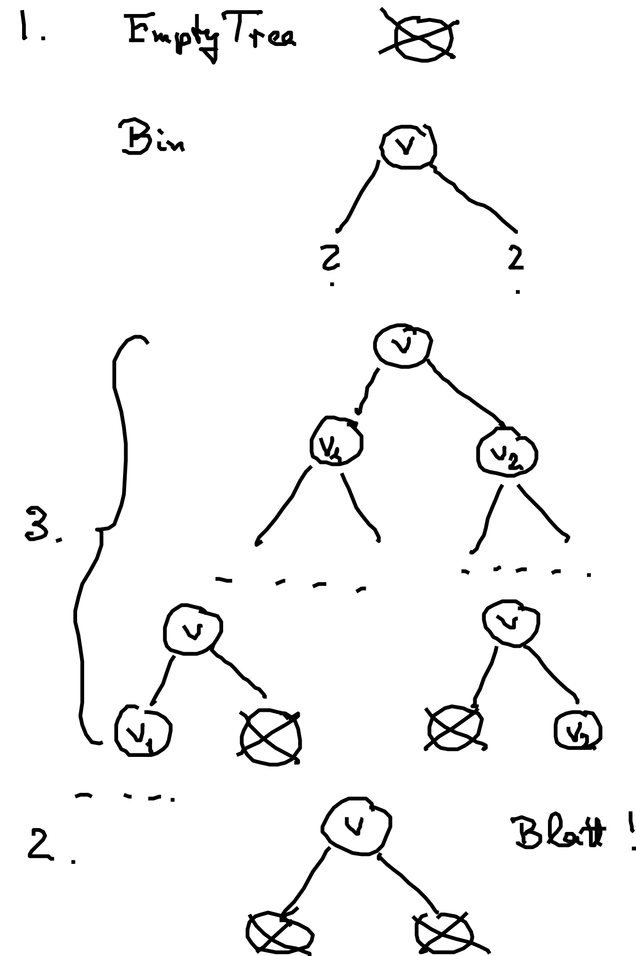
- **Datentyp:**

1. Kind   Wert   2. Kind

```
data BinTree α = EmptyTree | Bin (BinTree α) α (BinTree α)
```

- **Signatur:**

```
left   :: BinTree α → BinTree α
right  :: BinTree α → BinTree α
value  :: BinTree α → α
empty  :: BinTree α → Bool
```



- **Axiome:**

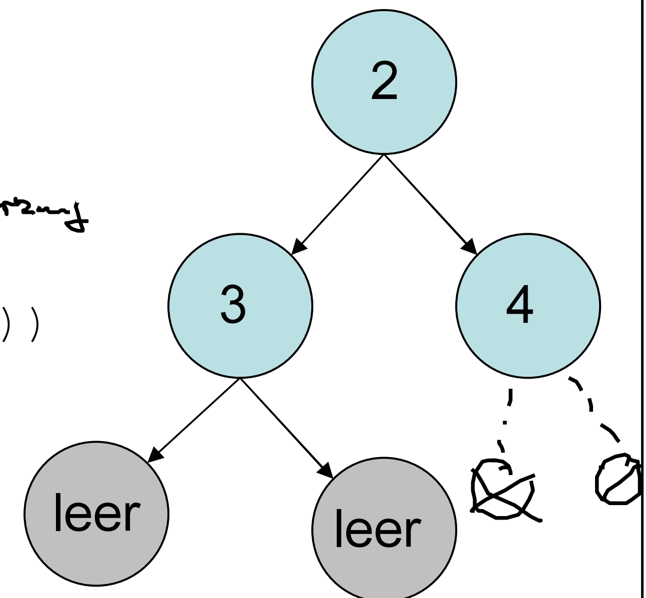
```
left   (Bin b1 _ _)   = b1
right  (Bin _ _ b2)   = b2
value  (Bin _ v _)    = v
empty  EmptyTree      = True
empty  (Bin _ _ _)    = False
```

6.3 Diskussion

- Es gibt zwei Arten von Bäumen:
  - leere Bäume (`EmptyTree`)
  - Ecken (`Bin`), bestehend aus einem Wert und zwei Bäumen
- Blätter repräsentiert durch (`Bin EmptyTree v EmptyTree`)

Beispiele:

- Definiere zur Vereinfachung die Funktion
  `leaf v = Bin EmptyTree v EmptyTree` Abkürzung
- `value( left( Bin( (leaf 3) 2 (leaf 4) )))`
  → `3`
- `left( Bin( (leaf 3) 2 (leaf 4) ))`
  → `Bin EmptyTree 3 EmptyTree`
  leaf 3
- `empty( left( left( Bin( (leaf 3) 2 (leaf 4) ))))`
  → `True`



6.3 Konstruktion von Kantorowitsch-Bäumen

Ziel:

Konstruktion eines Kantorowitsch-Baumes aus einem Ausdruck, gegeben in Infix-Form

$(a + b) * c$

gegeben:

- Datentyp Baum
- Ausdrucksgrammatik in EBNF

$A \rightarrow T \{+ T\}^*$

$T \rightarrow F \{* F\}^*$

$F \rightarrow bez \mid \text{'('}\ A\ \text{')'}$

Vereinfachungen:

- Beschränkung auf Operanden aus einzelnen Zeichen wie 'a', '1',…
- Verzicht auf unäre Operatoren + und –

Frage:

Wie lautet die Signatur der benötigten Funktionen ?

Grundstruktur von rekursivem Abstieg (recursive descent)

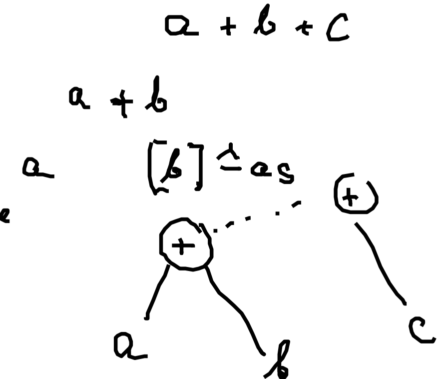
Funktionen: Text ≙ arithm. in Infix form

zur Erinnerung:
until p f x = if p x then x
else until p f (f x)

```
ausdruck xs = until p f (term xs) where
   p (b, [])        = True
   p (b, (a:_))     = ((a ≠ '+') && (a ≠ '-'))
   f (b, (a:as))    = (Bin b a (fst c), snd c)
                           where c = term as
```

ausdruck :: [Char] → (BinTree Char, [Char]) — Rest der Eingabe

```
term xs = until p f (faktor xs) where
   p (b, [])        = True
   p (b, (a:_))     = ((a ≠ '*') && (a ≠ '/'))
   f (b, (a:as))    = (Bin b a (fst c), snd c)
                           where c = faktor as

faktor (x:xs)
   | isAlphanum x = (leaf x, xs)
   | x == '('     = (formel, rest) where
                           formel         = fst(ausdruck xs)
                           ( ')':rest )  = snd(ausdruck xs)
```



Beispiel: Schrittweiser Aufbau des Baumes zu `x * (y * z)`

- `ausdruck (EmptyTree,"x*(y*z)")`
  ⇒ Kein '+' oder '-' am Anfang ⇒ Aufruf von `term`

- `term (EmptyTree,"x*(y*z)")`
  ⇒ Kein '*' oder '/' am Anfang ⇒ Aufruf von `faktor`

- `faktor (EmptyTree,"x*(y*z)")`
  ⇒ Kein '(' am Anfang ⇒ ersten Operanden abspalten,
  Rückgabe von `(leaf 'x', "*(y*z)"` an `term`
- `term (leaf 'x',"*(y*z)")`
  ⇒ `'*'` am Anfang ⇒ Einfügen einer übergeordneten Ecke,
  Rückgabe von `(Bin(leaf 'x') '*, fst(faktor("(y*z)")),`
  `snd(faktor("(y*z)")`

- usw.,
- Endergebnis:
  `(Bin (leaf 'x') '*' (Bin (leaf 'y') '*' (leaf 'z')), [])`
  wobei `leaf 'x'` usw. gemäß der Definition von `leaf` ausgeschrieben ist

6.3 Diskussion

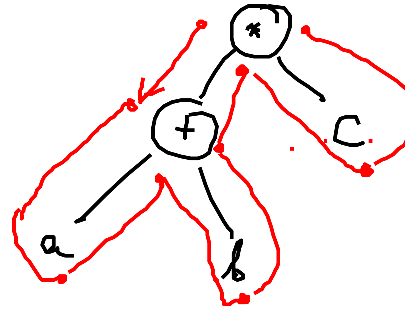
- Die Funktionen arbeiten nach dem Prinzip des **Durchreichens von Zwischenergebnissen:**
  - Funktionen erhalten Tupel bestehend aus bisher berechnetem Baum und noch zu verarbeitendem Rest der Eingabe.
  - Sie analysieren die Eingabe, erweitern den Baum, kürzen die Eingabe passend und geben das überarbeitete Tupel zurück.
- Die Funktionen entsprechen genau den EBNF-Produktionen:
  - `ausdruck` zerlegt Eingabe von links nach rechts mit Hilfe von `term` in Summanden und baut Baum auf (**Beispiel**: `Bin sum1 + sum2`)
  - `term` zerlegt Summanden unter Verwendung von `faktor` in Operanden und gibt Baumknoten zurück (**Beispiel**: `Bin fak1 * fak2`)
  - `faktor` liefert Operanden als Teilbäume zurück, entweder einzelne Zeichen (**Beispiel**: `Bin EmptyTree 'a' EmptyTree`) oder ganze Klammerausdrücke

Ziel:

Erzeugung einer linearen Liste aus einem Binärbaum.

Mehrere Möglichkeiten:

- Präfix-Form
- Postfix-Form
- Infix-Form



Funktion:

```
createList (Bin EmptyTree v EmptyTree)      = [v]
createList EmptyTree                        = []
createList (Bin b1 v b2)
              = tb1 ++ tb2 ++ [v]           -- Postfix
              = [v] ++ tb1 ++ tb2           -- Präfix
              = tb1 ++ [v] ++ tb2           -- Infix
                where  tb1 = createList b1
                       tb2 = createList b2
```

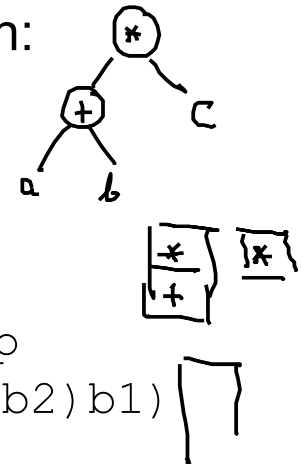
Tiefensuche:

Verfahren zum Durchlaufen aller Ecken eines Baumes. Man geht solange „links-abwärts" wie möglich und dann wieder zurück

Es gilt: Ein rechter Sohn wird erst untersucht, wenn der Unterbaum des linken Sohnes vollständig abgearbeitet wurde.

Dieses Verhalten kann durch einen Keller implementiert werden:

```
tiefensuche b =
 fst (until p f ([], Push CreateStack b)) where
   p(_, k)    = isEmpty k
   f(erg, k)  |(top k)== EmptyTree  =(erg, pop k)
              |otherwise =(erg ++ [v], Push(Push(pop
                                                  k)b2)b1)
              where (Bin b1 v b2)= top k
```



Hinweis: Die `isEmpty` Funktion gehört zum Datentyp `Keller`!

Breitensuche:

Hier werden die Ecken „zeilenweise von rechts nach links“ durchlaufen.

Es gilt: eine Ecke der Tiefe k+1 wird erst untersucht, wenn alle Ecken der Tiefe k abgearbeitet sind.

Dieses Verhalten ist durch eine FIFO-Schlange implementierbar:

```
breitensuche b =
 fst (until p f ([], Enqueue CreateQueue b)) where
     p(_, k) = emptyQueue k
     f(erg, k) | (front k) == EmptyTree =(erg, dequeue k)
               | otherwise                =
                (erg ++ [v],Enqueue(Enqueue(dequeue k)b2)b1)
        where (Bin b1 v b2)= front k
```

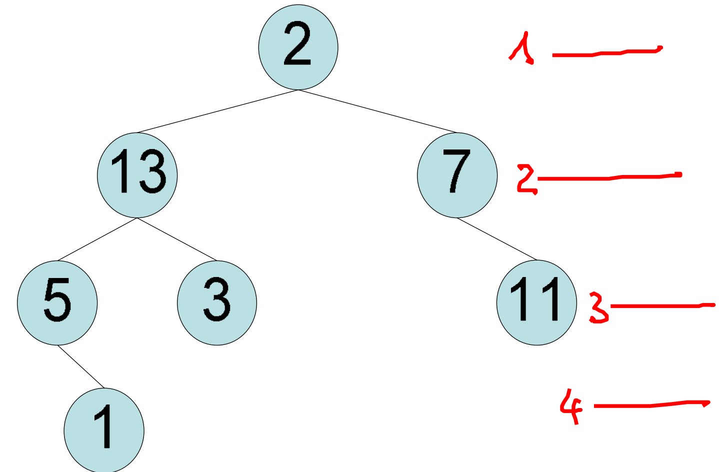
Übung: Für beide Suchen ist noch `==` auf `BinTree` zu implementieren.

6.3 Suchablauf

Beispiel:

- Tiefensuche auf dem rechts abgebildeten Baum liefert das Ergebnis:

  `[2, 13, 5, 1, 3, 7, 11]`



- Breitensuche liefert:

  `[2, 7, 13, 11, 3, 5 ,1]`

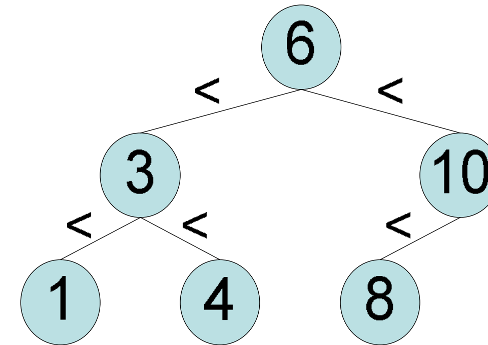
- Um im Binärbaum etwas zu finden, braucht man eine weitere Funktion `findIt` (hier für Tiefensuche):

```
                                              breitensuche
findIt a b             = findHelp a (tiefensuche b)
findHelp a (x:xs)      | a == x = True
                       | xs == []      = False
                       | otherwise     = findHelp a xs
```

<u>Binärer Suchbaum</u>:

Binärbaum mit Bedingung an jede Ecke k:

linker Nachfolger < k < rechter Nachfolger



(falls ein entsprechender Nachfolger existiert)

Voraussetzung:

Es muß eine Ordnungsrelation auf dem Elementtyp existieren.

Vorüberlegungen zur Implementierung:

- Erweitere Signatur des Datentyps Binärbaum um Funktionen zum Einfügen (`insert`) und Suchen (`find`) von Elementen.
- Zugriff auf Baum nur über diese Funktionen, zum Wahren der Suchbaum-Bedingung.

  ☛ Entfernen der anderen Funktionen oder Kapselung

6.3 Erweiterung der Implementierung

- Weitere Signaturen:

```
insert  :: (Eq α,Ord α) ⇒ BinTree α → α → BinTree α
find    :: (Eq α,Ord α) ⇒ BinTree α → α → Bool
```

- Weitere Axiome:

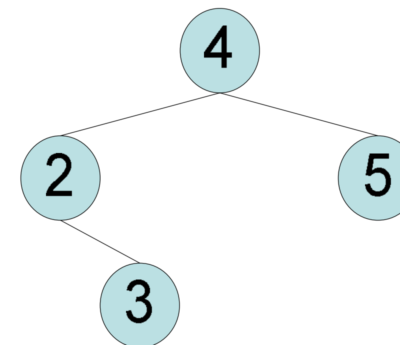
```
insert EmptyTree w    = Bin EmptyTree w EmptyTree
insert (Bin b1 v b2) w
    | w == v          = Bin b1 v b2
    | w < v           = Bin (insert b1 w) v b2
    | w > v           = Bin b1 v (insert b2 w)

find EmptyTree w      = False
find (Bin b1 v b2) w
    | w == v          = True
    | w < v           = find b1 w
    | w > v           = find b2 w
```

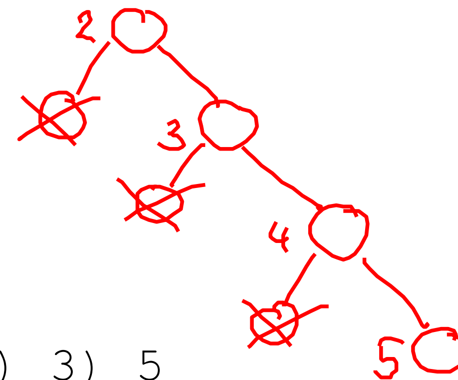
6.4 Diskussion

- Operationen `left`, `right` und `value` werden nicht mehr benötigt.
- Operation `delete` zum Löschen von Elementen als Übung.
- Einfügen und Suchen sind schnell (bestenfalls in O(log n)) möglich.
- Der Baum sieht je nach Reihenfolge des Einfügens anders aus

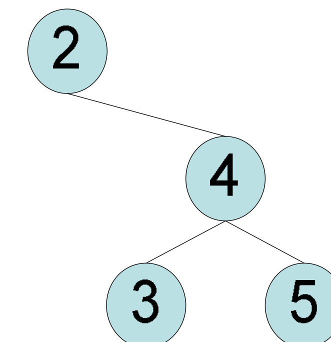
Beispiele:

- `insert (insert (insert (insert EmptyTree 4) 2) 5) 3`





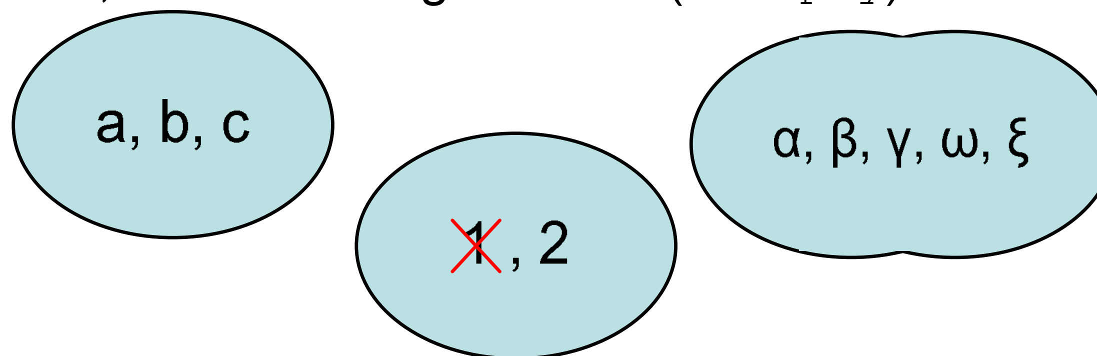
- `insert (insert (insert (insert EmptyTree 2) 4) 3) 5`



Ziel:

Mengenalgebra mit Operationen zum

- Erzeugen leerer Mengen
- ? Erzeugen einelementiger Mengen (`single`)
- Einfügen eines Elementes in eine Menge (`insert`)
- Prüfen, ob ein Element in einer Menge ist (`isElem`)
- Löschen eines Elementes aus einer Menge (`delete`)
- Vereinigen (`union`) und Schneiden (`intersect`) zweier Mengen
- Berechnen der Differenz zweier Mengen (`diff`)
- Nachprüfen, ob eine Menge leer ist (`isEmpty`)



6.4 Probleme

Ausgangspunkt: abstrakte Algebra der Mengen

Konstruktoren: leere Menge

```
data Set α = CreateSet | Insert (Set α) α
```

Problem 1: Nach abstrakter Algebra gilt z.B.

```
Insert (Insert s x) x = Insert s x
```
notwendiges Gesetz

☛ Funktionale Sprachen erlauben allerdings keine Gleichungen über Konstruktoren.

Abhilfe: Verberge Konstruktoren zum Aufbau einer Menge

☛ Definiere Datentyp als Modul.

Problem 2: Mengenoperationen stützen sich auf den Vergleich von Elementen

☛ Elementtyp α muß Instanz der Typklasse `Eq` sein.

- Kapselung:

```
module Set ( Set, CreateSet, single, insert, isElem, delete,
             union, intersect, diff, isEmpty ) where
```

- Datentyp:

```
data Set α =  CreateSet | Add (Set α) α
```

- Signatur:

```
single      :: α → Set α
insert      :: Eq α ⇒ Set α → α → Set α
delete      :: Eq α ⇒ Set α → α → Set α
union       :: Eq α ⇒ Set α → Set α → Set α
intersect   :: Eq α ⇒ Set α → Set α → Set α
diff        :: Eq α ⇒ Set α → Set α → Set α
isElem      :: Eq α ⇒ α → Set α → Bool
isEmpty     :: Set α → Bool
```

- `Eq α ⇒` heißt auch Zwangsbedingung (engl. *constraint*)

6.4 Mengen als ADT II

- Axiome:

```
single x                        = Add CreateSet x
insert s x                      = union s (single x)
delete s x                      = diff s (single x)


union CreateSet s               = s
union (Add s x) z
             | isElem x z       = union s z
             | otherwise        = Add (union s z) x


intersect CreateSet s           = CreateSet
intersect (Add s x) z
             | isElem x z       = Add (intersect s z) x
             | otherwise        = intersect s z
```

6.4 Mengen als ADT III

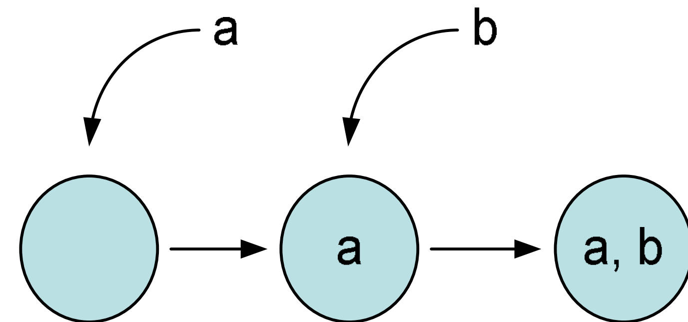
- Axiome (fortgesetzt):

```
diff CreateSet s                          = CreateSet
diff (Add s x) z
             | not (isElem x z)           = Add (diff s z) x
             | otherwise                  = diff s z


isElem x CreateSet                        = False
isElem x (Add s y)
             | x == y                     = True
             | otherwise                  = isElem x s


isEmpty CreateSet                         = True
isEmpty (Add s x)                         = False
```

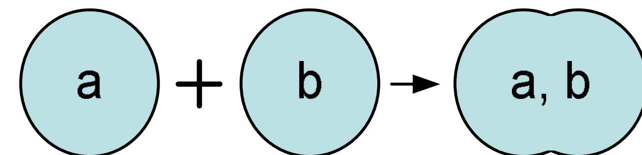
6.4 Verwendung des Datentyps Menge

Beispiele:

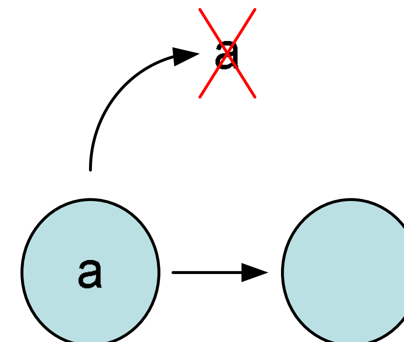
- `insert (insert CreateSet 'a') 'b'`
  → `Add (Add CreateSet 'a') 'b'`



- `union (insert CreateSet 'a') (insert CreateSet 'b')`
  → `Add (Add CreateSet 'b') 'a'`



- `delete (insert CreateSet 'a')`
  → `CreateSet`



Beispiel:

Ziel: {{a}}

- `insert CreateSet (insert CreateSet 'a')`

  → „Error: Illegal Haskell 98 class constraint in inferred type"

für Typ „Set Char" ist keine Gleichheit definiert!

- Grund:
  - Der Term hat Typ `Set (Set Char)`
  - Damit hat `insert` den Typ

```
Eq (Set Char) ⇒ Set (Set Char) → Set Char → Set (Set Char)
```

  - Also müßte wegen des Constraints `Eq(Set Char)` die Operation

```
(==) :: Set Char → Set Char → Bool
```

auf `Set Char` definiert sein, damit obiger Ausdruck funktioniert.

6.4 Datentyp Vielfachmenge

multiset bag

Intuitiv: Mengen, in denen Elemente mehrfach vorkommen können.

Vielfachmenge über Universum U (engl. *bag*): Abbildung $Q: U \to \mathbb{N}$

- $Q(x)$ gibt an, wie oft x vorkommt (**Vielfachheit von x**)
- $X \in Q$ gdw. $Q(x) > 0$ (**Elementbeziehung**)
- $Q \subseteq P$ gdw. $\forall x \in U: Q(x) \leq P(x)$ (**Teilmengenbeziehung**)
- $(Q \cup P)(x) \triangleq \max(Q(x), P(x))$ (**Vereinigung**)
- $(Q \cap P)(x) \triangleq \min(Q(x), P(x))$ (**Durchschnitt**)
- $\uplus$ $(Q \uplus P)(x) \triangleq Q(x) + P(x)$ disjoint union (**disjunkte Vereinigung**)
- $(Q \setminus P)(x) \triangleq \max(Q(x) - P(x), 0)$ (**Differenz**)

1. Implem.: Funktion (endliches U!)
2. [. . . $(x, Q(x))$, . . .]

Leere Vielfachmenge: $\forall x \in U: Q(x) = 0$

Endliche Vielfachmenge: $Q(x) > 0$ nur für endlich viele $x \in U$

Beispiel: Menge der Primfaktoren von 2250: [2, 3, 3, 5, 5, 5]

6.4 Implementierung

Datentyp Vielfachmenge:

Vielfachmengenalgebra mit Operationen zum

- Erzeugen leerer Vielfachmengen
- Erzeugen einelementiger Vielfachmengen (`single`)
- Einfügen eines Elementes in eine Vielfachmenge (`insert`)
- Prüfen, ob ein Element in einer Vielfachmenge ist (`isElem`)
- Löschen eines Elementes aus einer Vielfachmenge (`delete`)
- Vereinigen (`union`), disjunkten Vereinigen (`plus`) und Schneiden (`intersect`) zweier Vielfachmengen
- Berechnen der Differenz zweier Vielfachmengen (`diff`)
- Nachprüfen, ob eine Vielfachmenge leer ist (`isEmpty`)

Ausgangspunkt: abstrakte Algebra
☛ Übung

Implementierung in Haskell: Übung