Überblick

1. Typen im Überblick
2. Deklarationen von Typabkürzungen in SML: type-Deklarationen
3. Definition von Typen: datatype-Deklarationen
4. Definition von rekursiven Typen
5. Beweisprinzip der strukturellen Induktion
6. Beispiele: Implementierungen des Konzepts der „Menge“
7. Beispiele: Grundlegende Funktionen für binäre (Such-) Bäume

Was ist eine Menge? (Wiederholung)

- Die „*Menge*“ ist ein grundlegender Begriff der Mathematik zur Zusammensetzung von Objekten.
- Die Zusammensetzung von Objekten als Menge besitzt weder Reihenfolge noch irgendeine sonstige Strukturierung der Objekte.
- Die von einer Menge zusammengefassten Objekte werden „Elemente“ dieser Menge genannt.

Referenzmenge (Wiederholung)

- Eine Menge wird immer bezüglich einer „Referenzmenge“ definiert, d.h. einer „Urmenge“, woraus die Elemente der zu definierenden Mengen stammen.
- Der Verzicht auf eine Referenzmenge würde Paradoxien ermöglichen, wie etwa das folgende Paradoxon:
  - *Sei* M *die (Pseudo-) Menge, die alle Mengen umfasst, die nicht Element von sich selbst sind: Gilt* $M \in M$ *?*
  - Falls ja, dann gilt nach Definition von M: $M \notin M$, ein Widerspruch.
  - Falls nein, dann gilt nach Definition von M: $M \in M$, ein Widerspruch.
- Die Bedingung, dass keine Menge M definiert werden kann, ohne eine Referenzmenge festzulegen, schließt einen solchen Fall aus.

Extensional und intensional definierte Mengen (Wiederholung)

- Eine Menge wird „*extensional*“ (oder „*explizit*“) definiert, wenn ihre Definition aus einer Auflistung ihrer Elemente besteht.
- Beispiel:

  $\{1.0, 23.5, 12.45\}$ ist eine extensional definierte Menge, deren Referenzmenge **R** ist.
- Eine Menge wird „*intensional*“ (oder „*implizit*“) definiert, wenn ihre Elemente durch eine Eigenschaft charakterisiert werden.
- Beispiel:

  $\{x*x \mid x \in \mathbf{N}\}$ ist eine intensional definierte Menge, deren Referenzmenge **N** ist.

Funktionen als Gegenstück intensional definierter Mengen

- Funktionen sind in einer Programmiersprache das Gegenstück zu *intensional* definierten Mengen.
- Die Funktion

```
fun quadrat(x : int) = x * x
```

drückt in SML die Menge $\{x*x \mid x \in \mathbf{Z}\}$ als Menge möglicher Ergebniswerte aus.

- Die Datenstruktur „Menge“, die es zu implementieren gilt, kann sinnvollerweise also nur *extensional* definierte, zudem endliche Mengen wiedergeben.

Mengenoperationen (Wiederholung)

- Mit dem Begriff „Menge“ werden die folgenden grundlegenden Operationen definiert:

Elementbeziehung:	$\in$
Vereinigung:	$M1 \cup M2 = \{x \mid x \in M1 \vee x \in M2\}$
Durchschnitt:	$M1 \cap M2 = \{x \mid x \in M1 \wedge x \in M2\}$
Gleichheit:	Zwei Mengen sind gleich, wenn sie genau dieselben Elemente haben.
Teilmengenbeziehung:	$M1$ ist eine Teilmenge von $M2$ ($M1 \subseteq M2$), wenn jedes Element von $M1$ auch Element von $M2$ ist.

- Zudem ist die „leere Menge“ eine ausgezeichnete Menge, die keine Elemente hat.

Was ist eine „Datenstruktur“?

- Unter einer „*Datenstruktur*“ versteht man in der Informatik
  - eine Darstellung einer mathematischen Struktur (z.B. Mengen, Vektoren oder Listen) in einer Programmiersprache zusammen mit
  - der Implementierung der grundlegenden Operationen dieser Struktur in derselben Programmiersprache basierend auf dieser Darstellung.
- In einer typisierten Programmiersprache wie SML ist die Definition eines Typs ein gewöhnlicher Teil der Implementierung einer Datenstruktur.
- Die Definition eines Typs allein reicht in der Regel nicht aus, weil damit die grundlegenden Operationen der betrachteten Struktur noch lange nicht gegeben sind.

In der Praxis besteht die Implementierung einer Datenstruktur typischerweise aus einem Typ und aus Prozeduren (die nicht immer Funktionen sind), die sich auf diesen Typ beziehen.

Die Menge als charakteristische Funktion

```
- type 'a menge = 'a -> bool;
type 'a menge = 'a -> bool

- val ziffer_menge : int menge =
      fn 0 => true
       | 1 => true
       | 2 => true
       | 3 => true
       | 4 => true
       | 5 => true
       | 6 => true
       | 7 => true
       | 8 => true
       | 9 => true
       | _ => false;
val ziffer_menge = fn : int menge
```

So eine Funktion, wie diese aus Kapitel 8.2, nennt man charakteristische Funktion (genauer: charakteristisches Prädikat) der Menge (aller Ziffern).

vereinigung, durchschnitt

- Die vorhergehende Implementierung gibt die Elementbeziehung unmittelbar wieder.
- Die Vereinigung und der Durchschnitt lassen sich sehr einfach realisieren:

```
- fun vereinigung(m1:'a menge,m2:'a menge) =
                              fn x => m1(x) orelse m2(x);
val vereinigung = fn :'a menge *'a menge -> 'a -> bool

- fun durchschnitt(m1:'a menge,m2:'a menge) =
                              fn x => m1(x) andalso m2(x);
val durchschnitt = fn :'a menge *'a menge ->'a -> bool
```


Beispiel: string menge

```
- val M1 : string menge =
      fn "ab" => true
      |  "bc" => true
      |  "be" => true
      |  _ => false;
val M1 = fn : string menge
- val M2 : string menge =
      fn "de" => true
      |  "fg" => true
      |  "be" => true
      |  _ => false;
val M2 = fn : string menge
```

Anwendung

```
- vereinigung(M1, M2);
val it = fn : string -> bool

- vereinigung(M1, M2)("be");
val it = true : bool

- durchschnitt(M1, M2)("ab");
val it = false : bool
```


Gleichheit und Teilmengenbeziehung

- Diese Implementierung ist zur Implementierung der Gleichheit (von Mengen) und der Teilmengenbeziehung nicht geeignet.
- Geeigneter wäre eine Darstellung, die es ermöglicht, die Auflistung der Elemente beider Mengen, die auf Gleichheit oder Teilmengenbeziehung untersucht werden sollen, direkt zu vergleichen.
- Da die Auflistung im Programm selbst statt in einem Aufrufparameter vorkommt, ist ein solcher Vergleich in dieser Implementierung nicht einfach.

Die Menge als Liste

- Man kann eine extensional definierte, endliche Menge als Liste darstellen.
- Die Elementbeziehung wird durch das polymorphe Prädikat `member` implementiert:

```
- fun member(x, nil) = false
  | member(x, head::tail) = if x = head
                            then true
                            else member(x, tail);
val member = fn : ''a * ''a list -> bool

- member(3,[1,2,3,4]);
val it = true : bool
```


Schätzung des Zeitbedarfs

- Der Zeitbedarf einer Überprüfung einer Elementbeziehung kann so geschätzt werden:
  - Als Schätzung der Problemgröße bietet sich die Größe (Kardinalität) der Menge an.
  - Als Zeiteinheit bietet sich die Anzahl der rekursiven Aufrufe des Prädikats `member` an.
- Zur Überprüfung einer Elementbeziehung bezüglich einer Menge der Kardinalität n wird man `member` bestenfalls ein Mal, schlechtestenfalls $(n + 1)$ Mal aufrufen.
- Der Zeitbedarf einer Überprüfung einer Elementbeziehung ist also schlechtestenfalls $O(n)$.

Die Vereinigung

- Die Vereinigung wird durch die polymorphe Funktion `append` (in SML auch als vordefinierte Infixfunktion „@“ vorhanden) implementiert:

```
- fun append(nil, L) = L
    | append(h :: t, L) = h :: append(t, L);
val append = fn : 'a list * 'a list -> 'a list

- append([1,2,3], [4,5]);
val it = [1,2,3,4,5] : int list
```

- Diese Implementierung der Vereinigung kann für manche Anwendungen ungeeignet sein, weil sie die Wiederholung von Elementen nicht ausschließt.

Der Durchschnitt

- Der Durchschnitt kann so implementiert werden:

```
- fun durchschnitt(nil, _) = nil
   | durchschnitt(h :: t, L) =
                  if member(h, L)
                  then h :: durchschnitt(t, L)
                  else durchschnitt(t, L);
val durchschnitt = fn : ''a list * ''a list -> ''a list

- durchschnitt([1,2,3,4], [3,4,5,6]);
val it = [3,4] : int list
```

Die Teilmengenbeziehung

- Die Teilmengenbeziehung kann so implementiert werden:

```
- fun teilmenge(nil, _) = true
   | teilmenge(h :: t, L) =
                  if member(h, L)
                  then teilmenge(t, L)
                  else false;
val teilmenge = fn : ''a list * ''a list -> bool

- teilmenge([6,4,2],[1,2,8,4,9,6,73,5]);
val it = true : bool

- teilmenge([4,2,3,1], [3,6,5,4]);
val it = false : bool
```

Um die Menge zu verändern, stehen in SML der Konstruktor `cons` (`::`) und die vordefinierten Funktionen `hd` und `tl` zur Verfügung.

Die Menge als sortierte Liste (1)

- Die eben eingeführte Implementierung der Menge setzt NICHT voraus, dass die Listenelemente in auf- oder absteigender Reihenfolge sortiert sind.
- Für eine Sortierung der Elemente braucht man eine Ordnung (d.h. eine reflexive, transitive und antisymmetrische Relation) über der Referenzmenge.
- Diese Ordnung muss total sein, d.h., dass je zwei beliebige Elemente in der einen oder anderen Richtung zueinander in Relation stehen.
- Wenn das im Folgenden angenommen wird, bleibt zu untersuchen, ob eine Sortierung der Elemente (nach der Ordnung der Referenzmenge) von Vorteil ist.

Die Menge als sortierte Liste (2)

- Sind die Elemente nach aufsteigenden (bzw. absteigenden) Werten sortiert, so kann die sequenzielle Suche durch die Liste, die das Prädikat `member` durchführt, abgebrochen werden, sobald ein Listenelement gefunden wird, das größer (bzw. kleiner) als das gesuchte Element ist.

Verbesserte Suche in einer aufsteigend sortierten Liste

```
- fun member_sort(x, nil) = false
    | member_sort(x, h::t) =
                  if x < h
                  then false
                  else if x = h
                        then true
                        else member_sort(x, t);
val member_sort = fn : int * int list -> bool
```

- Schlechtestenfalls wird mit `member_sort` und einer sortierten Liste sowie `member` und einer beliebigen Liste die Liste ganz durchlaufen.
- Die Suche mit `member_sort` und einer sortierten Liste benötigt eine Zeit, die $O(n)$ ist (wenn n die Länge der Liste ist).

Ermöglicht eine Sortierung der Elemente eine effizientere Suche als die sequenzielle Suche mit `member_sort`?

- Beispiel Telefonbuch:

  Wenn man im Telefonbuch nach der Telefonnummer eines Herrn Zimmermann sucht, schlägt man es am Ende auf, für die Nummer einer Frau Ackermann am Anfang und für die Nummer eines Herrn Klein in der Mitte usw.
- Der Bereich der möglichen Namen ist bekannt:

  Alle Namen fangen mit einem Buchstaben zwischen A und Z an.
- Die Verteilung der Namen ist ebenfalls einigermaßen bekannt: Deutsche Familiennamen fangen häufiger mit K oder S an als mit I, N oder O.
- Man kann also ziemlich schnell in die Nähe des gesuchten Namen kommen.

Korrektheit der Überlegung

- Die Annahme, dass die Überprüfung der Elementbeziehung in einer sortierten Liste schneller erfolgen kann als in einer unsortierten Liste ist inkorrekt:

  Eine Liste (sortiert oder unsortiert) kann immer nur linear von ihrem ersten Element an durchlaufen werden.
- Die Darstellung der endlichen, extensional definierten Menge als sortierte Liste würde also schlechtestenfalls (wie im Falle der Suche nach dem Namen Zimmermann im Telefonbuch) keinen Vorteil gegenüber der Darstellung als unsortierte Liste bringen.
- Die Verwendung von sortierten Listen hätte sogar einen großen Nachteil:

  Den Aufwand für die Sortierung der Elemente.

Die Menge als binärer Suchbaum

- Das Prinzip soll am Beispiel des Telefonbuchs erläutert werden.
- Dazu muss das Beispiel verallgemeinert werden:
  - Der vom Telefonbuch abgedeckte Namensbereich ist unbekannt (es ist z.B. nicht sicher, dass es für jeden möglichen Anfangsbuchstaben Namen gibt).
  - Die Namensverteilung kann völlig beliebig sein.
  - Man kann zusätzlich annehmen, dass das Buch statt Namen (und Telefonnummern) lediglich ganze (negative sowie positive) Zahlen enthält, die beliebig klein oder groß sein können.
- Eine gegebene Zahl findet man am schnellsten dadurch, dass man das Buch (ungefähr) in seiner Mitte aufschlägt und das Verfahren im linken oder rechten Buchteil (rekursiv) wiederholt, je nach dem, ob die Zahl in der Mitte des Buches größer oder kleiner als die gesuchte Zahl ist.

Nicht-leerer binärer Suchbaum (1)

- Diese Suche lässt sich anhand von nichtleeren Binärbäumen implementieren, wenn nicht nur die Blätter Zahlen (oder sonstige Werte) beinhalten (oder damit „markiert“ sind), sondern auch die sonstigen Knoten:

```
- datatype binbaum3 = Blt of int
                    | Knt of binbaum3 * int * binbaum3;
datatype binbaum3 = Blt of int
                  | Knt of binbaum3 * int * binbaum3
```

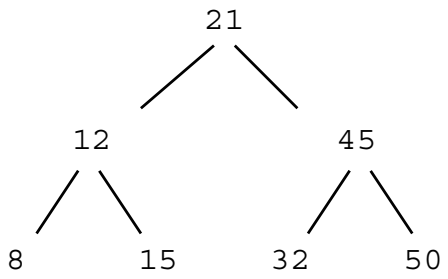
- Dieser Datentyp schließt die leere Menge aus (für manche Anwendungen kann das ungeeignet sein).

Nicht-leerer binärer Suchbaum (2)

```
- val b1 = Knt(Blt(8),12,Blt(15));
val b1 = Knt (Blt 8,12,Blt 15) : binbaum3

- val b2 = Knt(Blt(32),45,Blt(50));
val b2 = Knt (Blt 32,45,Blt 50) : binbaum3

- val b3 = Knt(b1,21,b2);
val b3 = Knt(Knt(Blt #,12,Blt #),21,Knt(Blt #,45,Blt #))
  : binbaum3
```

- Das Zeichen # wird in der gedruckten Mitteilung verwendet, um diese zu kürzen.
- Diese Kürzung betrifft nur die gedruckte Mitteilung und nicht den in der Umgebung gespeicherten Wert des Namens `b3`.

Graphische Darstellung von `b3`



- In diesem Baum gilt für jeden Knoten, dass alle Markierungen im linken Teilbaum kleiner sind als die Markierung des Knotens und alle Markierungen im rechten Teilbaum größer.

Verbesserung des „nicht-leeren“ binären Suchbaums

- Binärbäume vom Typ `binbaum3` haben den Nachteil, nicht alle endlichen Mengen sortiert darstellen zu können.
- Beispiel:

  Die Mengen {1, 2} und {1, 2, 3, 4} können nicht als Binärbäume vom Typ `binbaum3` dargestellt werden.
- Tatsächlich kann man unter Anwendung der strukturellen Induktion beweisen, dass jeder Binärbaum vom Typ `binbaum3` eine ungerade Anzahl von Knotenmarkierungen hat.

Beweis

- Basisfall:

  Ein Binärbaum der Gestalt `Blt(W)` hat genau eine Knotenmarkierung, also eine ungerade Anzahl von Knotenmarkierungen.
- Induktionsfall:

  Seien `B1` und `B2` zwei Binärbäume vom Typ `binbaum3` und `W` eine ganze Zahl.
- Induktionsannahme:

  `B1` und `B2` haben jeweils eine ungerade Anzahl von Knotenmarkierungen.
  Der Binärbaum `Knt(B1,W,B2)` hat $k = |\texttt{B1}| + 1 + |\texttt{B2}|$ Knotenmarkierungen.

  Da $|\texttt{B1}|$ ungerade ist, gibt es $n_1 \in \mathbf{N}$ mit $|\texttt{B1}| = 2 * n_1 + 1$.

  Da $|\texttt{B2}|$ ungerade ist, gibt es $n_2 \in \mathbf{N}$ mit $|\texttt{B2}| = 2 * n_2 + 1$.
  Also $k = (2 * n_1 + 1) + 1 + (2 * n_2 + 1) = 2 * (n_1 + n_2 + 1) + 1$, d.h. k ist ungerade.

qed.

binbaum4 (1)

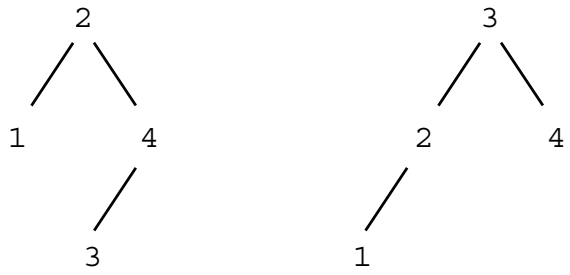
- Der Typ `binbaum4` beseitigt den Mangel der Binärbäume vom Typ `binbaum3`:

```
- datatype binbaum4 = Knt1 of int
                    | Knt2 of int * int
                    | Knt3 of binbaum4 * int * binbaum4;
datatype binbaum4 = Knt1 of int
                  | Knt2 of int * int
                  | Knt3 of binbaum4 * int * binbaum4
```

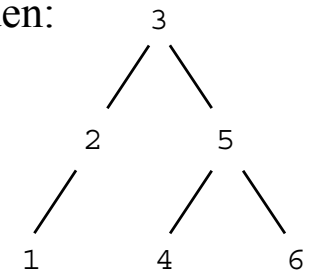

binbaum4 (2)

```
- val c0 = Knt2(1,2);
val c0 = Knt2 (1,2) : binbaum4

- val c1 = Knt3(Knt1(1),2,Knt2(3,4));
val c1 = Knt3 (Knt1 1,2,Knt2(3,4)) : binbaum4

- val c2 = Knt3(Knt2(1,2),3,Knt1(4));
val c2 = Knt3 (Knt2 (1,2),3,Knt1 4) : binbaum4

- val d = Knt3(Knt2(1,2),3,Knt3(Knt1(4),5,Knt1(6)));
val d = Knt3 (Knt2(1,2),3,Knt3(Knt1 #,5,Knt1 #))
: binbaum4
```

Graphische Darstellung von `c1`, `c2` und `d`

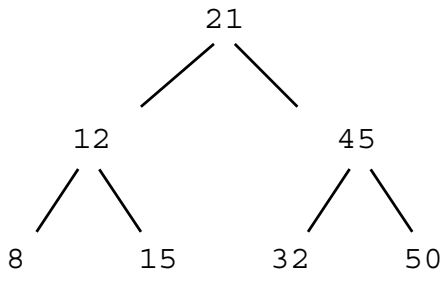


- `c1` und `c2` sind die zwei Möglichkeiten, die Menge {1, 2, 3, 4} als binärer Suchbaum vom Typ `binbaum4` darzustellen.
- Der Binärbaum `d` kann so dargestellt werden:



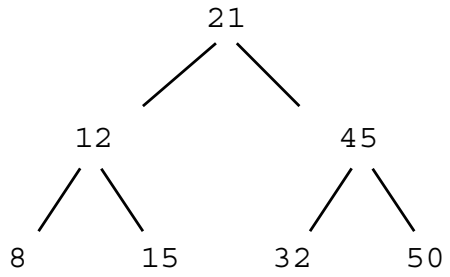
Suche in einem binären Suchbaum vom Typ `binbaum4`

```
- val b1 = Knt3(Knt1(8),12,Knt1(15));
val b1 = Knt3 (Knt1 8,12,Knt1 15) : binbaum4

- val b2 = Knt3(Knt1(32),45,Knt1(50));
val b2 = Knt3 (Knt1 32,45,Knt1 50) : binbaum4

- val b3 = Knt3(b1,21,b2);
val b3 = Knt3 (Knt3 (Knt1 #,12,Knt1 #),21,
               Knt3 (Knt1 #,45,Knt1 #)) : binbaum4
```



Suche nach 25 (bzw. 32) in `b3` (1)

- Da 25 > 21 (bzw. 32 > 21) ist, wird die Suche im rechten Teilbaum `b2` von `b3` fortgesetzt.
- Da 25 < 45 (bzw. 32 < 45) ist, wird die Suche im linken Teilbaum `Knt1(32)` von `b2` fortgesetzt.
- Da 25 ≠ 32 ist, terminiert die Suche erfolglos (bzw. da 32 = 32 ist, terminiert die Suche erfolgreich).



Suche nach 25 (bzw. 32) in `b3` (2)

- Das Verfahren ist nur sinnvoll, wenn die Knotenmarkierungen so sortiert sind, dass für jeden Haupt- oder Teilbaum der Gestalt `Knt2(Markierung1,Markierung2)` gilt:

  `Markierung1 < Markierung2`

  und für jeden Haupt- oder Teilbaum der Gestalt

  `Knt3(LBaum,Markierung,RBaum)` für jeden Knoten `Kl` im linken Teilbaum `LBaum` und für jeden Knoten `Kr` im rechten Teilbaum `RBaum` gilt:

  `Kl < Markierung < Kr`

Eine mögliche Implementierung in SML

```
- fun suche(x, Knt1(M)) = (x = M)
    | suche(x, Knt2(M1, M2)) = (x = M1) orelse (x = M2)
    | suche(x, Knt3(LBaum, M, RBaum)) =
            if x = M
            then true
            else  if x < M
                  then suche(x, LBaum)
                  else suche(x, RBaum);
val suche = fn : int * binbaum4 -> bool

- suche(25, b3);
val it = false : bool

- suche(32, b3);
val it = true : bool

- suche(4, d);
val it = true : bool
```


Zeitbedarf der Suche in einem binären Suchbaum

- Als Problemgröße bietet sich die Anzahl der Knotenmarkierungen an, d.h. die Kardinalität der Menge, die der binäre Suchbaum darstellt.
- Als Zeiteinheit bietet sich die Anzahl der Knoten an, die besucht werden, bis eine Antwort geliefert wird.
- Die Suche nach einer Zahl in einem binären Suchbaum nimmt die folgenden Zeiten in Anspruch:
  - 1 Zeiteinheit, wenn der Binärbaum die Gestalt `Knt1(M)` hat (d.h. aus einem Blatt besteht),
  - 2 Zeiteinheiten, wenn der Binärbaum die Gestalt `Knt2(M1,M2)` hat (d.h. nur eine einzige Kante enthält),
  - höchstens die Anzahl der Knoten entlang eines längsten Astes des Baumes.

Ausgeglichenheit ist vorteilhaft

- Eine Suche in einem binären Suchbaum hat also die besten Zeiten, wenn die Äste des Baumes alle gleich lang sind.
- Längenunterschiede von einem Knoten zwischen zwei Ästen eines Binärbaumes können aber nicht vermieden werden, sonst hätten alle nichtleeren Binärbäume eine ungerade Anzahl an Markierungen (d.h. an Knoten).

Definition (Ausgeglichener Baum)
Ein Binärbaum vom Typ `binbaum4` heißt „ausgeglichen“, wenn sich die Längen zweier beliebiger Äste dieses Baumes um höchstens 1 unterscheiden.

Die Tiefe eines Baums

Definition (Tiefe)
Ein Knoten *k* eines Baumes hat die Tiefe *t*, wenn der Pfad von der Wurzel nach *k* in dem Baum *t* Knoten enthält.
Die Tiefe eines Baumes ist die maximale Tiefe eines seiner Knoten.

Eigenschaften ausgeglichener Binärbäume

Satz A
Die Länge eines Astes eines ausgeglichenen Binärbaums vom Typ `binbaum4`, der n Knoten hat, ist $O(\ln n)$.

- Aus dem Satz A folgt, dass die Suche in einem ausgeglichenen Binärbaum die Zeitkomplexität $O(\ln n)$ hat, wenn n die Kardinalität der Menge ist.
- Der Satz A folgt aus der folgenden Beobachtung:

Satz B
Ein ausgeglichener Binärbaum der Tiefe t kann bis zu 2^t Knoten mit Tiefe t haben.

Induktiver Beweis von Satz B

- Basisfall:

  Die Aussage gilt für Binärbäume der Tiefe 0 und 1.
- Induktionsfall:

  Sei $t \in \mathbf{N}$
- Induktionsannahme:

  Ein ausgeglichener Baum der Tiefe t kann bis zu 2^t Knoten mit Tiefe t haben.

  Ein Binärbaum der Tiefe $t + 1$ besteht aus einer Wurzel mit zwei Nachfolgern, die die Wurzeln von Teilbäumen maximal der Tiefe t sind.

  Jeder dieser Teilbäume kann nach Induktionsannahme bis zu 2^t Knoten mit Tiefe t in dem jeweiligen Teilbaum haben.

  Diese Knoten sind genau die Knoten mit Tiefe $t + 1$ im gesamten Baum, zusammen sind es also bis zu $2 * (2^t) = 2^{t+1}$ Knoten.

qed.

Beweis von Satz A

- Aus dem Satz B folgt, dass ein ausgeglichener Binärbaum der Tiefe *t* maximal $2^0 + 2^1 + \ldots + 2^t = 2^{t+1} - 1$ Knoten hat.
- Ebenso kann man zeigen, dass ein ausgeglichener Binärbaum der Tiefe *t* minimal 2^t Knoten hat.
- Hat also ein ausgeglichener Binärbaum *n* Knoten, so gilt:
  $2^t - 1 < n \leq 2^{t+1} - 1$
  wobei *t* die Tiefe des Baumes ist.
- Daraus folgt:
  $t < \log_2(n + 1) \leq t + 1$
- Das heißt:
  $t \in O(\log_2 n + 1)$, d.h. $t \in O(\ln n)$.

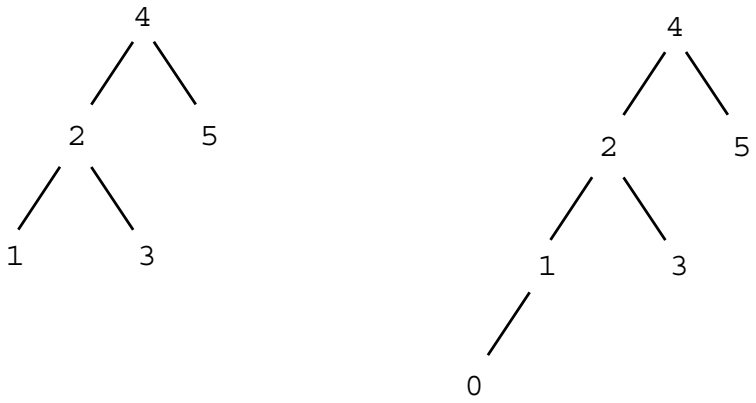
qed.

Preis der Ausgeglichenheit

- Wird die Menge durch Hinzufügen oder Löschen von Elementen verändert, so kann nach und nach der Binärbaum, der diese Menge darstellt, seine Ausgeglichenheit verlieren.
- Man braucht ein Verfahren um die Ausgeglichenheit nach jeder oder nach einigen Änderungen der Menge wiederherzustellen.
- Der Algorithmus dazu ist nicht trivial!

Beispiel

- Durch Einfügen des Wertes `0` entsteht aus dem ausgeglichenen Binärbaum `B1` der Baum `B2`, der nicht mehr ausgeglichen ist:



Überblick

1. Typen im Überblick
2. Deklarationen von Typabkürzungen in SML: type-Deklarationen
3. Definition von Typen: datatype-Deklarationen
4. Definition von rekursiven Typen
5. Beweisprinzip der strukturellen Induktion
6. Beispiele: Implementierungen des Konzepts der „Menge“
7. Beispiele: Grundlegende Funktionen für binäre (Such-) Bäume

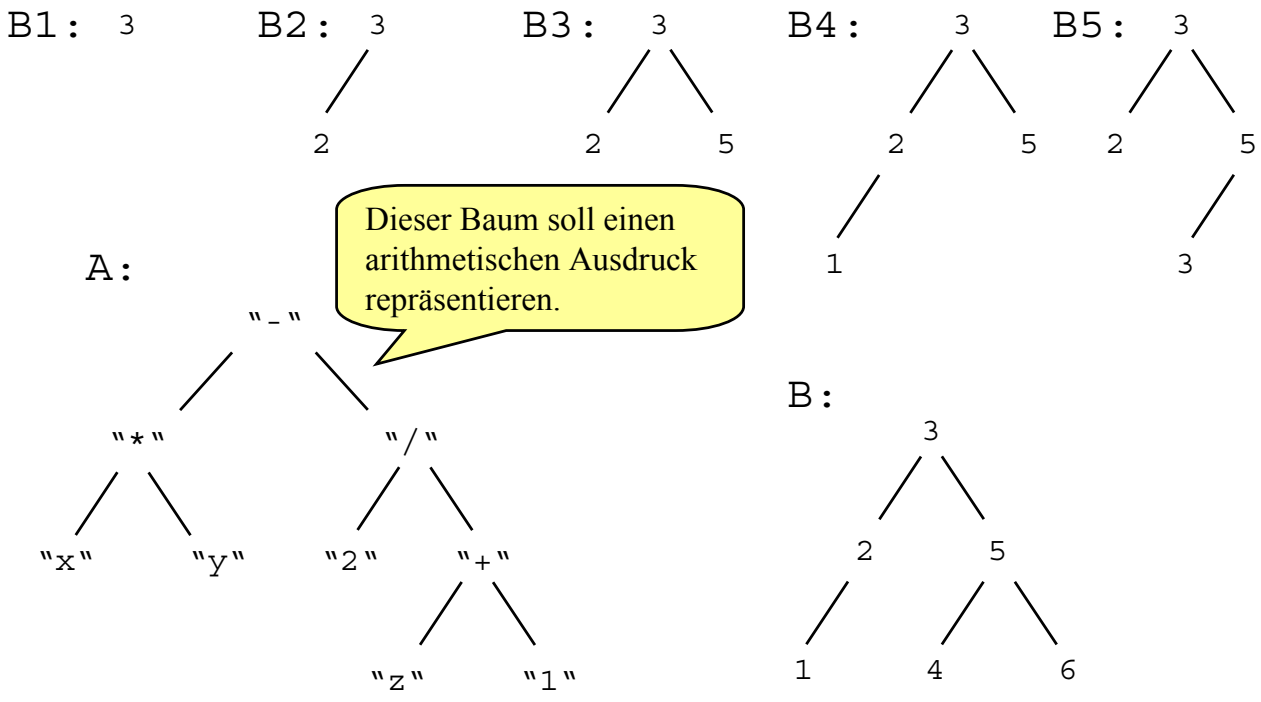
binbaum5

- Die Funktionen dieses Abschnitts beziehen sich auf den folgenden polymorphen Typ „Binärbaum mit Knotenmarkierungen“:

```
- datatype 'a binbaum5 = Leer
                       | Knt1 of 'a
                       | Knt2 of 'a * 'a
                       | Knt3 of 'a binbaum5 * 'a *
                         'a binbaum5;
```

- Wir nehmen an, dass in einem Ausdruck `Knt2(M1,M2)` der Knoten mit Markierung `M1` als linker Nachfolger des Knotens mit Markierung `M2` aufgefasst werden soll.

Graphische Darstellung einiger Binärbäume



B,B1,B2,B3,B4,B5 als Ausdrücke des Typs int binbaum5

```
- val B = Knt3(Knt2(1,2),3,Knt3(Knt1(4),5,
                Knt1(6)));
- val B1 = Knt1(3);
- val B2 = Knt2(2,3);
- val B3 = Knt3(Knt1(2),3,Knt1(5));
- val B4 = Knt3(Knt2(1,2),3,Knt1(5));
- val B5 = Knt3(Knt1(2),3,Knt2(4,5));
```


A als Ausdruck des Typs string binbaum5

```
- val A = Knt3( Knt3 ( Knt1("x"),
                       "*",
                       Knt1("y")
                     ),
                "-",
                Knt3( Knt1("2"),
                      "/",
                      Knt3( Knt1("z"),
                            "+",
                            Knt1("1")
                          )
                    )
              );
```

Ausblick

- In den folgenden Abschnitten werden Funktionen angegeben,
  - die Durchläufen durch Binärbäume (mit Knotenmarkierungen) entsprechen und
  - die Markierungen der Knoten in Listen aufsammeln.

Selektoren und Prädikate (1)

- Selektoren und Prädikate für Binärbäume lassen sich in naheliegender Weise definieren:

Mehr über `exceptions` in Kapitel 10.

```
- exception binbaum5_leer;
- exception binbaum5_kein_nachfolger;

- fun wurzel(Leer) = raise binbaum5_leer
    | wurzel(Knt1(M)) = M
    | wurzel(Knt2(_, M)) = M
    | wurzel(Knt3(_, M, _)) = M;
```

Selektoren und Prädikate (2)

```
- fun linker_baum(Leer) = raise binbaum5_kein_nachfolger
    | linker_baum(Knt1(_))= raise binbaum5_kein_nachfolger
    | linker_baum(Knt2(_,_))=raise binbaum5_kein_nachfolger
    | linker_baum(Knt3(B, _, _)) = B;

- fun rechter_baum(Leer) = raise binbaum5_kein_nachfolger
    | rechter_baum(Knt1(_))= raise binbaum5_kein_nachfolger
    | rechter_baum(Knt2(_,_))= raise
                                  binbaum5_kein_nachfolger
    | rechter_baum(Knt3(_, _, B)) = B;

- fun ist_leer(Leer) = true
    | ist_leer(_) = false;
```


Zugriffe auf Bestandteile von Binärbäumen

```
- wurzel(B);
val it = 3 : int

- wurzel(rechter_baum(B));
val it = 5 : int

- wurzel(linker_baum(A));
val it = "*" : string
```

Da die folgenden Definitionen alle mit Hilfe des Pattern Matching aufgebaut sind, werden die eben besprochenen Funktionen darin nicht benötigt.

Durchlauf in Infix-Reihenfolge (1)

- *Infix-Reihenfolge* bedeutet, dass zunächst die Knoten des linken Teilbaums, dann die Wurzel und anschließend die Knoten des rechten Teilbaums aufgesammelt werden.

Durchlauf in Infix-Reihenfolge (2)

```
- fun infix_collect(Leer) = nil
   | infix_collect(Knt1(M)) = [M]
   | infix_collect(Knt2(M1,M2)) = [M1,M2]
   | infix_collect(Knt3(B1,M,B2)) =
          infix_collect(B1) @ [M] @ infix_collect(B2);
val infix_collect = fn : 'a binbaum5 -> 'a list

- infix_collect(B);
val it = [1,2,3,4,5,6] : int list

- infix_collect(A);
val it = ["x","*","y","-","2","/","z","+","1"]: string list
```

Infix-Reihenfolge

- Die Bezeichnung „*Infix-Reihenfolge*“ wird aus dem Beispiel des Baums `A` leicht verständlich:

  Die Ergebnisliste entspricht dem durch den Baum `A` repräsentierten arithmetischen Ausdruck in Infix-Notation (wobei die Information über die Klammerung verloren gegangen ist).
- Die rechte Seite des letzten Falls der Definition könnte eigentlich umformuliert werden:

  ```
  infix_collect(B1) @ M :: infix_collect(B2)
  ```
- In der Definition wurde trotzdem die umständlichere Form

  ```
  infix_collect(B1) @ [M] @ infix_collect(B2)
  ```

  geschrieben, weil dadurch Analogien und Unterschiede zu den folgenden Funktionen offensichtlicher werden.

Durchlauf in Präfix-Reihenfolge (1)

- *Präfix-Reihenfolge* bedeutet, dass die Wurzel vor den Knoten des linken Teilbaums, und diese vor den Knoten des rechten Teilbaums aufgesammelt werden.
- Erinnerung:

  In einem Ausdruck `Knt2(M1,M2)` ist der Knoten `M1` linker Nachfolger des Knotens `M2`.

Durchlauf in Präfix-Reihenfolge (2)

```
- fun praefix_collect(Leer) = nil
    | praefix_collect(Knt1(M)) = [M]
    | praefix_collect(Knt2(M1,M2)) = [M2,M1]
    | praefix_collect(Knt3(B1,M,B2)) =
          [M] @ praefix_collect(B1) @ praefix_collect(B2);
val praefix_collect = fn : 'a binbaum5 -> 'a list

- praefix_collect(B);
val it = [3,2,1,5,4,6] : int list

- praefix_collect(A);
val it = ["-","*","x","y","/","2","+","z","1"]: string list
```


Präfix-Reihenfolge

- Die Kenntnis der Stelligkeit der Operationen reicht aus, um aus der „Linearisierung“ in Präfix-Reihenfolge die im Baum `A` repräsentierte Klammerung wiederherzustellen.
- Auch hier wäre es angebracht

  ```
  [M] @ praefix_collect(B1)
  ```

  zu vereinfachen zu

  ```
  M :: praefix_collect(B1).
  ```

Durchlauf in Postfix-Reihenfolge (1)

- In *Postfix-Reihenfolge* werden zunächst die Knoten des linken Teilbaums, dann des rechten Teilbaums, und schließlich die Wurzel aufgesammelt.

Durchlauf in Postfix-Reihenfolge (2)

```
- fun postfix_collect(Leer) = nil
   | postfix_collect(Knt1(M)) = [M]
   | postfix_collect(Knt2(M1,M2)) = [M1,M2]
   | postfix_collect(Knt3(B1,M,B2)) =
        postfix_collect(B1) @ postfix_collect(B2) @ [M];
val postfix_collect = fn : 'a binbaum5 -> 'a list

- postfix_collect(B);
val it = [1,2,4,6,5,3] : int list

- postfix_collect(A);
val it = ["x","y","*","2","z","1","+","/","-"] : string list
```

Postfix-Reihenfolge

- Die Kenntnis der Stelligkeit der Operationen reicht aus, um aus der „Linearisierung“ in Postfix-Reihenfolge die im Baum `A` repräsentierte Klammerung wiederherzustellen.

Infix-/ Präfix-/ Postfix-Reihenfolge mit Akkumulatortechnik

- Diese Definitionen haben ein ähnliches Manko wie die Funktion `naive-reverse` in Kapitel 5:

  Durch den Aufruf von `append` in jedem rekursiven Aufruf summiert sich die Gesamtanzahl der Aufrufe von „`::`“ auf einen unnötig hohen Wert.
- Die Funktionen sammeln zwar die Markierungen in der gewünschten Reihenfolge auf, aber verschachtelt in Listen, die dann erst wieder von `append` (bzw. „@“) zu einer „flachen“ Liste zusammengefügt werden müssen.
- Ähnlich, wie mit Hilfe der Akkumulator-Technik aus der Funktion `naive-reverse` die Funktion `reverse` entwickelt wurde, kann man auch hier mit Hilfe der Akkumulator-Technik bessere Definitionen entwickeln.

infix_collect, collect

```
- fun infix_collect(B) =
  let
      fun   collect(Leer,L) = L
      |     collect(Knt1(M),L) = M :: L
      |     collect(Knt2(M1,M2),L) = M1 :: M2 :: L
      |     collect(Knt3(B1,M,B2),L) = collect(B1,M ::
                                            collect(B2,L))
  in
      collect(B, nil)
  end;
val infix_collect = fn : 'a binbaum5 -> 'a list
```


Nicht endrekursiv!

- Die Reihenfolge, in der die Parameter `B1`, `M`, `B2` in den rekursiven Aufrufen weitergereicht werden, ist jeweils die gleiche wie in den vorhergehenden Definitionen.
- Die lokale Hilfsfunktion `collect` ist trotz der Verwendung der Akkumulator-Technik in allen drei Fällen nicht endrekursiv:

  Einer der rekursiven Aufrufe kommt innerhalb eines zusammengesetzten Ausdrucks vor!
- Die Definition eines iterativen Prozesses zum Durchlauf durch beliebige Binärbäume ist äußerst schwierig und erfordert zusätzliche, aufwändige Datenstrukturen zur Verwaltung.

Tiefendurchlauf (Depth-First-Durchlauf)

- Infix-, Präfix- und Postfix-Reihenfolge haben eine Gemeinsamkeit:
  Die Teilbäume des Baumes werden unabhängig voneinander durchlaufen.
- Infix-, Präfix- und Postfix-Reihenfolge haben einen Unterschied:
  Das Ergebnis wird jeweils unterschiedlich zusammengesetzt.
- Die Gemeinsamkeit der drei Durchlaufreihenfolgen ist als „*Tiefendurchlauf* “ bekannt und kann mit Hilfe von Funktionen höherer Ordnung leicht als Abstraktion der drei ursprünglichen Funktionen definiert werden.

depth_first_collect

```
- fun depth_first_collect f0 f1 f2 f3 Leer = f0
    | depth_first_collect f0 f1 f2 f3 (Knt1(M)) = f1(M)
    | depth_first_collect f0 f1 f2 f3 (Knt2(M1,M2)) =
      f2(M1,M2)
    | depth_first_collect f0 f1 f2 f3 (Knt3(B1,M,B2)) =
      f3(depth_first_collect f0 f1 f2 f3 B1,
         M,
         depth_first_collect f0 f1 f2 f3 B2
         );
```

infix_collect

```
- fun infix_collect(B) =
      depth_first_collect nil
                          (fn M => [M])
                          (fn (M1,M2) => [M1,M2])
                          (fn (R1,M,R2) => R1 @ [M] @ R2)
                           B;
```


praefix_collect

```
- fun praefix_collect(B) =
      depth_first_collect nil
                          (fn M => [M])
                          (fn (M1,M2) => [M2,M1])
                          (fn (R1,M,R2) => [M] @ R1 @ R2)
                           B;
```

postfix_collect

```
- fun postfix_collect(B) =
      depth_first_collect nil
                          (fn M => [M])
                          (fn (M1,M2) => [M1,M2])
                          (fn (R1,M,R2) => R1 @ R2 @ [M])
                           B;
```


anzahl_knoten

- Auch andere nützliche Funktionen auf Binärbäumen lassen sich mit Hilfe des Tiefendurchlaufs leicht implementieren:

```
- fun anzahl_knoten(B) =
      depth_first_collect 0
                          (fn M => 1)
                          (fn (M1,M2) => 2)
                          (fn (R1,M,R2) => R1 + 1 + R2)
                           B;
```

anzahl_blaetter tiefe

```
- fun anzahl_blaetter(B) =
      depth_first_collect 0
                          (fn M => 1)
                          (fn (M1,M2) => 1)
                          (fn (R1,M,R2) => R1 + R2)
                           B;

- fun tiefe(B) =
      depth_first_collect 0
                          (fn M => 1)
                          (fn (M1,M2) => 2)
                          (fn (R1,M,R2) => 1 + Int.max(R1,R2))
                           B;
```


element

```
- fun element(x,B) =
      depth_first_collect false
                          (fn M => x=M)
                          (fn (M1,M2) => x=M2 orelse x=M1)
                          (fn (R1,M,R2) => x=M orelse R1
                           orelse R2)
                           B;
```

Aufrufe (1)

```
- anzahl_knoten(B);
val it = 6 : int

- anzahl_knoten(A);
val it = 9 : int

- anzahl_blaetter(B);
val it = 3 : int

- anzahl_blaetter(A);
val it = 5 : int

- tiefe(B);
val it = 3 : int
```


Aufrufe (2)

```
- tiefe(A);
val it = 4 : int

- element(5, B);
val it = true : bool

- element(7, B);
val it = false : bool

- element("2", A);
val it = true : bool

- element("3", A);
val it = false : bool
```

Breitendurchlauf (Breadth-First-Durchlauf)

- Bei einem *Breitendurchlauf* werden die Knoten nach wachsender Tiefe besucht:

  Zuerst wird die Wurzel aufgesammelt, dann die Wurzeln der Nachfolger, dann die Wurzeln von deren Nachfolgern usw.
- Das Ergebnis des Breitendurchlaufs ist für Baum `B` die Liste `[3,2,5,1,4,6]`

  und für Baum `A` die Liste

  `["-","*","/","x","y","2","+","z","1"]`.
- Zur Implementierung kann man eine Hilfsfunktion `entwurzeln` verwenden, die (angewandt auf eine Liste von Bäumen) die Wurzel jedes Baumes aufsammelt und die Nachfolger der Wurzel am Ende der Liste für die spätere Weiterverarbeitung einfügt.

Einelementige Liste von Binärbäumen

```
[
  Knt3( Knt3(
              Knt1("a"),
              "*",
              Knt1("b")
              ),
        "+",
        Knt3(
              Knt1("e"),
              "-",
              Knt1("f")
              )
      )
]
```

Aus einer *einelementigen* Liste von Binärbäumen soll nach dem Aufsammeln von „+“ die *zweielementige* Liste auf der nächsten Folie entstehen.

Zweielementige Liste von Binärbäumen

```
[
  Knt3(
       Knt1("a"),
       "*",
       Knt1("b")
       ),
  Knt3(
       Knt1("e"),
       "-",
       Knt1("f")
       )
]
```


`entwurzeln`

- Die Funktion `entwurzeln` zerlegt einen Baum von der Wurzel her, gerade entgegengesetzt zum Aufbau des Baums gemäß der mathematischen Definition oder mit den Konstruktoren des induktiv definierten Typs `'a binbaum5`.
- Der Unterschied zu den Selektoren ist:

  Die Teilbäume werden nicht einfach als Ergebnisse geliefert, sondern in einer Liste nach dem Prinzip *first-in-first-out* verwaltet.

breadth_first_collect

```
- fun breadth_first_collect(B) =
let
  fun entwurzeln nil = nil
    | entwurzeln(Leer::L) = entwurzeln(L)
    | entwurzeln(Knt1(M)::L) = M::entwurzeln(L)
    | entwurzeln(Knt2(M1,M2)::L) = M2::entwurzeln(L @ [Knt1(M1)])
    | entwurzeln(Knt3(B1,M,B2)::L) = M::entwurzeln(L @ [B1,B2])
in
  entwurzeln(B::nil)
end;
val breadth_first_collect = fn : 'a binbaum5 -> 'a list

- breadth_first_collect(B);
val it = [3,2,5,1,4,6] : int list

- breadth_first_collect(A);
val it = ["-","*","/","x","y","2","+","z","1"] : string list
```


Breitendurchlauf

- Beim Breitendurchlauf werden die Teilbäume nicht unabhängig voneinander durchlaufen, sondern nach Tiefe verzahnt.
- Das verletzt die Grundannahme des Tiefendurchlaufs, so dass der Breitendurchlauf nicht mit Hilfe des Tiefendurchlaufs definiert werden kann.