Begleitmaterial zur Vorlesung Grundlagen der Programmierung I im WS 2003/2004

Prof. Dr. Andreas Schwill

(Fassung vom 18.4.2007)

Einführung in ML

1 Überblick

ML (Abk. für Meta Language) ist eine funktionale Programmiersprache, die Ende der 70er Jahre an der University of Edinburgh entwickelt und 1987 standardisiert wurde. Wir stellen hier Standard ML (SML) vor. Die wichtigsten Eigenschaften von ML sind:

- interaktive Programmentwicklung durch inkrementellen Übersetzer
- strenge und statische Typisierung
- strikte Auswertung von Ausdrücken (mit geringen Einschränkungen), Parameterübergabe: call-by-value
- leistungsfähiges Typinferenzsystem (daher können fast alle Typangaben unterbleiben)
- Polymorphie bei Typen und Funktionen
- übersichtlicher Zugriff auf Datenstrukturen mithilfe von Pattern-matching anstelle von oder zusätzlich zu Selektoren
- benutzergesteuerte Behandlung von Laufzeitfehlern (exception handling)
- umfangreiches Modulkonzept.

2 Elementare Datentypen und Funktionen

2.1 Bezeichner

ML unterscheidet zwischen *alphabetischen* Bezeichnern und *symbolischen* Bezeichnern. Alphabetische Bezeichner beginnen mit einem Buchstaben gefolgt von einer *beliebigen* (!) Zahl von Buchstaben, Ziffern, Unterstrichen und Hochkommata. Groß- und Kleinbuchstaben werden unterschieden.

Symbolische Bezeichner bestehen aus einer beliebigen Folge von Zeichen aus dem Zeichenvorrat

```
! % & $ # + - * / : < = > ? @ \ ~ ' ^ |
```

Einige symbolische Zeichenfolgen besitzen eine besondere Bedeutung und dürfen nicht als Bezeichner verwendet werden. Dies sind:

```
: _ | = => -> #
```

2.2 Elementare Datentypen

ML verfügt über die folgenden elementaren Standarddatentypen:

Name	Beispiele für Konstanten	Operationen
unit	()	keine
bool	true, false	not, =, <>
int	17, 0, ~23, 001	+, -, *, div, mod, abs, =, <>, <, >, >=, <=
real	0.01, 3.1415, ~1.6E3, 7E~5	+, -, *, /, sin, cos, tan, ln, exp, sqrt, =, <>, <, >, >=, <=
string	"O.K.", "Apfel", "ich sagte:\"Nein!\""	size (Länge des strings), ^ (Konkatenation), chr, ord

Erläuterungen: () ist einziges Element des Datentyps unit. ~ ist das Vorzeichen Minus. Ein Vorzeichen Plus existiert nicht. \ in Strings symbolisiert den Beginn einer sog. escape-Sequenz. Oben zeigt z. B. \" an, daß das Zeichen " hier als Teil des Strings und nicht als Begrenzer des Strings zu betrachten ist.

Wegen der strengen Typisierung von ML sind die Typen int und real disjunkt. Folglich kann ein int-Objekt nicht mit einem real-Objekt arithmetisch verknüpft werden. Hier sind vorher *Typanpassungen* vom Benutzer vorzunehmen. Sie erfolgen nicht, wie in anderen Sprachen, automatisch. Die zugehörigen Anpassungsfunktionen lauten:

real: int→real bzw.

floor: real→int.

floor(x) liefert die größte ganze Zahl ≤x.

Ferner beachte man, daß es im Typ bool keine and- und keine or-Funktion gibt. Stattdessen stellt ML zwei Infix-Funktionen zur Verfügung:

andalso: bool×bool→bool mit

false, falls x=false

x andalso y =

y, sonst.

orelse: bool×bool→bool mit

true, falls x=true

x orelse y =

y, sonst.

Das zweite Argument wird nur ausgewertet, wenn es für den Wert des Ausdrucks noch relevant ist. Beide Funktionen sind also nicht strikt, da z.B. (true orelse y) auch einen Wert liefert, wenn y nicht definiert ist.

2.3 Ausdrücke

Ein Ausdruck besteht entweder aus einem *einfachen Ausdruck*, der über den elementaren Datentypen mittels der vordefinierten oder selbstdefinierter Funktionen in der üblichen Weise gebildet worden ist, aus einem *bedingten Ausdruck* oder aus einem *Funktionsausdruck*.
Der bedingte Ausdruck hat die Form

$$\underline{\text{if}}\ B\ \underline{\text{then}}\ E\ \underline{\text{else}}\ E' = \begin{cases} E, & \text{falls } B \text{ der Wert } \underline{\text{true}} \text{ hat,} \\ E', & \text{sonst.} \end{cases}$$

Analog zu andalso und orelse ist auch die Funktion if-then-else nicht strikt, denn es wird entweder der then-Zweig oder der else-Zweig ausgewertet, aber nie beide. So liefert der bedingte Ausdruck also auch einen Wert, wenn der nicht benutzte Zweig undefiniert ist. Alle übrigen Ausdrücke werden in ML strikt ausgewertet.
Beispiele für elementare und bedingte Ausdrücke:

```
5,  7+4,  3.14*real(17),  if n>0 then 1 else if n=0 then 0 else -1,   sqrt(x+y),
flaeche x*y,   is_digit d.
```

Hier wird vorausgesetzt, daß die Bezeichner flaeche und is_digit vorher als Funktionen definiert wurden.

In ML werden Funktionen ebenso wie Werte der elementaren Typen als Werte eines Funktionstyps aufgefaßt. Man kann daher auch Ausdrücke bilden, die Funktionen als Argumente besitzen und Funktionen als Werte liefern. Einen Funktionsausdruck einfachster Form schreibt man durch

```
fn <Bezeichner> => <Ausdruck>
```

z.B.

```
fn x => 2*x.
```

Der Wert dieses Ausdrucks ist eine (namenlose) Funktion vom Typ int→int mit <Bezeichner> als formalem Parameter und <Ausdruck> als Rumpf. Anstelle des Bezeichners kann man auch ein Pattern (Parametermuster) angeben, doch dazu später.
Beispiele für Funktionsausdrücke sind:

```
fn x => if x>0 then 1 else
      if x=0 then 0 else -1,
fn f => (fn g => (fn x => f(g(x)))).
```

Im zweiten Fall handelt es sich um ein polymorphes Funktional des Typs

$$(\Delta\to\Delta')\to((\Delta''\to\Delta)\to(\Delta''\to\Delta')).$$

Funktionsausdrücke kann man auf natürliche Weise auf aktuelle Parameter anwenden und erhält ein Ergebnis, das wiederum eine Funktion sein kann.

Beispiele für Funktionsanwendungen:

```
(fn x => 2*x) 7
```

wird ausgewertet zu 14:int,

```
(fn f => (fn g => (fn x => f(g(x))))) sin cos
```

wird ausgewertet zu

```
fn x => sin(cos(x)) : real→real.
```

2.4 Benutzung des ML-Systems

Wir können nun unsere ersten einfachen Versuche starten, mit dem ML-System in Kontakt zu treten. Zunächst zum Aufruf: Das UNIX-Kommando

```
/opt/ml/sml
```

startet das System, welches sich mit dem prompt-Symbol "-" meldet. Nun kann man beliebige ML-"Kommandos" (Deklarationen, Ausdrücke) eingeben. Jede Eingabe wird durch ein Semikolon abgeschlossen. Die Eingabe wird sofort ausgewertet und das Ergebnis dem Benutzer mitgeteilt. Ergebnis ist dabei z.B. der Wert eines Ausdrucks mit seinem Datentyp oder der Typ einer soeben deklarierten Funktion. Systemausgaben beginnen immer mit dem Symbol >.

Beispiel:

```
7+4;
> 11: int
sqrt(2.0);
> 1.4142: real
sqrt;
> fn: real -> real
```

Hier zeigt ML an, daß es sich bei sqrt um eine Funktion (besser: einen Funktionsausdruck) vom Typ real→real handelt.

```
"Was"^"soll"^"das?";
> "Wassolldas?": string
ord;
> fn: string -> int
ord "Was soll das?"
> 119: int
if ord "Was soll das?"=97 then 1 else 0;
> 0: int
```

Hat man bereits eine Datei mit einem ML-Programm erstellt, so wird die Datei mit dem Funktionsaufruf

use "Dateiname"

in das ML-System eingelesen.

2.5 Wertdeklarationen

In ML können Werte eines beliebigen ML-Typs deklariert werden. Werte können also nicht nur Elemente der elementaren Typen sein, sondern auch Listen Tupel, Bäume usw. sowie Funktionen und Funktionale.

Wertdeklarationen besitzen die allgemeine Form

val <Bezeichner> = <Ausdruck>.

Hierdurch wird der Wert des Ausdrucks an den Bezeichner gebunden.

Beispiele:

1)
```
val sekunden=60;
> val sekunden=60:int;
val minuten=sekunden;
> val minuten=60:int
val stunden=24;
> val stunden=24:int
val sekunden_pro_tag=sekunden*minuten*stunden;
> val sekunden_pro_tag=86400: int
val doppel=fn x => 2*x;
> val doppel = fn: int -> int
```

Hier hat das ML-System also eine Funktionsdefinition erkannt und aus den Typen der im Ausdruck vorkommenden Objekte geschlossen, daß die Funktion den Typ int→int besitzt. Warum? Die Zahl 2 ist offenbar ein int-Objekt. Wegen der strengen Typisierung von ML kann ein int-Objekt nur mit einem int-Objekt arithmetisch verknüpft werden. Folglich muß auch x vom Typ int sein.

Wir haben in diesem Beispiel den ersten Typ-Konstruktor von ML kennengelernt: die Bildung von Funktionenräumen. Sind D und D' beliebige Typen, so bezeichnet D→D' die Menge aller Funktionen von D nach D'.

2)
```
val apply = fn f => (fn g => (fn x => f(g(x))))
> val apply=fn: ('a -> 'b) -> (('c -> 'a) -> ('c -> 'b))
```

Hier wurde eine polymorphe Funktion definiert, deren Typ

('a -> 'b) -> (('c -> 'a) -> ('c -> 'b))

ML inferiert hat. Dabei sind 'a, 'b und 'c Typvariablen. Typvariablen, die wir bisher immer durch große griech. Buchstaben (meist Δ) bezeichnet haben, werden in ML also durch ein vorangestelltes Hochkomma markiert. Bei der Anwendung auf konkrete Objekte inferiert ML den genauen Typ und bindet die Typvariablen entsprechend.

Man beachte, daß apply vollständig gecurryt ist.

```
doppel sekunden;
> 120: int
apply sin cos;
> fn: real -> real
```

Hier hat ML die Typvariablen 'a, 'b und 'c von apply jeweils an den Typ real gebunden.

```
apply sin cos 0.5;
> 0.7691: real
```

Wegen der relativ unübersichtlichen Darstellung von Wertdeklarationen, an denen Funktionen beteiligt sind, und weil diese Funktionen nur einen Parameter besitzen dürfen, gibt es in ML die abkürzende Darstellung:

fun <Bezeichner> <formale Parameterliste> = <Ausdruck>.

Beispiele:

1) Wir deklarieren eine Funktion zur Volumenberechnung eines Zylinders. Zunächst die Kreisfläche:

```
val pi=3.1415;
> val pi=3.1415: real
fun flaeche r=pi*r*r;
> val flaeche=fn: real -> real
```

Nun berechnen wir

```
flaeche 17.0;
> 907.8935: real
flaeche (2.0/3.0);
> 1.396: real
```

Nun die Funktion für das Zylindervolumen:

```
fun volumen r h=(flaeche r)*h;
> val volumen=fn: real -> (real -> real)
volumen 17.0 3.0;
> 2723.6805: real
```

Hier ist volumen vollständig gecurryt. Probieren wir noch die ungecurryte Version, wo die Parameter der Funktion als Paar übergeben werden:

```
fun volumen (r,h)=(flaeche r)*h;
> val volumen=fn: real * real -> real
volumen (17.0,3.0);
> 2723.6805: real
```

Nun kennen wir einen weiteren Typ-Konstruktor, die Aggregation: Sind D, D' beliebige Typen, so ist D*D' das kartesische Produkt von D und D', also die Menge aller Paare (d,d')∈D×D'.

In der Vorlesung hatten wir als einen Vorteil des Currying die partielle Auswertung von Funktionen genannt: Möchte man das Volumen mehrerer Zylinder gleicher Grundfläche 12 berechnen, so definiert man eine neue Funktion volumen12, die man aus volumen durch partielle Auswertung erhält:

```
val volumen12=volumen 12.0;
> val volumen12=fn: real -> real
```

2) Rekursive Funktionen definiert man auf natürliche Weise:

```
fun fak x= if x=0 then 1 else x*fak(x-1);
> val fak=fn: int -> int
fak 5;
> 120: int
```

Statische Bindung.

In ML werden bei einer Deklaration Bezeichner *statisch* an Werte gebunden.

Typeinschränkungen.

Wir definieren eine neue Funktion flaeche für die Fläche eines Rechtecks:

```
fun flaeche a b=a*b;
> Unresolvable overloaded identifier: *
```

Mit dieser Definition bekommt ML Probleme. Das leistungsfähige Typinferenzsystem ist hier nicht in der Lage, den Typ der Funktion zu ermitteln. Warum? Die Multiplikation ist *überladen*; sie erfüllt zwei Aufgaben: Einerseits ist sie eine Operation vom Typ int*int→int, aber auch vom Typ real*real→real. Häufig geht aus dem Zusammenhang, in dem diese Operation verwendet wird, hervor, welcher Typ gemeint ist (etwa bei pi*r*r: pi ist vom Typ real, also auch der Gesamtausdruck). In den wenigen übrigen Fällen muß der Benutzer selbst festlegen, welche Operation er meint. Hierzu dienen Typeinschränkungen. Eine Typeinschränkung besitzt die Form

```
: <Typ>
```

und kann an fast jeder Stelle innerhalb einer Deklaration verwendet werden, um den Typ des vorangehenden Objekts festzulegen. Möglicher Einsatz in obigem Beispiel ist etwa:

```
fun flaeche (a:real) b=a*b;
fun flaeche (a:real) (b:real)=a*b;
fun flaeche a b=a*b :real;
```

Im ersten Fall wird nur x auf real eingeschränkt, im zweiten x und y und im dritten nur der Typ des Ergebnisses. Aus jeder dieser drei Vorgaben kann das ML-System den Typ der übrigen Objekte inferieren, so daß in allen drei Fällen die Ausgabe lautet:

```
> val flaeche=fn: real -> real
```

Beispiel: Das folgende Beispiel zeigt noch einmal in der Gesamtschau den Umgang mit Typeinschränkungen, Funktionen als Werten und partieller Auswertung:

```
fun arithmetik operator :int =
      if operator="+" then fn (x,y) => x+y else
            if operator="-" then fn (x,y) => x-y else fn (x,y) => x*y;
> val arithmetik=fn: string -> ((int*int) -> int)
val add = arithmetik "+";
> val add=fn: int*int -> int
val sub = arithmetik "-";
> val sub=fn: int*int -> int
add(2,7);
> 9: int
sub(13,6);
> 7: int
```

3 Strukturierte Datentypen

Im letzten Abschnitt haben wir schon zwei der wichtigsten Typkonstruktoren von ML angeschnitten, die Bildung von Paaren

$$D * D'$$

und die Bildung von Funktionsräumen

$$D \rightarrow D'$$

für zwei beliebige Datentypen D und D'.
Im folgenden besprechen wir diese und die übrigen Konstruktoren genauer.

3.1 Enumeration

Man definiert man einen Aufzählungstyp D schematisch durch

```
datatype D= d1 | d2 | ... | dn.
```

$d_1, d_2, ..., d_n$ sind die Elemente des Wertebereichs von D. Hierbei ist jedes d_i explizit anzugeben. Faktisch ist die Enumeration in ML nur ein Spezialfall der Generalisation (s. 3.5).
Beispiele:

1)
```
datatype farbe = blau | gruen | rot;
> datatype farbe = blau | gruen | rot
      con rot=rot: farbe
```

```
con blau=blau: farbe
con gruen=gruen: farbe
```

2) Der elementare Standardtyp bool ist ein Aufzählungstyp und definiert durch

```
datatype bool = true | false;
```

Die zugehörige not-Funktion lautet:

```
fun not x = if x then false else true;
```

3.2 Aggregation

ML besitzt kein besonderes Sprachelement für die Unterscheidung von homogenen und inhomogenen Aggregationen, wie sie in anderen Sprachen üblich sind (array versus record). Vielmehr unterscheidet ML zwischen

- anonymen unter Verwendung des polymorphen Standardkonstruktors $(\cdot,\cdot,...,\cdot)$ gebildeten **Tupeln**,
- selbstdefinierten mit einem Bezeichner versehenen **Tupeln**,
- **Records**.

Nur die selbstdefinierten Tupel mit Bezeichner sind neu; die beiden anderen Konstruktoren kennen wir bereits aus FUN.

Der Unterschied zwischen Tupeln und Records besteht bekanntlich darin, daß die einzelnen Komponenten bei Records durch frei wählbare Bezeichner selektiert werden können. Bei Tupeln ist dagegen kein direkter Zugriff auf die Komponenten möglich, sie können nur indirekt über das sog. Pattern-Matching (s. 3.3) selektiert werden.

Tupelbildung.

Sind $d_1,...,d_n$ Werte der beliebigen Datentypen $D_1,...,D_n$, so bildet man durch

$$(d_1,...,d_n)$$

ein Standardtupel des Datentyps

$$D=D_1\times...\times D_n.$$

D definiert man in ML durch

```
type D=D1*...*Dn.
```

Beispiele:

1) Bruchrechnung:

```
type rational=int*int
(2,7);
> (2,7): int*int
```

Hier kann ML natürlich nicht unterscheiden, ob es sich bei dem Paar um eine rationale Zahl im Sinne der Typdefinition oder um ein Paar ganzer Zahlen handeln soll.

```
(2,7): rational;
> (2,7): rational
fun bruchmult((a,b): rational,(a',b'): rational )=(a*a',b*b'): rational;
> val bruchmult=fn: rational * rational -> rational
bruchmult((2,7),(3,6));
> (6,42): rational
```

2) Die polymorphe Funktion paar bildet aus zwei Argumenten x,y zweier beliebiger Datentypen das Paar (x,y):

```
fun paar x y=(x,y);
> val paar=fn: 'a -> ('b -> 'a*'b)
paar true (paar 2 "x");
> (true, (2, "x")): bool*(int*string)
```

3) Die folgende polymorphe Funktion verschiebt die Elemente eines Tripels zyklisch nach rechts:

```
fun cycleshift (x,y,z)=(z,x,y);
> val cycleshift=fn: 'a*'b*'c -> 'c*'a*'b
cycleshift((2,3),18.4,(true,false));
> ((true,false),(2,3),18.4): (bool*bool)*(int*int)*real
```

Selbstdefinierte mit einem Bezeichner versehene Tupeltypen D über den Datentypen $D_1,...,D_n$ definiert man allgemein in der Form

```
datatype D = c of D1*…*Dn.
```

Hierbei ist c der frei gewählte Bezeichner des Konstruktors für Objekte aus D, also eine Abbildung

$$c: D_1 \times \ldots \times D_n \rightarrow D.$$

Beispiel:

```
datatype rational = Bruch of int*int;
> con Bruch=fn: int*int -> rational
Bruch(2,7);
> Bruch(2,7): rational
fun Bruchmult (Bruch(a,b),Bruch(a',b'))=Bruch(a*a',b*b');
> val Bruchmult=fn: rational*rational -> rational
Bruchmult(Bruch(2,7),Bruch(3,6));
> Bruch(6,42): rational
```

Auf die einzelnen Komponenten eines aggregierten Tupeltyps kann man nicht direkt zugreifen. Es gibt also keine *Selektoren*. Ein indirekter Zugriff ist über *Pattern-Matching* möglich. Man verwendet hierbei als formalen Parameter keinen einzelnen Bezeichner, sondern ein Muster.

Die einzelnen Bestandteile des aktuellen Parameters werden dann mit den Symbolen des Musters assoziiert.

Beispiel: In der obigen Funktion Bruchmult haben wir bereits intuitiv als ersten formalen Parameter das Muster

```
Bruch(a,b)
```

verwendet. Der aktuelle Parameter

```
Bruch(2,7)
```

wird beim Aufruf der Funktion mit dem formalen "gematcht"

```
Bruch ( a , b )
   ↓   ↓  ↓
Bruch ( 2 , 7 )
```

Hierbei werden die entsprechenden Variablenbindungen a=2, b=7 hergestellt. Auf die genauen Abläufe beim Pattern-Matching gehen wir in Abschnitt 3.3 ein.

Beispiel: Selektoren eine aggregierten Typs kann man nun simulieren, z.B. durch

```
fun Zaehler(Bruch(a,b))=a;
> val Zaehler=fn: rational -> int
```

oder allgemein für ein Paar durch polymorphe Projektionsfunktionen

```
fun Px(x,y)=x;
> val Px=fn: 'a*'b -> 'a
fun Py(x,y)=y;
> val Px=fn: 'a*'b -> 'b
```

Records.

Einen Recordtyp D über den Typen $D_1,...,D_n$ definiert man schematisch in der Form

```
type D = {s1:D1, s2:D2, ..., sn:Dn}.
```

$s_1,...,s_n$ sind die Bezeichner der einzelnen Komponenten von D (die *Selektoren*).

Beispiel:

```
type rational={Zaehler:int, Nenner:int};
> type rational={Nenner:int, Zaehler:int}
val Bruch1={Zaehler=2, Nenner=7};
> val Bruch1= {Nenner=7, Zaehler=2} : {Nenner:int, Zaehler:int}
val Bruch2={Zaehler=3, Nenner=6};
> val Bruch2= {Nenner=6, Zaehler=3} : {Nenner:int, Zaehler:int}
```

Eine Komponente eines Records selektiert man, indem man dem Komponentenbezeichner ein # voranstellt. Dann erhält man eine Funktion, die als Parameter den Bezeichner eines Records erwartet.

Beispiel:

```
#Zaehler Bruch2;
```

```
> 3:int
val Bruch1malBruch2={Zaehler=(#Zaehler Bruch1) * (#Zaehler Bruch2),
                     Nenner=(#Nenner Bruch1) * (#Nenner Bruch2)};
> val Bruch1malBruch2= {Nenner=18, Zaehler=6} : {Nenner:int, Zaehler:int}
fun Bruchmult(x:rational,y:rational)={Zaehler=(#Zaehler x) * (#Zaehler y),
                     Nenner=(#Nenner x) * (#Nenner y)}: rational;
> val Bruchmult=fn: rational*rational -> rational
```

Bemerkung: Zwischen Standardtupeln und Records besteht in ML eine enge Beziehung: Tupel der Form

$d=(d_1,...,d_n)$: $D_1*\ldots*D_n$

werden implizit als Records aufgefaßt, wobei die Selektoren den Indizes der Tupelelemente entsprechen. Das Tupel d ist also äquivalent zu einer Wertdeklaration mit einem Record der Form

val $d=\{1=d_1,2=d_2,...,n=d_n\}$.

Auf die einzelnen Komponenten eines Tupels kann dann wie bei Records zugegriffen werden, z.B.

```
#3 d;
> d3:D3.
```

In diesem Abschnitt haben wir zwei Formen von Typdeklarationen kennengelernt, die type-Deklaration und die datatype-Deklaration. Während die datatype-Deklaration einen neuen Datentyp zusammen mit benutzerdefinierten Konstruktoren beschreibt, werden bei der type-Deklaration bereits bekannte Datentypen mittels der *Standardkonstruktoren*

$D*D'$	(Tupelbildung)
$D \rightarrow D'$	(Funktionsraumbildung)
$\{s_1:D_1, s_2:D_2, ..., s_n:D_n\}$	(Record-Bildung)

zu neuen Datentypen verknüpft.

3.3 Pattern-Matching

ML stellt mit dem Pattern-Matching ein leistungsfähiges und übersichtliches Konzept zur Parameterübergabe zur Verfügung, mit dem man auf eine explizite Anwendung von Selektoren auf Datenstrukturen und die damit verbundene explizite Fallunterscheidung durch geschachtelte bedingte Ausdrücke in Abhängigkeit vom Aufbau der Datenstruktur weitgehend verzichten kann. Stattdessen gibt man Muster für den aktuellen Parameter an und legt für jedes Muster fest, welchen Wert eine Funktion besitzen soll, wenn der Parameter in das Muster paßt. Die Zuordnung der Bezeichner und Strukturen des aktuellen Parameters an das Muster des

formalen Parameters (*Matching*) wird von ML übernommen. Implementiert wird das Pattern-Matching durch einen effizienten Algorithmus zur Ermittlung der Isomorphie von geordneten, markierten Bäumen.

Beispiele:

1) Wir definieren die Funktion

andalso: bool×bool→bool.

Ohne Nutzung des Pattern-Matching definiert man wie üblich

```
fun andalso(x,y) =  if x then
                        if y then true else false
                    else false.
```

Mit Pattern-Matching schreiben wir stattdessen übersichtlicher

```
fun  andalso(true,true) = true |
     andalso(_,_) = false.
```

Hier haben wir zwei Muster für den aktuellen Parameter vorgegeben, das Muster

(true,true)

und das Muster

(_,_).

Paßt der aktuelle Parameter (a,b) in das erste Muster, also a=b=true, so ist true das Ergebnis der Funktion and. Jede andere Belegung des aktuellen Parameters paßt in das Muster (_,_) und führt zum Funktionswert false. Hierbei steht das Zeichen _ (sog. *wildcard*) für ein nicht spezifiziertes Objekt eines Musters. Es kann mit jedem Objekt gematcht werden.

2) Wir kodieren eine 2×2-Matrix zeilenweise durch ein Paar von Paaren. Eine Funktion, die eine Matrix daraufhin überprüft, ob es sich um die Einheitsmatrix handelt, lautet:

```
fun  einheit ((1,0),(0,1)) = true |
     einheit (_,_) = false.
```

Um zu überprüfen, ob die Elemente der Hauptdiagonalen übereinstimmen, verwendet man:

```
fun  eqdiag ((x,_),(_,y) = (x=y).
```

Abb. 1 veranschaulicht die Abläufe beim Aufruf von eqdiag((7,2),(3,7)).

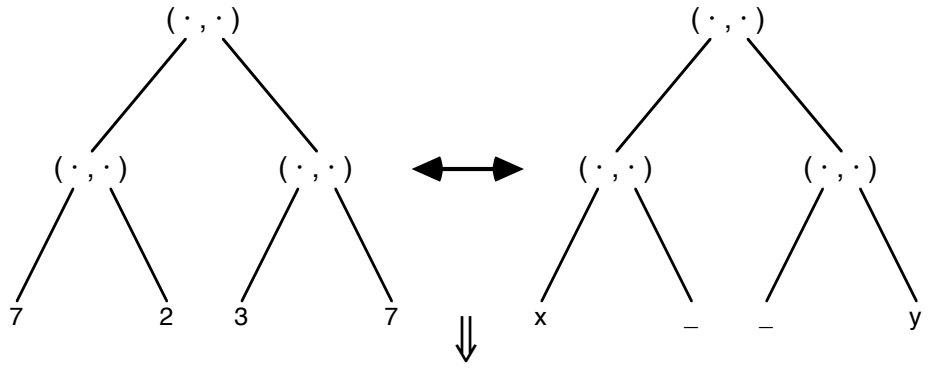


Abb. 1: Pattern-Matching

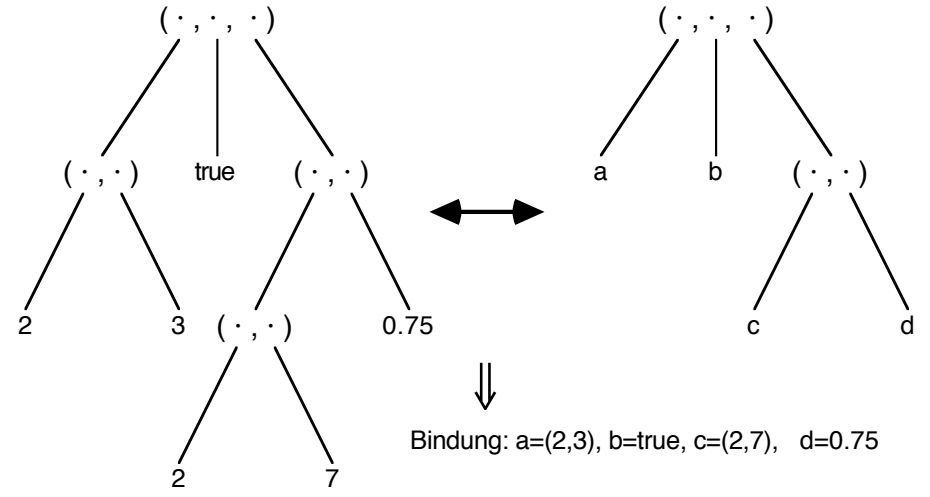


Abb. 2: Pattern-Matching

3) Pattern-Matching funktioniert auch in Verbindung mit Wertdeklarationen (Abb. 2):

val (a,b,(c,d))=((4-2,3),true,((2,7),3.0/4.0)).

Definition:

Ein **Pattern** in ML ist ein Ausdruck, der nur aus Bezeichnern für Variablen, Konstruktoren und dem Zeichen _ (wildcard) besteht. Dabei müssen alle Variablenbezeichner paarweise verschieden sein.

Es ist zu beachten, daß zu den Konstruktoren auch die Konstanten der Grundtypen int, real, bool, string gehören (es sind 0-stellige Konstruktoren) sowie die Standardkonstruktoren

(·,·,...,·) (Tupel)

{·,·,...,·} (Records)

[·,·,...,·] (Listen, s. 3.6).

Wie werden nun die Objekte eines Musters einem konkreten Objekt zugeordnet (gematcht)? Hierfür gelten folgende Regeln:

1. *Variablen:* Variablen können mit jedem Objekt, sei es strukturiert oder elementar, gematcht werden.

Beispiel: fun id x = x.

Hier ist x ein Muster, das mit jedem aktuellen Parameter gematcht werden kann, z.B.

id(2,3) Bindung: x=(2,3)

id({zaehler=2,nenner=3},(2,true)) Bindung: x=({zaehler=2,nenner=3},(2,true)).

2. *Wildcards:* Sie können wie Variablen mit jedem Objekt gematcht werden. Im Unterschied zu Variablen wird jedoch keine Bindung durchgeführt.

Beispiel: fun projx(x,_)=x;

projx(2,(("a","b"),true)) Bindung: x=2.

3. *Konstruktoren:* Ein Konstruktor innerhalb eines Pattern kann nur mit identischen Konstruktoren gematcht werden. Auch hier wird im Erfolgsfall keine Bindung durchgeführt.

Beispiel: fun test(true,(_,(x,{komp1=y,komp2=_}))) = ...

Aufruf: test(2+3=5,(2-7,(2.0/3.0,{komp1=(2,7),komp2=(1,8)}))).

Bindung: Abb. 3.

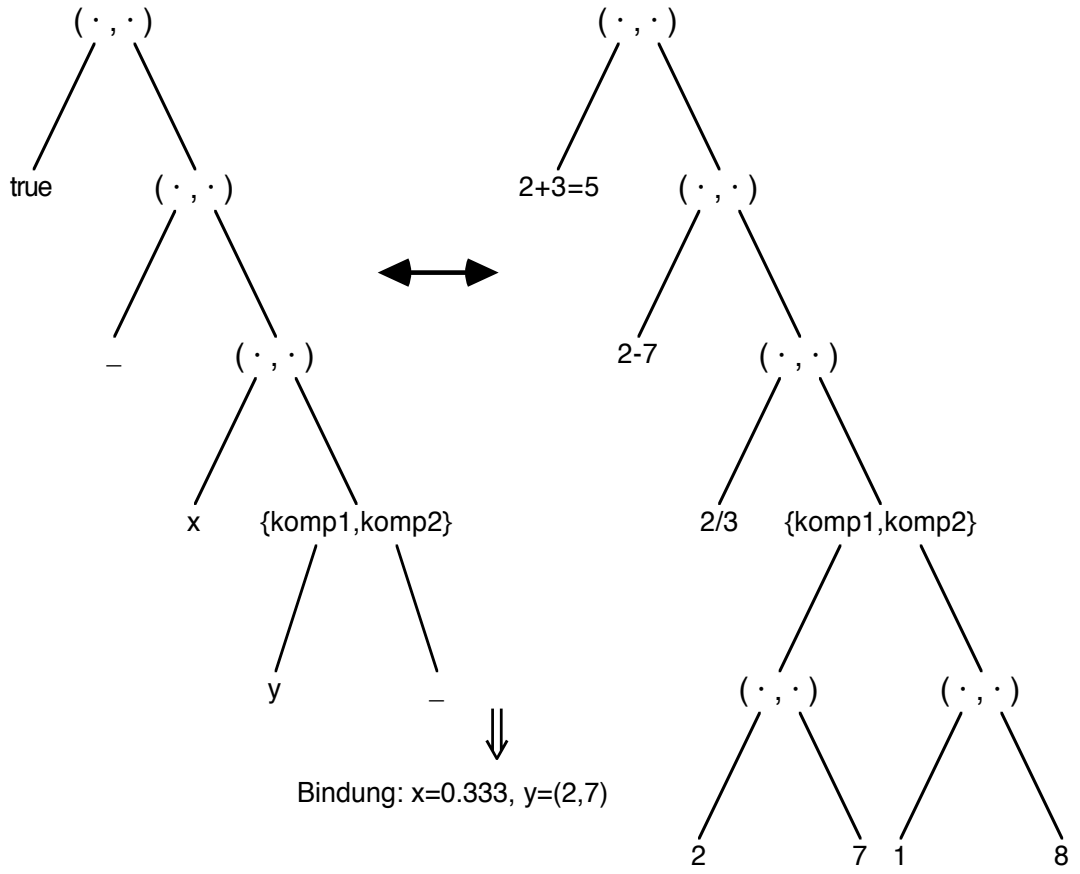


Abb. 3: Pattern-Matching

Zur übersichtlichen Steuerung unterschiedlicher Aktionen in Abhängigkeit von dem Muster, das einem Argument zugrundeliegt, bietet ML die Möglichkeit, mehrere Musteralternativen bei Funktionsdefinitionen anzugeben. Das System versucht dann die Alternativen der Reihe nach gegen das Argument zu matchen, bis eine Variante Erfolg hat. Die einzelnen Alternativen trennt man durch einen senkrechten Strich.

Beispiele:

1)
```
fun  not true = false |
     not false = true.
```

2) Beim Kinderspiel Papier, Schere, Stein gilt folgender Gewinnplan:

gewinnt gegen	Papier	Schere	Stein
Papier	X	-	+
Schere	+	X	-
Stein	-	+	X

In ML:
```
datatype Spiel = Papier | Schere | Stein;
fun  gewinnt (Papier,Stein) = true |
     gewinnt (Schere,Papier) = true |
     gewinnt (Stein,Schere) = true |
     gewinnt (_,_) = false.
```

Eine besondere Unterstützung bei der Vermeidung von Fehlern bietet ML dem Benutzer, der die Möglichkeit des Pattern-Matching intensiv nutzt: Das ML-System erkennt einerseits, wenn die für eine Funktion angegebenen Musteralternativen nicht alle möglichen Fälle abdekken. Andererseits meldet es auch redundante Alternativen, also Muster, die bereits durch die vorhergehenden vollständig erfaßt werden. Auf diese Weise werden bereits frühzeitig typische Programmierfehler erkannt.

3.4 Polymorphe Datentypen

Wie bereits in den vorigen Abschnitten mehrfach sporadisch erwähnt, erlaubt ML die Definition polymorpher Funktionen und Datentypen. Polymorphe Funktionen sind schon hinlänglich benutzt und erläutert worden. Hier wollen wir uns mit polymorphen Typen befassen. Diese Typen definiert man über Typausdrücke, die mithilfe folgender Elemente gebildet werden:

- Typvariablen (Hochkomma gefolgt von einem Bezeichner, z.B. 'a,'b, 'alphabet)
- Typkonstanten (int, real, ...)
- Typkonstruktoren (*, →)
- Typfunktionen (list (s. Abschnitt 3.3.6), selbstdefinierte Typen)
- Klammern, um Abweichungen von der Prioritätenregelung der vorstehenden Konstruktoren und Funktionen festzulegen, die wie folgt gegeben ist:

Typfunktionen	höchste Priorität
*	mittlere Priorität
→	niedrigste Priorität

Ein wichtiger Unterschied besteht in der Anwendung der Typfunktionen: Der Funktionsname steht hinter seinem Parameter, also z.B. (int*int) folge statt folge(int*int) für einen Typ Folge

mit Elementen aus Paaren des Typs int. Hier ist also int*int der aktuelle Parameter der Typfunktion list.

Beispiele:

1) Den polymorphen Typ aller Paare $\Delta\times\Delta$ über einem beliebigen Datentyp Δ definiert man durch

```
type 'param paare='param*'param.
```

Hier wird eine Typfunktion paare definiert mit dem Typparameter 'param und Wert 'param*'param. Konkretisierungen dieses Typs sind z.B.

```
type ip=int paare;
type bp=bool paare;
type ipaarvonpaar=(int paare) paare.
```

2) Der Typ

```
type ('a,'b,'c) T='a*'a list -> 'b*'c
```

beschreibt anschaulich den Typ aller Abbildungen f: $\Delta\times\Delta^*\rightarrow\Delta'\times\Delta''$. T ist also eine Typfunktion mit drei Parametern 'a, 'b, 'c und Ergebnis 'a*'a list ->'b*'c. Hierbei ist die Prioritätenregelung zu beachten, nach der

```
'a*'a list ->'b*'c = ('a*('a list)) -> ('b*'c)
```

gilt. Durch

```
type neu=(int,bool,real) T
```

konkretisieren wir den polymorphen Typ T zum Typ

```
int*int list -> bool*real
```

3.5 Generalisation

In ML schreibt man schematisch

```
datatype D=c1 of D1 | c2 of D2 | ... | cn of Dn
```

oder in parametrisierter Form

```
datatype (D1,...,Dn) D=c1 of D1 | c2 of D2 | ... | cn of Dn.
```

Die durch "|" getrennten Teile heißen *Varianten*. c_i, i=1,...,n, sind hier frei gewählte Bezeichner, die Konstruktoren von D. Sie repräsentieren Abbildungen

$c_i: D_i \rightarrow D.$

Man kann sie als Marken betrachten, um die Varianten voneinander zu unterscheiden. Sie besitzen die Funktion der Typdiskriminatoren in FUN.

Beispiel: Wir definieren eine Datenstruktur für eine Personaldatei:

```
type grunddaten={vorname: string, gehalt: real};
datatype mitarbeiter=mann of grunddaten*{bart: bool} |
                        frau of grunddaten*{gebname: string};
> datatype mitarbeiter = mann of grunddten*{bart:bool} |
```

```
frau of grunddaten*{gebname:string}
con mann = fn : (grunddaten*{bart:bool}) -> mitarbeiter
con frau = fn : (grunddaten*{gebname:string}) -> mitarbeiter
```

Hier wird der Datentyp mitarbeiter als disjunkte Vereinigung des Produkts zweier Records definiert. Die beiden Varianten werden über die Konstruktoren mann bzw. frau angesprochen:

```
val HerrMeier=mann({vorname="Paul", gehalt=3500.0},{bart=true});
> val HerrMeier = mann({gehalt=3500.0,vorname="Paul"},{bart=true}) : mitarbeiter
val FrauMeier=frau({gehalt=2500.0,vorname="Else"},{gebname="Kunze"});
> val FrauMeier = frau({gehalt=2500.0,vorname="Else"},{gebname="Kunze"}) :
                  mitarbeiter
```

3.6 Rekursive Datentypen

Einen rekursiven Datentyp D definiert man in ML mittels des Generalisationskonstruktors schematisch in der Form

```
datatype D=c0 of D' | c of D".
```

Hierbei bezeichnet D' den terminalen Datentyp mit dem Konstruktor c_0 und D" mit Konstruktor c einen Datentyp, in dessen Definition wiederum D vorkommt.

Beispiele:

1) Wir definieren den Datentyp nat (natürliche Zahlen):

```
datatype nat=null | succ of nat.
```

Ein Element $succ^n(null)$, $n \geq 0$, repräsentiert die natürliche Zahl n. Zur Konversion der Objekte $succ^n(null)$ in die lesbare Darstellung n und umgekehrt eignen sich die Funktionen

```
fun  conv_to_int null = 0 |
     conv_to_int (succ x) = 1+ (conv_to_int x);
fun  conv_to_nat 0 = null |
     conv_to_nat n = succ (conv_to_nat (n-1)).
```

Man beachte, daß conv_to_nat für $n<0$ nicht definiert ist.

2) Ein Text über dem Alphabet {a,b} ist entweder das leere Wort eps, oder er besteht aus einem einzelnen Zeichen a oder b, dem ein Text folgt. Wir definieren daher eine Linkssequenz

```
datatype zeichen= a | b;
datatype text=eps | conc of zeichen*text.
```

Elemente des Typs text sind also z.B.

```
eps, conc(a,eps), conc(a,conc(b,conc(a,eps)))
 ↓       ↓                  ↓
 ε       a                 aba
```

Verallgemeinern wir den Datentyp text durch Übergang zu einem Text über einem beliebigen Alphabet, so definieren wir den polymorphen Typ

```
datatype 'alphabet text=eps | conc of 'alphabet * 'alphabet text.
```

Den obigen Typ text über {a,b} erhalten wir dann auch durch

```
type abtext=zeichen text.
```

Typische Funktionen auf Texten sind z.B.:

Erstes Zeichen eines Textes: `fun first(conc(x,y))=x;`

Text ohne das erste Zeichen: `fun rest(conc(x,y))=y;`

Konkatenation zweier Texte:

```
fun append eps x = x |
    append (conc(x,y)) z = conc(x,append(y,z)).
```

Gleichheit auf Datentypen.

Auf jedem Datentyp, vorausgesetzt es handelt sich um einen Gleichheitstyp, ist in ML automatisch die Gleichheitsoperation vordefiniert. Diese stimmt häufig nicht mit der intendierten Gleichheit der Objekte des Typs überein.

Beispiel: Wir definieren den Typ der ganzen Zahlen int:

```
datatype int=null | succ of int | pred of int.
```

Hier ist nun +n standardmäßig durch $succ^n(null)$ und -n durch $pred^n(null)$ repräsentiert. Es gibt aber noch unendlich viele weitere Repräsentationen einer Zahl, z.B. für die Null:

null, succ(pred(null)), pred(succ(null)), $pred^n(succ^n(null))$, $n \geq 0$, usw.

Die vordefinierte Gleichheitsoperation unterscheidet jedoch die einzelnen Repräsentationen der Null, denn jeder Konstruktor erzeugt ein eindeutiges Objekt, das verschieden ist von allen Objekten, die auf eine andere Art erzeugt wurden; es gilt also

null≠succ(pred(null))≠pred(succ(null)) usw.

Die ML-Sicht des Datentyps int stimmt also nicht mit der wirklichen Struktur des Typs int überein.

Wenn jedes der durch einen Datentyp beschriebenen Objekte durch genau ein Datenobjekt dargestellt werden kann, dann spricht man von einer *freien* Repräsentation, anderenfalls von einer *unfreien*. Verwendet man für die Beschreibung von Objekten eine freie Repräsentation, so stimmt die vordefinierte Gleichheitsrelation mit der intendierten Gleichheit überein, anderenfalls muß man selbst eine neue Gleichheitsrelation definieren, die alle Datenobjekte gleichmacht, die das gleiche Objekt darstellen. Eine Menge von Objekten heißt *frei*, wenn es eine freie Repräsentation der Menge gibt.

3.6.1 Lineare Listen

Den Typ text aus Abschnitt 3.6 mit den angegebenen Operationen first, rest, append kann man auch als (polymorphe) Linkssequenz auffassen. Dieser Typ wird von ML standardmäßig unter der Typfunktion list zur Verfügung gestellt. Sind $d_1,...,d_n$ Objekte eines beliebigen Datentyps Δ, so faßt der Konstruktor [· ,..., ·] diese zu einer Liste

$[d_1,...,d_n]$: Δ list

zusammen. Die Konstanten

[ ] oder nil

repräsentieren die leere Liste.

Beispiele: [2,3]: int list

[(2,3),(4,5)]: (int*int) list

[sin,sqr,cos]: (real -> real) list

Zur Selektion von Listenelementen kann man die Standardfunktionen (s.u.) heranziehen, oder man verwendet die elegantere Methode des Pattern-Matching. Die hierfür wichtigste Operation auf Listen ist die Infix-Operation

:: : $\Delta \times \Delta^* \rightarrow \Delta^*$ mit

$a::b=[a,b_1,...,b_k]$, falls $b=[b_1,...,b_k]$, $k \geq 0$,

die aus einem Objekt a von Typ Δ und einer Liste b über Δ eine neue Liste generiert, in der a erstes Element vor den Elementen von b ist.

Mit der Konstanten [ ] und dieser Operation kann man Listen also als einen Datentyp auffassen, der wie folgt definiert ist:

```
datatype 'a list= nil | :: of 'a*'a list.
                  ↑   ↑
                  [ ]  infix-Konstruktor
```

Standardfunktionen auf Listen.

Test auf Leerheit:

```
fun  null [ ] = true |
     null (_::_) = false.
```

Erstes Element (head) einer Liste:

```
fun  hd (x::_) = x.
```

Liste ohne ihr erstes Element (tail):

```
fun  tl (_::x) = x.
```

Länge einer Liste:

```
fun  len [ ] = 0 |
     len (_::x) = 1+len(x).
```

Konkatenation zweier Listen:

```
fun  append [ ] x = x |
     append (x::y) z = x::(append y z).
```

Hier kann man auch die vordefinierte Infixoperation @ verwenden.

Elementtest:

```
fun  member [] x = false |
     member (z::y) x = x=z orelse member y x.
```

Standardfunktionale auf Listen.

Generation von Listen: Ausgehend von einem Anfangswert a wird mithilfe einer Funktion f eine Liste der Form

$$[a,f(a),f(f(a)),...,f^k(a)]$$

bis zu einem Index k gebildet:

```
fun  generate f a 0 = [a] |
     generate f a k = [a]@generate f (f a) (k-1).
```

Beispiel:

```
generate (fn x => x+1) 0 10;
> [0,1,2,3,4,5,6,7,8,9,10]: int list
```

Transformation einer Liste, d.h. Anwendung einer Funktion f auf alle Listenelemente:

```
fun  map f [ ] = [ ] |
     map f (x::y) = (f x)::(map f y).
```

Beispiele:

```
map sin [0.1,0.2,0.3,0.4,0.5];
> [0.0998,0.1987,0.2955,0.3894,0.4794]: real list
map (fn x => x+1) [1,2,3,4];
> [2,3,4,5]: int list
map (fn x => (x,2*x)) [1,2,3,4];
> [(1,2),(2,4),(3,6),(4,8)]: (int*int) list
```

Heraus*filtern* aller Elemente einer Liste, die ein bestimmtes Prädikat p erfüllen:

```
fun  filter p [ ] = [ ] |
     filter p (x::y) = if p x then x::(filter p y) else filter p y.
```

Beispiel:

```
filter (fn x => x>0) [-7,-3,8,-2,5,3,6,-1];
> [8,5,3,6]: int list
```

Existenzquantor. Existieren Elemente in einer Liste, die das Prädikat p erfüllen:

```
fun  exists p [ ] = false |
     exists p (x::y) = (p x) orelse (exists p b).
```

Beispiel:

```
exists (fn x => x<0) [-7,-3,8,-2,5,3,6,-1];
> true: bool
```

3.6.2 Bäume

Mit unserem Vorwissen aus Kapitel 9 der Vorlesung und über die Generalisation in ML ergibt sich aus der Definition von Bäumen unmittelbar eine Datentypdefinition in ML:

```
datatype bintree = empty | node of bintree*bintree.
```

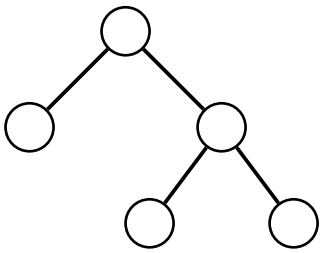
Blätter (äußere Knoten) in einem Baum sind dann Objekte der Form

```
node(empty,empty).
```

Die übrigen Knoten sind *innere* Knoten.

Beispiele:

```
node(empty,empty): bintree
```



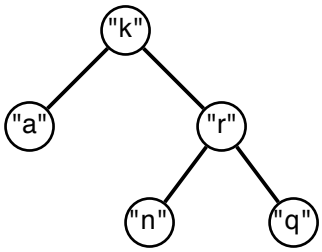
```
node(node(empty,empty),
     node(node(empty,empty),node(empty,empty))): bintree
```

Von größerer Bedeutung für die Praxis sind markierte binäre Bäume. Als Markierung lassen wir beliebige Datentypen zu, wir definieren daher einen polymorphen Datentyp

```
datatype 'label bintree = empty | node of 'label bintree*'label*'label bintree.
```

Beispiele:


```
node(empty,1,empty): int bintree
```




```
node(node(empty,"a",empty),"k",
     node(node(empty,"n",empty),"r",node(empty,"q",empty))):
          string bintree
```

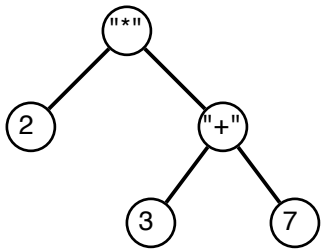
Eine weitere Verallgemeinerung bilden markierte binäre Bäume, bei denen die Blätter und die inneren Knoten Markierungen unterschiedlichen Datentyps besitzen:

```
datatype ('leaflabel,'label) bintree = leaf of 'leaflabel |
                node of ('leaflabel,'label) bintree*'label*('leaflabel,'label) bintree.
```

Ein typisches Anwendungsgebiet dieser Datenstruktur sind Bäume für arithmetische Ausdrükke.

Beispiele:

leaf(1): (int,'label) bintree



node(leaf(2),"*",node(leaf(3),"+", leaf(7))): (int,string) bintree