3 Einführung in ML

3.1 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 behandeln in dieser Vorlesung Standard ML (**SML**).

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.

3.2 Elementare Datentypen und Funktionen

3.2.1 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:

: _ | = => -> #

3.2.2 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 escape-Sequenz. Oben zeigt z. B. \" an, daß das Zeichen " hier als Teil des Strings und nicht als Begrenzer zu betrachten ist.

Wegen der strengen Typisierung von ML sind die Typen int und real disjunkt. Dies bedeutet, ein int-Objekt kann 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

x andalso y = y, falls x=false

false, sonst.

orelse: bool×bool→bool mit

x orelse y = true, falls x=true

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 andalso y) auch einen Wert liefert, wenn y nicht definiert ist.

3.2.3 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

if B then E else E' = E, falls B der Wert true hat,

E', sonst.

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.
Man spricht in diesem Zusammenhang auch von *Orthogonalität* der Programmiersprache, weil alle Sprachelemente in *beliebiger* Weise miteinander kombiniert werden können. In gängigen imperativen Programmiersprachen sind dieser Beliebigkeit meist enge Grenzen gesetzt: So sind z.B. Funktionen und manche strukturierte Datentypen meist Objekte zweiter Klasse (*2nd class citizens*), weil gewisse Manipulationen mit ihnen nicht zugelassen sind (z.B. Arrays als Funktionsergebnis in PASCAL). Diese Manipulationen darf man nur für die Objekte erster Klasse (hierzu gehören i.a. die elementaren Datentypen) ausführen.

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

$$(\Delta\rightarrow\Delta')\rightarrow((\Delta''\rightarrow\Delta)\rightarrow(\Delta''\rightarrow\Delta')).$$

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.
```

3.2.4 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

```
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
```

3.2.5 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 gecurried ist. Man hat in ML die Wahl, eine Funktion als vollständig oder teilweise gecurried zu definieren oder die Argumente zu einem Tupel zusammenzufassen (s. unten Beispiel 1).

```
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 gecurried. Probieren wir noch die ungecurriedte 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: Sind D, D' beliebige Typen, so ist $D*D'$ das kartesische Produkt von D und D', also die Menge aller Paare $(d,d') \in D \times D'$.

In Abschnitt 2.2.3 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 einen 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. Ein Zugriff auf einen Bezeichner wird bezgl. der Deklarationsstelle aufgelöst. Dies hat folgende Konsequenzen: Wird ein Bezeichner für einen anderen Zweck neu deklariert, so beziehen sich alle bisherigen Zugriffe auf diesen Bezeichner auf seinen ursprünglichen Wert. Die neue Bedeutung des Bezeichners kommt erst für die nachfolgenden Zugriffe zum Tragen. Hiervon zu unterscheiden ist die dynamische Bindung, bei der alle Bezeichner bezgl. ihrer Zugriffsstelle an Werte gebunden werden. Diese Bindungsart verwendet z.B. LISP.

In diesem Zusammenhang sind die Begriffe von gebundenen und freien Variablen (Bezeichnern) von Bedeutung. Ein Bezeichner innerhalb des Rumpfes einer Funktionsdeklaration heißt *gebunden*, wenn er ein formaler Parameter der Funktion ist, anderenfalls heißt er *frei*.

Beispiel: Wir definieren eine neue Funktion flaeche für die Oberfläche einer Kugel:

```
fun flaeche r=4.0*pi*r*r;
> val flaeche = fn: real -> real
```

Hier ist r eine gebundene, pi eine freie Variable.

Alle folgenden Anwendungen von flaeche beziehen sich jetzt auf diese Deklaration. Alle Zugriffe bis zu diesem Zeitpunkt beziehen sich weiterhin auf die ursprüngliche Definition von flaeche. So berechnet also volumen nach wie vor das Volumen eines Zylinders (Abb. 1).

Statische Bindung	**Dynamische Bindung**
val pi=3.1415;	val pi=3.1415;
fun flaeche r=pi*r*r;	fun flaeche r=pi*r*r;
fun volumen r h=(flaeche r)*h;	fun volumen r h=(flaeche r)*h;
volumen 17.0 3.0;	volumen 17.0 3.0;
fun volumen (r,h)=(flaeche r)*h;	fun volumen (r,h)=(flaeche r)*h;
volumen (17.0,3.0);	volumen (17.0,3.0);
val volumen12=volumen 12.0;	val volumen12=volumen 12.0;
fun flaeche r=4.0*pi*r*r;	fun flaeche r=4.0*pi*r*r;
volumen (17.0,3.0);	volumen (17.0,3.0);

Abb. 1: Vergleich von statischer und dynamischer Bindung

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
```