Kapitel 2. Problemnahe Programmierung am Beispiel der Funktionalen Programmierung: Standard ML

Dieses Kapitel gibt eine Einführung in die funktionale Programmiersprache SML. Nach einer informellen Vorstellung der wesentlichen Merkmale der funktionalen Programmierung werden die wichtigsten Konzepte dieses Programmierstils anhand von SML präsentiert: Grunddatentypen, Funktionen, Rekursion, Deklarationen, Produkt- und Listentypen, Pattern Matching, Funktionen höherer Ordnung.

2.1 Wesentliche Merkmale der funktionalen Programmierung

"Problemnahe Programmierung" nennt man einen Programmierstil, der sich soweit wie möglich an der Anwendung orientiert. Rechnerspezifische Konzepte sollen im Programm so wenig wie möglich sichtbar werden. Anstatt anzugeben, *wie* etwas berechnet wird, gibt man an, *was* berechnet werden soll. Um dies als Algorithmus verstehen zu können, muß man festlegen, wie ein Programm ausgeführt werden kann. In der funktionalen Programmierung ist das Grundkonzept der Begriff der *Funktion*.
Mathematisch definiert man eine Funktion folgendermaßen:

Definition (Funktion)
Eine (totale) *Funktion* f: A $\rightarrow$ B ist eine Teilmenge des kartesischen Produkts A$\times$B mit der Einschränkung:

$$\forall a \in A. \exists! \, b \in B. \, (a, b) \in f$$

$\forall$ bezeichnet den Allquantor ("für alle..."), $\exists!$ bezeichnet die *eindeutige* Existenz ("es gibt *genau* ein..."), A heißt Urbildmenge oder Definitionsbereich, B die Bildmenge oder Wertebereich von f. Da es genau ein b für jedes a gibt, schreibt man auch f(a) oder sogar fa für b. Die Teilmenge von B, die alle Bilder von f enthält, bezeichnet man mit f(A).
Die Eindeutigkeit des Bildes einer Funktion kann umformuliert werden zur *Substitutionsregel von Leibniz*:

$$\forall x, y \in A. \; x=y \Rightarrow f(x) = f(y)$$

Dies ist ein wichtiges Axiom der funktionalen Programmierung. Es schließt "Seiteneffekte" (wie in der imperativen Programmierung) aus.

Beispiele für Funktionen:

Kombinatorische Funktionen: ggt, Binomialkoeffizienten,
Trigonometrische Funktionen: sin, cos
Geometrische Funktionen: Volumen, Fläche

Eine vom Rechner diktierte Einschränkung ist unerläßlich, Problemlösungen müssen *kon-*

struktiv sein. D. h. es genügt nicht, die Existenz einer Lösung nachzuweisen, sondern die Lösung muß durch Ausführung eines Algorithmus in endlicher Zeit konstruiert werden können. Von diesem Algorithmus wird ein Minimum an Effizienz erwartet. Damit ist die Definition einer Funktion als Menge von Wertepaaren uninteressant:

(1) Unendliche Mengen können nicht in endlicher Zeit aufgezählt oder in einem endlichen Raum dargestellt werden.

(2) Die Suche nach einem bestimmten Funktionswert ist selbst in einer endlichen Menge zu aufwendig.

(3) Der Speicheraufwand bei einer Darstellung von Funktionen als Menge ist zu aufwendig.

Deshalb verwenden funktionale Programmiersprachen gleichungsbasierte Darstellungen, die einfach (von links nach rechts) ausgewertet werden können.

Beispiel (Fakultät)

```
fac(x) = if x = 0 then 1
         else x * f(x-1)
```

Auswertung von f(1)

```
f(1)  →  if 1 = 0 then 1
         else 1 * f(1-1)
      →  1 * f(1-1)
      →  1 * f(0)
      →  1 ∗ (if 0 = 0 then 1
           else 1 * f(0-1))
      →  1∗1
      →  1
```

Bekannt sind folgende funktionale Programmiersprachen (in historischer Reihenfolge):

1960 LISP
LISP wurde von John McCarthy entwickelt. LISP ist eine Abkürzung für "Listenprozessor". Listen bilden die grundlegende Datenstruktur.
Beispiel: ("+" 3 ("-" 7 2))
LISP ist eine ungetypte Sprache, die für Anwendungen in der KI entwickelt wurde und immer noch sehr beliebt ist. Bekannte LISP-Dialekte sind Scheme und CommonLISP.

1978 FP
FP wurde von John Backus in seiner Turing-Preis-Rede vorgestellt. In FP werden Programme aus Grundbausteinen durch Funktionskomposition und andere "combining forms" zusammengesetzt. Obwohl die Sprache nicht mehr eingesetzt wird, sind die Ideen noch relevant. Wie baut man wiederverwendbare Softwarebibliotheken?

1980 ML
ML wurde um 1980 von Robin Milner als Metasprache für seinen Theorembeweiser LCF ("Logic for Computable Functions") entwickelt und Ende der achtziger Jahre standardisiert("SML"). SML besitzt ein mächtiges (strenges, polymorphes, stabiles) Typsystem und ein richtungsweisendes Modulsystem (entwickelt von Dave MacQueen).

SML wird für viele Anwendungen eingesetzt. Ein Beispiel ist der Theorembeweiser Isabelle.

Neuere funktionale Programmiersprachen sind

1985 Miranda
Miranda wurde von John Turner entwickelt, es ist ähnlich zu ML, basiert aber auf dem Prinzip der verzögerten Auswertung.
Eine Weiterentwicklung davon ist Haskell.

2.2 Programmaufbau von Standard ML

Ein ML Programm besteht aus einer Serie von durch Strichkommas getrennten **Deklarationen**.

- Wertdeklaration: `val a = 5.0–3.0;`
  Bindet einen Namen an einen Wert.
- Funktionsdeklaration: `fun half x = x/2.0;`

  Bindet einen Namen an eine Funktionsvorschrift. Eine Funktionsdeklaration kann auch als Wertdeklaration geschrieben werden:
  `val half = fn x => x/2.0;`
- Term oder Ausdruck: `half a;`

  Wird unter Konsultation der vorherigen Bindungen ausgewertet. Ein Ausdruck ist eine implizite Wertdeklaration mit dem Namen `it`:
  `val it = half a;`

Damit ist jede ML Anweisung als Wertdeklaration darstellbar.
Ein ML Programm stellt eine sich verändernde Sammlung von Bindungen dar, die **Umgebung (environment)** genannt wird. Elemente einer Umgebung sind Name/Wert–Paare.

	ML Direktiven	**Umgebung**
	(Nach Laden von SML)	`<<it,()>>`
–	`val a = 5.0–3.0;`	`<<it,()>,<a,2.0>>`
>	`val a = 2.0 : real;`	
–	`fun half x = x/2.0;`	`<<it,()>,<a,2.0>>,`
>	`val half = fn: real –> real`	`<half,fn x => x/2.0>>`
–	`half a;`	`<<it,1.0>,<a,2.0>,`
>	`val it = 1.0 : real`	`<half,fn x => x/2.0>>`

Eine ML Anweisung kann sich über mehrere Zeilen erstrecken, und mehrere Anweisungen können eine Zeile bilden.

Der grundlegende Baustein eines ML Programms (und einer funktionalen Sprache im allgemeinen) ist der **Ausdruck (expression)**. Eine ML Deklaration gibt einem Ausdruck einen Namen. Der **Wert (value)** eines Ausdrucks wird durch seine Umgebung bestimmt und vom ML System wiedergegeben. ML fügt den **Typ (type)** des Wertes (die Bezeichnung der Menge, zu der der Wert gehört) automatisch selbst hinzu (Typinferenz).

Beispiele:

```
–   3;
>   3 : int                    (ganze Zahl)
–   true;
>   true : bool                (Wahrheitswert)
–   3.0;
>   3.0 : real                 (Gleitpunktzahl)
–   "hello"
>   "hello" : string           (Zeichenfolge)
```

Die Variable `it` enthält jeweils den Wert des letzten Ausdrucks, dem kein Name gegeben wurde.

Beispiele:

```
–   [1,2,3];
>   [1,2,3] : int list
–   it;
>   [1,2,3] : int list
–   fn x => x/2.0;
>   fn: real -> real           (Funktion)
–   it 3.0;
>   1.5 : real
```

Basistypen:

`bool` (Wahrheitswerte), `int` (ganze Zahl), `real` (Gleitpunktzahl), `string` (Buchstabenfolgen; Characters sind einelementige Strings) und `unit`.

Typkonstruktoren:

– > (Funktion), `*` (kartesisches Produkt), `list`

2.3 Namen und Deklarationen

Definition: (Name)

Ein **(alphanumerischer) Name** (Identifikator) ist eine Folge von Zeichen (characters), die mit einem Buchstaben (a,...,z,A,...,Z) beginnt und sonst neben Buchstaben Ziffern, Hochkommas (') und Underscores (_) enthalten darf.

Ein **symbolischer Name** ist eine beliebige Sequenz über dem Zeichensatz:

```
! $ # % & * + – / : < = > ? @ ´ ^ ` | ~
```

Beachte: Alphanumerische und symbolische Namen bedienen sich getrennter Zeichensätze und sind deshalb auch ohne Trennzeichen isolierbar. Gewisse Zeichensequenzen sind "reserviert" (reserved words) und können nicht als Identifikatoren benutzt werden:

```
abstype and andalso as case do datatype else end exception fn
    fun handle if in infix infixr let local nonfix of op orelse
    raise rec then type val while
( ) [ ] { } , : ; ... | = => -> _ #
```

Beispiele:

```
val Zwei = 2;
    zwei ?
    zwei_zwei
    zwei2zwei
    val ! = 1 ?
```

2.3.1 Wertdeklaration

Werte und Ausdrücke können durch Namen bezeichnet werden.

Syntax:

`val` <Name> = <Ausdruck>`;`

Das System gibt den Namen, den Wert des **Ausdrucks** und den **Typ** zurück.
Deklarierte Namen können in Ausdrücken verwendet werden.

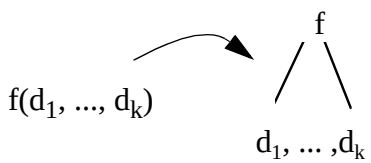
2.3.2 Auswertung

Jeder Ausdruck wird zu seinem Wert/Resultat ausgewertet.
z.B. 5-3 wird ausgewertet zu 2
half 2.0 wird ausgewertet zu 1.0

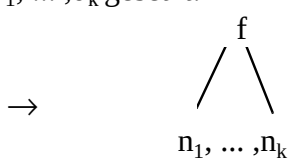
Beispiel:
~1, 0, 1, 2, 3, ..n sind Normalformen für ganze Zahlen.
0.1, ~1.7, ... sind Normalformen für Gleitpunktzahlen.

Die Auswertung eines Ausdrucks $f(d_1, ..., d_k)$, wobei f ein Funktionssymbol und $d_1, ..., d_k$ Ausdrücke sind, erfolgt in 3 Schritten.

1) Man transformiert den Ausdruck in einen Baum

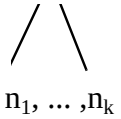


2) Die Ausdrücke $d_1, ... ,d_k$ werden zu Normalformen $v_1, ..., v_k$ ausgewertet und um die Stelle von $d_1, ... ,d_k$ gesetzt.



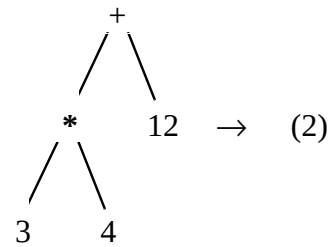
Ist einer der d_i ein Identifikator, so wird der Wert aus der Umgebung geholt.

3) Der Ausdruck f wird zu einem Wert v ausgewertet.



Beispiel:

(1) (3*4) + 12 → (1)



→ (2)

→ (3)

24

Definition: (Bindung; Umgebung)

(1) Ein Paar <Name, Wert> heißt **Bindung** (**binding**).

(2) Eine Liste von Bindungen

$<<x_1, v_1>,\ldots,<x_n, v_n>>$

heißt **Umgebung (environment)**.

Beispiele:

(1)
```
– val dutzend = 3*4;
  > val dutzend = 12 : int
  – 5 * dutzend + 30;
  > 90 : int
```

Zur Ausführung einer Wertdeklaration

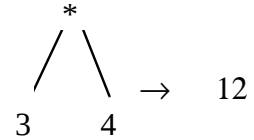
val <Name> = <Ausdruck>

wird zuerst der Wert v von <Ausdruck> bestimmt und dann die Bindung < <Name>, v > erzeugt.

Beispiel:

val dutzend = 3*4

1) Auswertung von 3*4

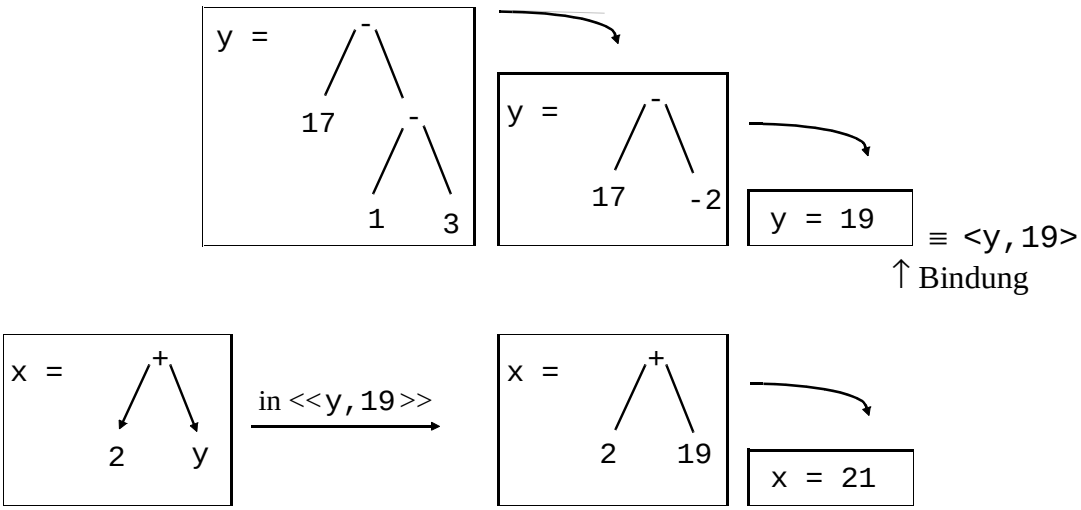


2) Erzeugung der Bindung
<dutzend, 12>

(2) Es ergibt sich eine Rechenersparnis, wenn ein Name mehrmals verwendet werden kann.

```
– val x = 13*(7+3);
> val x = 130 : int
– val y = (5*x) – (13+x);
> val y = 507 : int
```

(3)
```
– val y = 17 –(1–3);
> val y = 19
– val x = 2+y;
> val x = 21
```



Umgebung: <<y,19>,<x,21>>

Anmerkungen:
Der Wert eines Ausdrucks ist durch seine Umgebung bestimmt.

Alle Namen in einem Ausdruck müssen gebunden sein.

(Gegen–)Beispiel:
```
– val x = 17;
> val x = 17
– x*y;
> unbound identifier: y
```

2.3.3 Statische Bindung

Bei der Auswertung eines Ausdrucks gilt derjenige Wert eines Namens, der bei der **Auf-**

schreibung gültig war (nicht der Wert zur Zeit der Auswertung, falls der Name einen neuen Wert erhalten hat – das wäre eine "dynamische" Bindung).

Beispiel:

```
–   val x = 17;

>   val x = 17 : int
–   val y = x+x;
>   val y = 34 : int
–   val x = 13 : int;
>   val x = 13 : int
–   y;
>   34 : int        !
–   x+x;
>   26 : int
```

Man spricht von **Gültigkeitsbereich (scope)** eines Namens. Auf der Ebene, auf der man mit dem ML System interagiert, ist der Bindungsbereich eines Namens das **ganze restliche Programm** (d.h. die verbleibende Dauer, für die das ML System geladen ist), es sei denn, es erfolgt eine **Neudeklaration** des Namens. Dann endet der Gültigkeitsbereich nach Auswertung der Deklaration.

Beispiel:

```
- val x = 13;
> val x = 13: int
- val x = x+7;
> val x = 20: int
```

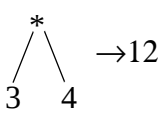
erster Bindungsbereich von x

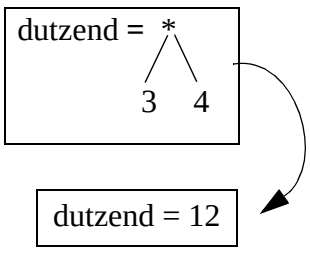
zweiter Bindungsbereich von x

Zuerst ist
x an 13 gebunden.
Dann ist
x an 20 gebunden.

Merke: Neudeklaration ist ein nicht–funktionales Konzept !!!

Beispiel:
(1) (1) val dutzend = 3*4:





(2) Bindung <dutzend, 12>

(2) 5*dutzend + 30: