3.1.4 Weitere Merkmale von FP

Hohe Abstraktion des physischen Modells

- keine explizite Verwaltung von Speicherzellen
  - ▷ Variablen sind keine Speicherzellen sondern Namen für einen Wert

  $\Longrightarrow$ keine explizite Anforderung/Freigabe des benötigten Speichers $\Longrightarrow$ Garbage-Collection
- keine Strukturen zur expliziten Kontrolle des Kontrollflußes
  - ▷ keine Schleifen $\Longrightarrow$ Iteration durch Rekursion ersetzt

```
fun fact n = if n<=0 then 1
             else n*fact(n-1)
```

Typ-Inferenz

Typen müssen (meist) nicht explizit spezifiert werden; sie werden automatisch vom Compiler inferiert

- Deklaration:

```
fun comp (f,g) = fn x => f(g(x))
```

- Compiler antwortet mit dem inferierten Typ:

$$val\ comp = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b$$

3.1.5 Typische Eigenschaften der FP-Sprachen

- Funktionen sind Werte erster Klasse
- Gute Unterstützung durch das Typ-System (Polymorphie, Typ-Inferenz)
- (Möglichst) keine Seiteneffekte
- Hoher Abstraktionsgrad (Programme sind eher für das Problem als für die Maschine repräsentativ)
- Fall-Unterscheidung für Funktionen durch Muster-Angabe (*pattern matching*)
- Dekomposition eines Wertes durch Pattern-matching
- Automatische Speicherfreigabe (*garbage collection*)

4 Funktionale Programmierung in SML

- ML, 1973 Robin Milner (Meta-Language: die Spezifikationssprache eines Theorem-Beweis-Programms, 1973 - Robin Milner)
- SML, 1983 Robin Milner: Versuch, die verschiedenen Dialekte von ML zu standardisieren (Standard: SML'97)
- SML-Compiler: SML/NJ, MoscowML, Poly/ML, MLton, SML.NET
- SML/NJ = Standard-Implementierung
- Verwandte Sprachen: Caml, OCaml

Struktur eines Programms

Programmspezifikation: Menge von Wert-Definitionen.

Programmausführung: Auswertung eines Wertes.

4.1 Die Interpreter-Umgebung

Die SMl/NJ Interpreter-Umgebung wird mit `sml` aufgerufen...

```
~/>sml
Standard ML of New Jersey, Version 110.0.7
-
```

Definitionen von Variablen, Funktionen u.s.w können direkt eingegeben werden.

Alternativ kann man sie aus einer Datei einlesen:

```
- use "test.sml";
[opening test.sml]
val it = () : unit
```

Die Interpreter Umgebung - Ausdrücksauswertung

```
- 1+2;
val it = 3 : int
- 1+
= 2;
val it = 3 : int
```

- Bei `-` wartet der Interpreter auf Eingabe.
- Bei unvollständiger Eingabe bittet `=` um weitere Eingabe.
- Das `;` bewirkt Auswertung der bisherigen Eingabe.
- Das Ergebnis wird berechnet und mit seinem Typ ausgegeben.

Vorteil: Das Testen von einzelnen Funktionen kann stattfinden, ohne jedesmal neu (alles) zu übersetzen ($\mapsto$ inkrementelle Übersetzung)

4.2 Vordefinierte Datentypen

- Eine funktionale Programmiersprache arbeitet mit einer Menge von Typen
- Ein Typ ist eine Menge von Werten (Konstanten)
- Typen werden als Definitions- und Bildbereich für Funktionen (Operatoren) benutzt

Vordefinierte Basis-Typen in SML

Typ	Konstanten (Beispiele)	Operatoren
int	`0 3 ~7`	`+ - * div mod` : int × int ↦ int `~` : int ↦ int
real	`3.0 7.0`	`+ - * /` : real × real ↦ real `~` : real ↦ real
bool	`true false`	`not` : bool ↦ bool `orelse andalso` : bool × bool ↦ bool
string	`"hallo"`	`^` : string × string ↦ string
char	`#"a" #"b"`	
unit	`()`	

Typen werden vom Compiler (meist) automatisch erkannt:

```
- 1;
val it = 1 : int

- 1.0;
val it = 1.0 : real

- ~3.0/4.0;
val it =  0.75 : real

- "So" ^ " " ^ "geht" ^ " " ^ "das";
val it = "So geht das" : string

- 1>2 orelse not (2.0<1.0);
val it = true : bool
```

4.3 Variablen

- In FP ist eine Variable ein Bezeichner für einen Wert (nicht für eine Speicherzelle wie bei der imperativen Programmierung)
- Die Variable behält dann für immer diesen Wert.
- Eine Variable wird mit `val` deklariert und belegt:

```
- val x = 1;
val x = 1 : int
- val y ="Eine Zeichenkette";
val y = "Eine Zeichenkette" : string
- val z = x+1;
val z = 2 : int
```

Variablen

- Eine erneute Definition für `x` weist nicht `x` einen neuen Wert zu, sondern erzeugt eine neue Variable mit Namen `x`. Dadurch ist die alte Definition nicht mehr sichtbar.

```
- val x = 1;
val x = 1 : int
- val y ="Eine Zeichenkette";
val y = "Eine Zeichenkette" : string
- val z = x+1;
val z = 2 : int
- val x = 20;
val x = 20 : int
- val t = x+1;
val t = 21 : int
```

4.4 Funktionen

4.4.1 Funktionen definieren (erste Methode)

- Ein Funktionswert wird mit Hilfe des Schlüsselsworts `fn` definiert
- Bsp.: `fn x => x` ist die Identitätsfunktion
- Variablen können Funktionswerte bezeichnen:

```
val identity = fn x => x;
val identity = fn : 'a -> 'a
val increment = fn x => x+1;
val increment = fn : int -> int
```

4.4.2 Funktionsanwendung

Wenn `f` ein Funktionswert und `x` ein Wert aus dem Definitionsbereich von `f` ist, dann ist `f x` (keine Klammern nötig) die Anwendung von `f` auf `x`, also der Wert der Funktion `f` an Stelle `x`.

```
val identity = fn x => x;
val identity = fn : 'a -> 'a
val increment = fn x => x+1;
val increment = fn : int -> int
identity 2;
val it = 2 : int
increment 3;
val it = 4 : int
increment (increment 3);
val it = 5 : int
```

4.4.3 Funktionen definieren (zweite Methode)

Für die Definition von Funktionswerte gibt es auch eine verkürzte Syntax. Statt:

```
val identity = fn x => x;
val identity = fn : 'a -> 'a
val increment = fn x => x+1;
val increment = fn : int -> int
```

kann man schreiben:

```
fun identity x = x;
val identity = fn : 'a -> 'a
fun increment x = x+1;
val increment = fn : int -> int
```

Rekursive Funktionen

- Ein Wert wie `fn x => x` ist eine namenlose (anonyme) Funktion
  $\Longrightarrow$ keine Definition rekursiver Funktionen möglich
- Nicht namenlose Funktionen erlauben die Definition von rekursiven Funktionen:

```
fun fact n = if n<=0 then 1 else n*(fact (n-1))
val fact = fn : int -> int
fact 3;
val it = 6 : int
```

4.5 Definition neuer Typen

Sei MT die Menge der Typen in der Sprache.

Neue Typen können mit Hilfe von Operatoren konstruiert werden (ähnlich wie Ausdrücke aufgebaut werden), die die Form haben:

$$op : MT^n \mapsto MT \text{ mit } n \geq 0$$

Solche Operatoren heißen Typ-Operatoren.

Die vordefinierten Basis-Typen (`real`,`int`,`bool`,`unit`) sind nullstellige Typ-Operatoren (Typ-Konstanten).

4.5.1 Produkt-Typen

Der Typ-Operator $*$

$$* : MT^n \mapsto MT \text{ mit } n \geq 2$$

$$\alpha_1 * \alpha_2 * \cdots * \alpha_n = \{(v_1, v_2, \ldots, v_n) \mid v_k \in \alpha_k \text{ für alle } k\}$$

Beispiele:

- $int * int = \{(x, y) \mid x, y \in int\}$
  $(1, 2) \in int * int$
- der Typ-Operator $*$ steht zwischen Operanden (*infix operator*)

Produkt-Typ: Beispiele

```
- (1,2);
val it = (1,2) : int * int

- (1,2,true);
val it = (1,2,true) : int * int * bool

- (1,(2,true));
val it = (1,(2,true)) : int * (int * bool)

- ((1,2),true);
val it = ((1,2),true) : (int * int) * bool
```

Produkt-Typ: Beispiele

```
- (1,2,true) = (1,(2,true));
stdIn:42.1-42.26 Error: operator and operand don't agree [tycon mismatch]
 operator domain: (int * int * bool) * (int * int * bool)
 operand: (int * int * bool) * (int * (int * bool))
 in expression:
 (1,2,true) = (1,(2,true))

- (1,2,true) = (1,2,true);
val it = true : bool
```

4.5.2 Records-Typen (Verbunde)

Ein Record-Typ besteht aus Tupeln mit benannten Komponenten:

```
- val p1 ={vorName="John",name="Smith",alter="23"};
val p1 = {alter="23",name="Smith",vorName="John"}
 : {alter:string, name:string, vorName:string}

- val p2 = {vorName="Jan",name="Smith",alter="23"};
val p2 = {alter="23",name="Smith",vorName="Jan"}
 : {alter:string, name:string, vorName:string}

- val p3 = {vorName="Jan",name="Smith"};
val p3 = {name="Smith",vorName="Jan"}
 : {name:string, vorName:string}
```

Records

- Zwei Record-Typen sind gleich wenn sie gleich viele Komponente haben, jeweils mit dem selben Namen und dem selben Typ:

```
- p2 = p3;
stdIn:41.1-41.8 Error: operator and operand don't agree [tycon mismatch]
  operator domain: {alter:string, name:string, vorName:string}
                 * {alter:string, name:string, vorName:string}
  operand: {alter:string, name:string, vorName:string}
                 * {name:string, vorName:string}
  in expression:
    p2 = p3
```

Records

- Zwei Record-Werte (≡ Records) sind gleich, wenn sie vom selben Typ sind, und die jeweiligen Komponenten gleich sind

```
- val p1 ={vorName="John",name="Smith",alter="23"};
val p1 = {alter="23",name="Smith",vorName="John"}
 : {alter:string, name:string, vorName:string}

- val p2 = {vorName="Jan",name="Smith",alter="23"};
val p2 = {alter="23",name="Smith",vorName="Jan"}
 : {alter:string, name:string, vorName:string}

- p1 = p2;
val it = false : bool
```

Records

- Reihenfolge ist irrelevant

```
- {name="Schwarz",vorName="Peter",alter="25"} =
  {name="Schwarz",alter="25",vorName="Peter"};
val it = true : bool
```

- Tupel sind eine Spezialschreibweise für Records

```
- {1=true,2="Martin"};
val it = (true,Martin") : bool * string
- {1=3,2=5};
val it = (3,5) : int * int
```

4.5.3 Summen-Typen

- Da ein Typ eine Menge von Werten ist, kann man ihn angeben, indem man spezifiziert, wie die Werte konstruiert werden.
- Werte eines Typs werden mit Hilfe von (Typ-)Konstruktoren aufgebaut.
- Bei endlichen Typen kann man alle Elemente aufzählen
  $\longrightarrow$ Aufzählungstypen.

Aufzählungstypen (*enumeration types*)

- werden durch das Schlüsselwort `datatype` definiert.
- bestehen aus Konstanten (nullstellige Konstruktoren) mit symbolischen Namen getrennt durch "`|`".

```
- datatype Farbe = Karo | Herz | Pik | Kreuz
= datatype Wert = Neun | Zehn | Bube | Dame | Koenig | As;
datatype Farbe = Herz | Karo | Kreuz | Pik
datatype Wert = As | Bube | Dame | Koenig | Neun | Zehn
```

Aufzählungstypen (*enumeration types*)

- Der Compiler erkennt den Typ eines Wertes anhand des Konstruktors:

```
- datatype Farbe = Karo | Herz | Pik | Kreuz
= datatype Wert = Neun | Zehn | Bube | Dame | Koenig | As;
datatype Farbe = Herz | Karo | Kreuz | Pik
datatype Wert = As | Bube | Dame | Koenig | Neun | Zehn
- Kreuz;
val Kreuz : Farbe
- val pik_bube = (Pik,Bube);
val pik_bube = (Pik,Bube) : Farbe * Wert
```

Aufzählungstypen vs. ad-hoc Kodierungen

Mögliche Kodierung: Benutze Paare von Strings und Zahlen, z.B.

`("Karo","10")`	$\equiv$	Karo Zehn
`("Kreuz","12")`	$\equiv$	Kreuz Bube
`("Pik","1")`	$\equiv$	Pik As

Nachteile:

- Beim Test auf eine Farbe muß immer ein String-Vergleich stattfinden $\longrightarrow$ ineffizient!
- Darstellung des Buben als `12` ist nicht intuitiv $\longrightarrow$ unleserliches Programm!
- Welche Karte repräsentiert das Paar `("Kaor","1")`?
  (Tippfehler werden vom Compiler nicht bemerkt)

Besser als ad-hoc Kodierungen: Aufzählungstypen

```
- datatype Farbe = Karo | Herz | Pik | Kreuz
= datatype Wert = Neun | Zehn | Bube | Dame | Koenig | As;
datatype Farbe = Herz | Karo | Kreuz | Pik
datatype Wert = As | Bube | Dame | Koenig | Neun | Zehn
```

Vorteile:

- Darstellung ist intuitiv.
- Tippfehler werden erkannt:

```
- (Kaor , As );
stdIn:29.2-29.6 Error: unbound variable or constructor: Kaor
```

- Interne Repräsentation ist effizient.

Aufzählungstypen vs. Basis-Typen

Manche Basis-Typen können als spezielle vordefinierte Aufzählungstypen aufgefasst werden:

- `datatype bool = true | false`
- `datatype char = #"a" | #"b" | #"c" | ...`
- `datatype int = ... | ~2 | ~1 | 0 | 1 | 2 | ...`

Verallgemeinerung: Summentypen

- Im Allgemeinen können nicht alle Werte eines Typen aufgezählt werden.
- Stattdessen konstruiert man neue Werte aus Werten eines anderen Typen $\mapsto$ einstellige Konstruktoren.
- Ein neuer Typ wird definiert indem man alle seine (nullstelligen und einstelligen) Konstruktoren spezifiziert $\mapsto$ Summentypen
- werden mit Hilfe von `datatype` eingeführt

  `datatype` = $Konstruktor_1$ | $Konstruktor_1$ | ... | $Konstruktor_n$

Summentypen

Ein Konstruktor c eines Typen $T \in MT$ kann als Funtion aufgefasst werden:

- nullstellig: $c : \bullet \mapsto T$
- einstellig: $c : T_1 \mapsto T$ mit $T_1 \in MT$
- Beispiel:

```
datatype Farbe = Rot | Blau | RGB of (int*int*int)
```

$\texttt{Farbe} = \{\texttt{Rot}\} \cup \{\texttt{Blau}\} \cup \{\texttt{RGB (x,y,z)} \mid x, y, z \in \texttt{int}\}$

Summentypen: Beispiel

```
datatype Farbe = Rot | Blau | RGB of (int*int*int)
```

- Der Compiler erkennt den Typ eines Wertes anhand des Konstruktors:

```
- Rot;
val it = Rot : Farbe
- RGB (80,200,130);
val it = RGB (80,200,130) : Farbe
```

4.5.4 Pattern-Matching

Werte eines selben Typs können unterschiedlich behandelt werden, je nachdem mit welchem Konstruktor sie erzeugt wurden $\Longrightarrow$
Fall-Unterscheidung (*pattern matching*)

Die Fall-Unterscheidung erfolgt mit Hilfe des case-Ausdruckes:

```
case Ausdruck of
    Muster_1 => Ausdruck_1
  | Muster_2 => Ausdruck_2
        ... =>
  | Muster_n => Ausdruck_n
```

Pattern-Matching

`case` $Ausdruck$ `of` $Muster_1$ $=>$ $Ausdruck_1$

$\ldots$

$|Muster_n$ $=>$ $Ausdruck_n$

- $Muster_1$, $Muster_2 \ldots$, $Muster_n$ spezifizieren die zutreffende Struktur via Konstruktoren und Variablen
- $Ausdruck_1$, $Ausdruck_2 \ldots$, $Ausdruck_n$ müssen alle den selben Typ haben; das ist der Typ des gesamten case-Ausdrucks
- Wenn $Muster_i$ das erste zutreffende Muster (*pattern*) für den Wert von $Ausdruck$ ist, ist der Wert des case-Ausdrucks gleich der Wert von $Ausdruck_i$.

Beispiel:

```
datatype Farbe = Rot | Blau | RGB of (int*int*int)
```

```
- val f = RGB (80,200,130);
val f = RGB (80,200,130) : Farbe
- val description = case f of
                      Rot => "pure red"
                    | Blau => "pure black"
                    | RGB(80,200,130) => "Possibly Jamaica"

val description = "Possibly Jamaica" : string
```

Der If-Ausdruck

Oft findet die Fallunterscheidung über einen Wert vom Typ `bool`:

```
val fun max (x,y) = case x>= y of true => x
                                | false => y;
val max = fn : int * int -> int

max (3,2);
val it = 3 : int
```

Dafür gibt es eine alternative, kürzere Syntax:

```
val fun max (x,y) = if x>= y then x
                    else y;
val max = fn : int * int -> int
```

Der If-Ausdruck

Im Allgemeinen:

`case` *Expr* `of` `true` `=>` $Expr_1$ `|` `false` `=>` $Expr_2$	$\equiv$	`if` *Expr* `then` $Expr_1$ `else` $Expr_2$

If ist ein Ausdruck

```
fun handlePlural number what =
  what ^ (if number>1 then "s" else "")
val handlePlural = fn : int -> string -> string
- handlePlural 5 "student";
val it = "students" : string
```

In imperativen Sprachen wird zwischen Anweisungen und Ausdrücken unterschieden.

In funktionallen Sprachen sind alle Konstrukte Ausdrücke.

Der If-Ausdruck

```
if Expr then Expr1
else Expr2
```

$Expr_1$ und $Expr_2$ müssen den selben Typ haben.

```
fun inverse x =
   if x>0.0001 then 1.0/x else "a to large number";
stdIn:43.17-43.52 Error: types of rules don't agree [tycon mismatch]
 earlier rule(s): bool -> real
 this rule: bool -> string
 in rule:
 false => "error"
```

Pattern-Matching mit Variablen-Bindung

- Patterns können Variablen enthalten
- Beim Pattern-Matching über einen Wert werden die Variablen automatisch zu den entsprechenden Teilen des Wertes gebunden
- Die im Muster $Muster_i$ gebundenen Variablen sind im Ausdruck $Ausdruck_i$ sichtbar

```
- val f = RGB (80,200,130);
val f = RGB (80,200,130) : Farbe
- val description = case f of Rot => "pure red"
                            | Blau => "pure blue"
                            | RGB(x,y,z) =>
                      if (x=y) andalso (y=z) then "white"
                      else "something else";
val description = "something else": string
```

Pattern-Matching mit Variablen-Bindung

- Wenn man eine Variablen-Bindung nicht braucht kann man _ (Unterstrich) benutzten

```
- val f = RGB (80,200,130);
val f = RGB (80,200,130) : Farbe
- val description = case f of
                      Rot => "pure red"
                    | Blau => "pure blue"
                    | RGB(x,y,_) =>
                      if (x=y) then "kind of yellow"
                      else "something else";
val description = "something else" : string
```

Pattern-Matching mit Variablen-Bindung

- ist ein wichtiges Feature, die die Dekomposition eines Wertes in seine Teile unterstützt
- Zusätzlich überprüft der Compiler automatisch, ob die Patterns redundant oder unvollständig sind...

Unvollständige Patterns

```
- val description = case f of Rot => "pure red"
                  | Blau => "pure blue"
                  | RGB(80,200,130) => "kind of green" ;
stdIn:78.13-108.57 Warning: match nonexhaustive
 Rot => ...
 Blau=> ...
 RGB (80,200,130) => ...
val description = "kind of green": string
```

Alle Fälle sollten behandelt werden, evt. ähnlich wie unten:

```
- val description = case f of Rot => "pure red"
                  | Blau => "pure blue"
                  | RGB(80,200,130) => "kind of green"
                  | _ => "don't know";
val description = "kind of green": string
```

Redundante Patterns

```
-   val description = case f of
                        Rot => "pure red"
                      | Blau => "pure blue"
                      | RGB(x,y,z) => "RGB colour"
                      | RGB(80,200,130) => "kind of green";
stdIn:125.8-139.57 Error: match redundant
 Rot => ...
 Blau => ...
 RGB (x,y,z) => ...
 --> RGB (80,200,130) => ...
```

Redundante Patterns

Achtung: Die Reihenfolge ist wichtig

```
- val description = case f of
                      Rot => "pure red"
                    | Blau => "pure blue"
                    | RGB(80,200,130) => "kind of green";
                    | RGB(x,y,z) => "RGB colour"
val description = "kind of green": string
```