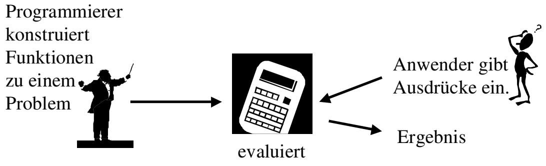
Funktionale Programmierung

- Funktionen, Ausdrücke, Typen -



Sitzungen und Skripte

Bsp. Sitzung in ML

```
- 45;
val it=45 : int

- 9*5;
val it=45 : int

- use("datei.ml");
[opening datei.ml]
val datei = fn : int -> int
val quadrat = fn : int -> int
val min = fn : int -> int -> int
val it = () : unit

- quadrat(min 3 4);
val it=9 : int
```

Bsp. „Externes“ Skript in “datei.ml“

```
fun quadrat x:int = x*x;
fun min x y:int = if x>y then y else x;
```

```
- val seite = 12;
- val flaeche = quadrat seite;
- flaeche;
144
```


Referenzielle Transparenz

1) **Extensionalitätsprinzip**:
   Das einzig wichtige Merkmal eines Ausdrucks ist sein Wert. Es ist völlig unwichtig, wie dieser Wert berechnet wird.
2) **Leibnizsches Prinzip der Ersetzung von Identitäten:**
   Der Wert eines Ausdrucks ändert sich nicht, wenn man irgendeinen seiner Teilausdrücke durch irgendeinen anderen gleichen Wert ersetzt.
3) **Prinzip der Definitheit:**
   Der Wert eines Ausdrucks ist innerhalb eines gewissen Kontexts (Gültigkeitsbereich) immer gleich, unabhängig von der Stelle, an der der Ausdruck im Kontext auftritt.

Ausdrücke und Werte

```
quadrat (3+4);  => quadrat 7    (+)
                => 7 * 7        (quadrat)
                => 49           (*)
```

Reduktion

oder

kanonische Repräsentation eines Wertes

```
quadrat (3+4);  => (3+4)*(3+4)  (quadrat)
                => 7*(3+4)      (+)
                => 7*7          (+)
                => 49           (*)
```

1/0; => ⊥

Typen

a) einfache Typen: unit, bool, int, real, char, string

b) abgeleitete Typen:	Tupel	(int, char)
	Funktionen	int → int
	Listen	[char]

Der Typ eines Ausdrucks hängt, genauso wie sein Wert, ausschließlich von den Werten der in ihm enthaltenen Ausdrücke ab. (**strong-typing).**

Funktionen und Definitionen

- In der funktionalen Programmierung können Funktionen auch als Werte angesehen werden, und daher als Argumente an andere Funktionen übergeben werden.

- Für ein und dieselbe Funktion gibt es verschiedene Definitionsmöglichkeiten.
  ```
  fun verdoppeln1  x  =  x + x;
  fun verdoppeln2  x  =  2*x;
  ```
- Manche Funktionen besitzen sehr allgemeine Quell- bzw. Zieltypen.
  ```
  - fun  id x = x;
  val id = fn : 'a -> 'a          (* polymorpher Typ *)
  ```
- Bestimmung des Werts einer Funktion durch Fallunterscheidungen.
  ```
  fun min x y:int = if  x>y  then  y  else x;
  ```
- In mathematischen Ausdrücken findet man häufig *lokale Definitionen*.
  ```
  fun f (x:real,y:real) =
      let val a=(x+y)/2.0          (* Sei ... *)
      in  (a+1.0)*(a+2.0)
      end;
  ```

Das Curry-Prinzip

```
- fun   min(x,y:int)  = if  x>y  then  y  else x;
val min1 = fn : int * int -> int

- fun   min x y:int   = if  x>y  then  y  else x;
val min2 = fn : int ->    int -> int

- min 5;
val it = fn : int -> int
- it 3;
val it = 3 : int
```


Spezifikation und Implementierung

Spezifikation

erhöhen x > quadrat x $\forall x \geq 0$

Implementierung

fun erhoehen x = quadrat(x+1)

Beweis, dass die Implementierung der Spezifikation entspricht:

a) erhöhen x = quadrat (x+1) (* erhöhen *)
= (x+1)*(x+1) (* quadrat *)
= x*x+2*x+1 (* algebra *)
> x*x (* angenommen x ≥0 *)
= quadrat x (*quadrat *)

b) Es gilt: x + 1 > x
=> quadrat x + 1 = quadrat x
=> erhoehen x = quadrat x + 1

Arbeiten mit SML/NJ

- Start mit sml
- CTRL-C unterbricht den Compiler und kehrt zur Interaktionsebene zurück.
- CTRL-D beendet SML.
- use(“dateiname“) lädt ein ML-Programm