# Programmierung und Modellierung

Martin Wirsing

in Zusammenarbeit mit
Moritz Hammer

SS 2009

# Die Lehrenden

- Martin Wirsing
  - Ordinarius für Informatik, LMU
  - wirsing@lmu.de
  - Tel.: 089-2180-9154
  - Sprechstunde: Di, 11-12 h

- Moritz Hammer
  - Dipl. Inform. LMU
  - hammer@ifi.lmu.de
  - Tel.: 089-2180-9182
  - Sprechstunde:
  - Mo, 13-14 h

# Programmierung and Softwaretechnik

## Forschung

- Global Computing, Web Enginrg
- Theorie & Techniken der SW Konstruktion

## Tools

- SDE: *SENSORIA* Dev. Environment
- UWE: Design von Web Systemen
- REFLECT: Middleware für adaptive Systeme

## Projekte

- MAEWA II: Web Engineering
- GLOWA: Entscheidgunterstütz Klimawandel
- SENSORIA: SOA Engineering
- REFLECT: Adaptive SW
- InterLink, RAJA
- Elite-Master „Software Engineering“

## Team

Martin Wirsing,
Rolf Hennicker +
17 wiss. Mitarbeiter u.
Doktoranden

# Studentenbetreuer im Fachbereich Programmierung und Programmiersprachen

**Kontakt**
Alexander Pohl
Zimmer Z1.03
alexander.pohl@pms.ifi.lmu.de
http://www.pms.ifi.lmu.de/mitarbeiter/pohl
Telefon: 089 | 2180-9807
Sprechstunde: Dienstag, 16:00 – 18:00 Uhr, und nach Vereinbarung

**Blog**
http://www.pms.ifi.lmu.de/nblogn

**Gebiet**
Programmierung, Programmiersprachen und -paradigmen, Entwicklung von Webanwendungen

**Kurs: Webanwendungen mit Java**
Dienstags 18-20 Uhr, CIP-Pool Z8, Oettingenstraße 67
http://www.pms.ifi.lmu.de/mitarbeiter/pohl/kurse/sose09/jweb/

# Studentenbetreuer im Fachbereich Softwaretechnik und Tools

**Kontakt**
Laith Raed
Zimmer E 5
raed@pst.ifi.lmu.de
http://www.pst.ifi.lmu.de/Personen/team/raed
Telefon: 089 | 2180-9126
Sprechstunde: Mittwoch,14:00 – 16:00 Uhr, und nach Vereinbarung


**Gebiet**
Programmierung und Softwaretechnik

**Kurs: Java für Anfänger**
Montag16-18 Uhr, Geschw.Schol.Plz 1, E006
http://www.pst.ifi.lmu.de/Lehre/sose-2009/java-kurs-fur-anfanger

# Studentenbetreuer im Fachbereich Betriebssysteme und Basissoftware

**Kontakt**
Michael Stübiger
RBG
stuebiger@rz.ifi.lmu.de
https://wiki.cip.ifi.lmu.de/tiki-index.php?page=Studbetr_os
Telefon: 089 | 2180-9137
Sprechstunde: Montag bis Freitag, 13:00 – 17:00 Uhr

**Gebiet**
Computerprobleme, Linux, Rechnernutzung

## Kurs: Rechnerkennung

https://wiki.cip.ifi.lmu.de/tiki-index.php?page=Studbetr_os

# Kurse im Überblick

**Kurs I: Webanwendung mit Java <Alexander Pohl PMS-Lehrstuhl>**
**(Dienstags 18-20 Uhr, CIP-Pool Z8, Oettingenstraße 67)**

✓Allgemeine Grundlagen zum Internet, Web und zu Webanwendungen
✓Java Server Pages
✓Groovy und Grails
✓JBoss Seam
Webseite: http://www.pms.ifi.lmu.de/mitarbeiter/pohl/kurse/sose09/jweb/

**Kurs II: Java für Anfänger < Laith Raed: PST-Lehrstuhl>**
**(Montag 16:00–18:00 Uhr, Geschw.Schol.Plz 1, E006)**

✓ Grundlegende Programmierungsstrukturen
✓ Grundprinzipien der OOP
✓ Input und Output sowie Ausnahmebehandlung
✓ Datenstrukturen und generische Programmierung
Webseite: http://www.pst.ifi.lmu.de/Lehre/sose-2009/java-kurs-fur-anfanger

**Kurs III: Linux/Unix <Michael Stuebiger: RZ>**
**(Donnerstag 18:00 – 19:00 und 19:00 – 20:00 Raum 28 in OE.67)**
https://wiki.cip.ifi.lmu.de/tiki-index.php?page=Studbetr_os

# Software-Entwicklung

- Tätigkeiten bei der Software-Entwicklung
  - Anforderungsanalyse & Fachliche Konzeption
  - Technische Konzeption (Design, Entwurf)
  - Realisierung
  - Integration & Test
  - Wartung
- Während der fachlichen und technischen Konzeption (Anforderungsspezifikation und Entwurf) ist es wichtig, adäquate Modelle zu bilden, die eine Software-Realisierung ermöglichen.
- Zur Modellbildung gibt es unterschiedliche Paradigmen
  - Zustandsbasierte Modelle
    - aktionsbasiert (imperativ)
    - objekt-orientiert
  - Deklarative (zustandslose) Modelle
    - **funktional**
    - relational

# Unterschiedliche Programmierparadigmen

- *Imperative Algorithmen*:
  - Imperative Algorithmen spezifizieren die Abfolge von Aktionen, die typischerweise bestimmte Größen verändern.
  - Diese Auffassung spiegelt auch die technischen Möglichkeiten von Rechneranlagen wider, die Schritt für Schritt bestimmte grundlegende Aktionen ausführen können.
- *Funktionale Algorithmen*:
  - Funktionale Algorithmen spezifizieren eine auswertbare Darstellung des funktionalen Zusammenhangs zwischen Ein- und Ergebniswerten.
  - Diese Auffassung spiegelt ein höheres Abstraktionsniveau wider: Obwohl intern (d.h. zur Auswertung der entsprechenden Abbildung) wiederum bestimmte Schritte ausgeführt werden, ist die eigentliche Spezifikation des Algorithmus als Abbildung praktisch losgelöst von den technischen Möglichkeiten der Rechenanlage.
- *Objektorientierte Algorithmen*:
  - Objektorientierte Algorithmen spezifizieren eine höhere Abstraktionsebene zur Darstellung von komplexen Eingabe- und Ergebnisdaten sowie Zwischenzuständen während der Berechnung.
  - Zur eigentlichen Lösung der gegebenen Aufgabe können sie s… funktionale als auch die imperative Auffassung verwenden.

**Aus: P. Kröger WS 08/09**

# Beispiel aus Einführung in die Programmierung, WS 08/09

Ein Kunde kauft Waren für $1 \leq r \leq 100$ EUR und bezahlt mit einem 100 EUR Schein ($r$ sei ein voller EUR Betrag ohne Cent-Anteil).

Gesucht ist ein Algorithmus, der zum Rechnungsbetrag $r$ das Wechselgeld $w$ bestimmt.

Zur Vereinfachung nehmen wir an, dass $w$ nur aus 1 EUR oder 2 EUR Münzen 5 EUR Scheinen bestehen soll. Es sollen möglichst wenige Münzen/Scheine ausgegeben werden (also ein 5 EUR Schein statt fünf 1 EUR Münzen).

***Aus: P. Kröger WS 08/09***

# Datendarstellung

- Zunächst müssen wir zur Lösung dieser Aufgabe die Darstellung (Modellierung) der relevanten Daten festlegen.
- Für den Rechnungsbetrag $r$ ist dies trivial, denn offensichtlich ist $r \in \mathbb{N}$. Wir nehmen an, dass $r$ in Dezimaldarstellung gegeben ist.
- Das Wechselgeld $w$ kann auf verschiedene Weise modelliert werden, z.B. als Folge oder Multimenge von Wechselgeldmünzen. Ein aus zwei 1-EUR-Münzen, einer 2-EUR-Münze und zwei 5-EUR-Scheinen bestehendes Wechselgeld könnte als Folge $(1, 1, 2, 5, 5)$ dargestellt sein.
- Wir legen folgende Datendarstellung fest:
  - $r$: als natürliche Zahl in Dezimaldarstellung.
  - $w$: als Folge von Werten 1, 2 oder 5.
- Wir benutzen dabei die Bezeichnung $()$ für die *leere Folge* und die Funktion $\circ$ zur Konkatenation zweier Folgen wie oben. Desweiteren sagen wir auch "nimm $x$ zu $w$ hinzu" für die Operation $w \circ x$.

**Aus: P. Kröger WS 08/09**

# Diskussion

- In unserem Beispiel ist die Zuordnung eines Wechselgelds *w* zu einem Rechnungsbetrag *r* mathematisch nichts anderes als eine Abbildung

  $h : r \rightarrow$ "herauszugebendes Wechselgeld"

- Ein wichtiges Prinzip für den Allgemeinfall von Abbildungen auf natürlichen Zahlen (und anderen Mengen) ist das Prinzip der *Rekursion* …
- Kernidee einer rekursiven Definition einer Abbildung *f* mit Definitionsbereich N0 ist es, für einen oder mehrere *Basisfälle* die Abbildung explizit anzugeben (typischerweise *f* (0)) und den allgemeinen *Rekursionsfall f* (*n*) auf den Fall *f* (*n* − 1) zurückzuführen.

**Aus: P. Kröger WS 08/09**

# Variante des zweiten Lösungsansatzes

- Idee:
  Auch im Fall der Abbildung $h : \mathbb{N}_0 \rightarrow$ "Folgen über 1,2,5" kann man sehr leicht eine rekursive Definition finden, die die Operationen *DIV* und *MOD* nicht mehr verwendet.
- Algorithmus *Wechselgeld 5*

$$h(r) = \begin{cases} \text{falls } r = 100, \text{ dann } (), & (1) \\ \text{falls } 100 - r \geq 5, \text{ dann } (5) \circ h(r+5), & (2) \\ \text{falls } 5 > 100 - r \geq 2, \text{ dann } (2) \circ h(r+2), & (3) \\ \text{falls } 2 > 100 - r \geq 1, \text{ dann } (1) \circ h(r+1), & (4) \end{cases}$$

- Hier ist der Basisfall (1) für $h(100)$ definiert und die Rekursionsfälle (2),(3),(4) werden auf die Fälle $h(r+5)$, $h(r+2)$ und $h(r+1)$ zurückgeführt.

***Aus: P. Kröger WS 08/09***

# Variante des zweiten Lösungsansatzes

- Beispielanwendung $r = 81$:

$$
\begin{aligned}
h(81) &= (5) \circ h(86) && \text{(Fall 2)} \\
&= (5) \circ (5) \circ h(91) && \text{(Fall 2)} \\
&= (5) \circ (5) \circ (5) \circ h(96) && \text{(Fall 2)} \\
&= (5) \circ (5) \circ (5) \circ (2) \circ h(98) && \text{(Fall 3)} \\
&= (5) \circ (5) \circ (5) \circ (2) \circ (2) \circ h(100) && \text{(Fall 3)} \\
&= (5) \circ (5) \circ (5) \circ (2) \circ (2) \circ () && \text{(Fall 1)} \\
&= (5, 5, 5, 2, 2)
\end{aligned}
$$

***Aus: P. Kröger WS 08/09***

# Realisierung mit SML

```
exception Undefined;

(*well-defined only for 100 >= r >=0*)
fun h(r) : int list =
  if r = 100 then nil
  else if (100 - r) >= 5 then   5 :: h(r+5)
       else if  5 > (100 - r) andalso  (100 - r) >= 2
            then  2::h(r+2)
            else  if  2>(100-r) andalso  (100-r)>=1
                  then 1::h(r+1)
                  else raise Undefined;
```

# Rekursion und Terminierung

- Algorithmen, die Bezug auf sich selbst (oder auf Teile von sich) nehmen, heißen "rekursiv".
- **Rekursion** ist eine zentrale Technik der Informatik
  - rekursive Algorithmen wirken gelegentlich befremdlich bzw. gewöhnungsbedürftig
  - aber: Rekursion ist eine natürliche und eigentlich wohlvertraute Beschreibungstechnik für Verfahren

- **Beispiel**: Zeichnen eines Vierecks auf dem Boden:

  Zeichne eine Seite wie folgt:
  - Gehe 3 Schritte nach vorne und zeichne dabei eine Linie;
    wende Dich dann um 90 Grad nach rechts.
  - Wenn Du nicht am Startpunkt stehst, dann rufe denselben Algorithmus auf, (d.h. zeichne eine Seite unter Anwendung des oben geschilderten Verfahrens).

# Rekursion und Terminierung (II)

- Dieses Beispiel eines rekursiven Algorithmus stört uns nicht, weil
  - das Verfahren überschaubar ist,
  - die Terminierungsbedingung leicht erkennbar ist.
- Die Überprüfung der Terminierung ist bei rekursiven Algorithmen oft **nicht** trivial!

- **Im Beispiel:**

  Ohne die Bedingung "wenn Du nicht am Startpunkt stehst” würde der Viereck-Algorithmus den Zweck erfüllen, ein Viereck zu zeichnen,aber nicht terminieren.

- **Frage:** Terminiert der Wechselgeldalgorithmus? (Antwort später)

# Ziel der Vorlesung

- Einführung in grundlegende Prinzipien der **funktionalen**
  - **Programmierung und Datenmodellierung**
  - am Beispiel der Sprache **SML**

- Wesentlichen Themen sind:
  - Funktionen und Rekursion
  - Datentypen
  - Auswertung und Terminierung von Programmen
  - Semantik von Programmiersprachen.

# Übersicht

- Einführung in die Programmierung mit SML
- Rekursive Funktionen
- Termauswertung
- Typprüfung und Typinferenz
- Induktive Datentypen
- Pattern Matching
- Funktionen höherer Ordnung
- Ausnahmen
- Abstrakte Typen und Module
- Formale Beschreibung der Syntax und Semantik von Programmiersprachen

# Termine

## Regelmäßige Termine

- **Vorlesung**
  - Mittwoch, 9:00 - 12:00 (ct), Geschwister-Scholl-Platz 1, A140
- **Übung**
  - Montag, 12:00-14:00 (ct), Theresienstrasse 39, B 046
  - Montag, 14:00-16:00 (ct), Theresienstrasse 39, B 046
  - Montag, 16:00-18:00 (ct), Theresienstrasse 39, B 046
  - Montag, 18:00-20:00 (ct), Theresienstrasse 39, B 046
  - Dienstag, 12:00-14:00 (ct), Theresienstrasse 39, B 046
  - Dienstag, 16:00-18:00 (ct), Theresienstrasse 41, C 112

# Literatur

- Literatur zur Vorlesung sind vor allem die Skripten
  - Fred Kröger: Informatik I, LMU, WS 2005/2006
  - Francois Bry: Informatik I, LMU, 2002

Außerdem:

- Robert Harper: Programming in Standard ML.
  - http://www.cs.cmu.edu/People/rwh/introsml/index.htm
- Michael R. Hansen, H. Rischel: Introduction to Programming using SML, Addison-Wesley, 1999
  - http://www.it.dtu.dk/introSML/
- Lawrence Paulson: ML for the Working Programmer (Second Ed.)
  - MIT Press, 1996
  - http://www.cl.cam.ac.uk/users/lcp/MLbook/
- Gert Smolka: “Programmierung - eine Einführung in die Informatik mit Standard ML”, Oldenbourg Verlag, 2008.

# 1. Einführung in die Programmierung mit SML

- Ziele heute:
  - Historischen Hintergrund kennen lernen
  - Erste SML-Programme schreiben lernen
    - Konstanten,
    - Arithmetische Ausdrücke
    - Nichtrekursive Funktionen

# Funktionale Programmiersprachen

- Ein **funktionales Programm** kann verstanden werden
  - als Sammlung von Gleichungen, die Funktionen (im mathematischen Sinne) beschreiben.
- Funktionale Programmierung erlaubt es, Programme auf viele verschiedene Weisen zu komponieren, insbesondere:
  - Funktionen sind Daten und können als
    - Argumente von Funktionen übergeben und als
    - Resultate zurückgegeben werden.
- Theoretische Basis
  - λ-Kalkül (Lambda-Kalkül) von Alonzo Church
- Bekannte funktionale Programiersprachen sind
  - LISP – die “Urmutter” der fkt. Sprachen, dynamische Bindung
  - Scheme – eine einfache LISP-Variante
  - SML – eine typisierte fkt. Sprache, stat. Bindung
  - OCaml – objekt-orientierte Erweiterung von SML-Dialekt
  - Haskell – nichtstrikte reine fkt. Sprache
  - XSLT – Transformationssprache für XML

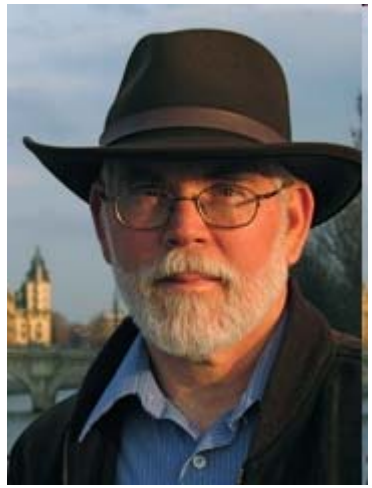
Alonzo Church
1903-1995
Logiker, λ-Kalkül
Church’sche
These

# SML

- Ca. 1975 Sprache ML entwickelt als Implementierungssprache (Metalanguage) von Robin Milner für den interaktiven LCF-Theorembeweiser
- 1984-87 Standard ML, definiert in

  Milner, Tofte, Harper, MacQueen: *Definition of Standard ML*, MIT Press, 1990
- Standard ML '97 aktuelle Version von SML

Robin Milner
* 1934, Entwickler von
LCF Theorembeweiser
CCS Kalkül für parallele Programme
Pi-Kalkül für mobile Programme
1991 Turing-Preis



David MacQueen
Koordinator für SML-NJ
Entwickler des Modulsystems für SML

# Zum Beginn

- SML wird mit dem Kommando “`sml`” aufgerufen.
- Eine SML-Sitzung wird mit ^D beendet (Ctrl + D bzw. Strg + D gleichzeitig).
- SML bietet eine interaktive, d.h. dialogorientierte, Benutzerschnittstelle mit einer sogenannten Treiberschleife:
  - "-” am Anfang der Zeile zeigt an, dass der Benutzer einen Ausdruck eingeben kann.
  - Das Ende des Ausdrucks wird mit “;” gekennzeichnet,
- Die Auswertung des Ausdruckes wird mit "enter” (bzw "return” / Zeilenumbruch) angefordert.

  SML wertet dann den Ausdruck aus und liefert den ermittelten Wert in einer neuen Zeile.

# Eine erste SML-Sitzung

```
Standard ML of New Jersey
   v110.69 [built: Thu Jan
   22 15:15:49 2009]
- 17;
val it = 17 : int
- 007;
val it = 7 : int
- ~5;
val it = ~5 : int
- ~(~5);
val it = 5 : int
- 5.1;
val it = 5.0 : real
```

```
- 3*(2+12);
val it = 42 : int
- 3*2+12;
val it = 18 : int
- (2*3)+12;
val it = 18 : int
- 12 div 5;
val it = 2 : int
- 14 div 5;
val it = 2 : int
- 14 mod 5;
val it = 4 : int
```

- "`it`" bezeichnet den unbenannten Wert des Ausdrucks, dessen Auswertung angefordert wird.
- "int" bezeichnet den (Daten-)Typ "integer" oder "ganze Zahl"
- "real" bezeichnet den (Daten-)Typ "Reelle Zahl" oder "Gleitpunktzahl"

# Die Datenstruktur “Ganze Zahl”

- Führende Nullen sind zulässig; z.B. 007 ist eine andere Notation für 7
- Vordefinierte Operationen über ganzen Zahlen:
  - Zweistellig: +, -, *, div, mod (alle infix notiert)
  - Einstellig: ~ (Vorzeichen für negative Zahlen)
    Vorsicht: ~ und – (Subtraktion sind nicht austauschbar)
- Präzedenzen der vordefinierten Operationen müssen beachtet werden; z.B. 2 * 3 + 1 steht für (2 * 3) + 1.
- Zum Vergleich von ganzen Zahlen bietet SML die vordefinierte Funktionen "=“ sowie “<“, “<=“, “>”, “>=“ an:
  - ```
    - 2 = 2;
    val it = true : bool
    ```
  - ```
    - 2 = 3;
    val it = false : bool
    ```
  - ```
    - 2 > 3;
    val it = false : bool
    ```
- Eine Funktion, die wie = als Wert entweder `true` oder `false` liefert, wird Prädikat oder auch Test genannt.
    ```

# Die Datenstruktur “Bool”

- Es gibt zwei Boole'sche Werte:
  - `true` (wahr) und `false` (falsch).
- Vordefinierte Operationen über Booleschen Werten:
  - **Zweistellig:** `andalso` (sequentielle Konjunktion) und `orelse` (sequentielle Disjunktion (beide infix notiert)
  - **Einstellig:** `not` (Negation)
- Vergleich Boolescher Werte mit "=“
  - ```
    - false = false;
    val it = true : bool
    ```
  - ```
    - true = false;
    val it = false : bool
    ```
- Weitere Beispiele
  - ```
    - not (not true);
    val it = true : bool
    ```
  - **Achtung**: `not not true` ist syntaktisch falsch, da `(not not) true` geklammert

George Boole
1815-1864
Engl. Mathematiker
Logikkalkül
Entscheidungs-verfahren für Aussagenlogik
    ```

# Überladen

- Gleichheit für Boole'sche Ausdrücke und für ganze Zahlen:
  - syntaktisch identischer Operator (=)
  - aber zwei grundverschiedene Funktionen, weil ihre Argumente verschiedene Typen besitzen
- SML betrachtet die Typen der Argumente, um zu entscheiden, welche Gleichheitsfunktion angewendet werden soll:
  - der Operator = ist “überladen” (overloaded),
  - Ebenso sind + und * überladen
- **Überladen**: Derselbe Name (Bezeichner) wird zur Bezeichnung unterschiedlicher Operationen oder Funktionen verwendet, die vom System unterschieden werden.
- Beispiele

```
- 2 + 3;
val it = 5 : int
- 2.1 + 3.3;
val it = 5.4 : real
```

```
- 2 * 3;
val it = 6 : int
- 2.1 * 3.3;
val it = 6.93 : real
```

# Überladen

- Achtung:
  - Überladen von Bezeichnern ist nicht ungefährlich und sollte nur in klar abgegrenzten Fällen verwendet werden.
  - In der Mathematik ist die Menge der ganzen Zahlen ein Untertyp (Teilmenge) der reellen Zahlen.
  - In SML wie in den meisten anderen Programmiersprachen stellen ganze Zahlen und reelle Zahlen verschiedene (Daten-)Typen dar.
    - Im Rechner werden ganze Zahlen und reelle Zahlen völlig unterschiedlich repräsentiert und die arithmetischen Operationen unterschiedlich implementiert.
    - Der Versuch, eine ganze Zahl und eine reelle Zahl zu addieren, führt folglich zu einer Fehlermeldung:

```
- 2 + 4.83;
stdIn:10.1-10.9 Error: operator and operand don't agree
  [literal]
operator domain: int * int
operand: int * real
in expression:
2 + 4.83
```

      Wegen des ersten Operanden 2 wird + als Addition für ganze Zahlen interpretiert. Da 4.83 keine ganze Zahl ist, wird ein Typfehler gemeldet.

# Weitere Typen und Operationen

- SML bietet
  - weitere vordefinierte Typen, wie z.B.
    - `string` “Zeichenfolge”, wie z.B. `“Hello World!”`
    - `char` "Zeichen”, wie z.B. `#"a", #”b", #”X”`
  - die Definition von eigenen, maßgeschneiderten Typen, wie etwa für
    - die Tage der Woche in einer beliebigen Sprache
    - Binärbäume, formale Sprachen, …

- Weitere nützliche Operationen
  - `Int.abs`: Betrag einer ganzen Zahl
    - `- Int.abs(~4);`
      `val it = 4 : int`
  - `Int.min, Int.max`: Minimum und Maximum zweier ganzen Zahlen
    - `- Int.min(5,2);`
      `val it = 2 : int`
  - `Int.sign`: Vorzeichen einer ganzen Zahl
    - `- Int.sign(0);`
      `val it = 0 : int`
    - `- Int.sign(~5);`
      `val it = ~1 : int`

# Ausdrücke, Werte, Typen

- SML (genauer: das SML-System) wertet Ausdrücke aus.
- Ein Ausdruck kann
  - atomar sein, wie z.B. `17`, `false`, oder
  - zusammengesetzt sein, wie z.B. `12+4`, `not(false andalso true)`.
- Jeder korrekt gebildete Ausdruck besitzt einen Typ:
  - ein Typ ist eine Menge von Werten; z.B. die Menge der ganzen Zahlen
  - Z.B. `int` ist der Typ von `12+4`
- Ein Ausdruck hat (meistens) auch einen Wert;
  - Dieser Wert ist ein Element des Typs des Ausdrucks,
  - Z.B. `16` ist der Wert von `12+4`
- Manche Ausdrücke haben keinen Wert;
  - Z.B. `1 div 0` (da hier eine nicht-totale Funktion verwendet wird)

# Ausdrücke, Werte, Typen

- Auch Operationen (und allgemein Funktionen) haben Typen, z.B.
  - die Funktion + erhält als Argumente zwei (atomare oder zusammengesetzte) Ausdrücke vom Typ `int` und liefert einen Wert ebenfalls vom Typ `int`.
  - die Gleichheit für ganze Zahlen ist eine Funktion, die als Argumente zwei Ausdrücke vom Typ `int` erhält und einen Wert vom Typ `bool` liefert.
- Man schreibt:
  - `+ : (int, int)` $\rightarrow$ `int`
  - `= : (int, int)` $\rightarrow$ `bool`
- Bei der Bildung zusammengesetzter Ausdrücke muss immer auf die Typen der verwendeten Operationen und der eingesetzten Teilausdrücke geachtet werden.

# Namen, Bindungen und Deklarationen

- Mit einer **Deklaration** kann ein **Wert** an einen **Namen** gebunden werden.
- Mögliche Werte, die an Namen gebunden werden können, sind u.a.
  - Konstanten (Kostantendeklaration) und
  - Funktionen (Funktionsdeklaration).

# Konstantendeklaration

- Beispiel
  - `- val zwei = 2;`
    `val zwei = 2 : int`
- Damit wird die Konstante `zwei` deklariert und der Name `zwei` kann genauso wie die Konstante 2 verwendet werden:
  - `- zwei + zwei;`
    `val it = 4 : int`
  - `- zwei * 8;`
    `val it = 16 : int`

# Funktionsdeklaration und Funktionsaufruf

- Beispiel
  - ```
    - fun zweimal(x) = 2 * x;
    val zweimal = fn : int -> int
    ```

  Damit wird die Funktion `zweimal` deklariert.
- Der Wert des Namens `zweimal` ist die Funktion, die als Eingabe eine ganze Zahl erhält und das Doppelte dieser Zahl als Ausgabe liefert.
- Anstelle des Wertes der Funktion, die an den Namen zweimal gebunden wird, gibt SML die Kurzmitteilung `fn` (für Funktion) und den Typ der Funktion aus.
- Nachdem eine Funktion deklariert wurde, kann sie **aufgerufen** werden:
  - ```
    - zweimal(8);
    val it = 16 : int
    ```
  - ```
    - zweimal(zweimal(8));
    val it = 32 : int
    ```
    ```

# Funktionsdeklaration

- Der Typ `int -> int` der Operation

```
fun zweimal(x) = 2 * x;
```

  wird wie folgt ermittelt:
  - Wegen des Schlüsselworts `fun` muss der Typ die Form `A -> B` haben.
  - Da 2 eine ganze Zahl ist, steht die überladene Operation * für die Multiplikation ganzer Zahlen.
  - Folglich muss der formale Parmater `x` vom Typ `int` sein (daher: `int ->`).
  - Da * die Multiplikation ganzer Zahlen ist, ist der Resultattyp ebenfalls `int` (daher: `int ->`).

# Funktion als Wert - Anonyme Funktion

- Für SML ist eine Funktion ein Wert.
  - Verwendung des Deklarationskonstrukts `val` möglich

- Die Funktion `zweimal` kann auch wie folgt definiert werden:
  - `val zweimal = fn x => 2 * x;`
- Hier passiert folgendes:
  - Die rechte Seite fn x => 2 * x definiert eine **anonyme** Funktion.
    In Anlehnung an den λ-Kalkül wird `fn` oft "lambda" ausgesprochen.
  - Diese anonyme Funktion wird an den Namen `zweimal` gebunden.

- **Vorsicht**: Verwechseln Sie die SML-Konstrukte `fn` und `fun` nicht!

# Formale, aktuelle Parameter und Rumpf einer Funktion

- In der Funktionsdeklaration
  - `fun zweimal(x) = 2 * x;`

  ist `x` ein **formaler Parameter** und `2 * x` der **Rumpf** (der Funktionsdeklaration oder Funktionsdefinition).
- Im Funktionsaufruf `zweimal(8)` ist `8` der **aktuelle Parameter** (des Funktionsaufrufes).

# Typ-Constraints

- Da `x + x = 2 * x` ist, könnte man die Funktion `zweimal` wie folgt definieren:
  - - fun zweimal(x) = x + x;
- Dies wird aber nicht von allen SML-Systemen als korrekt angenommen:
  - Der Typ des formalen Parameters `x` ist nicht eindeutig ( `int` oder `real`).
  - Manche Systeme nehmen an, dass `x` den Typ `int` hat, weil sie im Zweifel + als Addition von ganzen Zahlen annehmen.
  - Andere SML-Systeme machen keine solche Annahme und verwerfen die vorangehende Funktionsdeklaration als inkorrekt.
- **Typ-Constraints** (auch: Typisierungsausdrücke) ermöglichen, die fehlende Typinformation anzugeben.

# Typ-Constraints

- Mit
  - `- fun zweimal(x: int) = x + x;`

  wird der Typ des Parameters angegeben.
- Mit
  - `- fun zweimal x: int = x + x;`

  wird der Typ des Ergebnisses (des berechneten und gelieferten Wertes) angegeben.
- Mit
  - `- fun zweimal(x: int): int = x + x;`

  werden sowohl der Typ des Ergebnisses als auch der Typ des Parameters angegeben.
- Mit folgendem Typ-Constraint wird `zweimal` für reelle Zahlen definiert:
  - ```
    - fun reell_zweimal (x:real) = x + x;
    val reell_zweimal = fn : real -> real
    ```
    ```

# Geek Logic

- Mathematische Ausdrücke zur Lösung von Problemen des täglichen Lebens
  - Eingeführt 1994 von Garth Sundem an der Universität Cornell
  - Aus G. Sundem: 50 Foolproof Equations for Everyday Life; New York: Workman Pub., 2006
- **„Soll ich heute das tun, was ich auf morgen verschieben kann?“**

$$doIt = \frac{50.0*schaden/tage^2 + aerger^2 + unangenehm*einfach}{aufschubErfahrung*anderer*unangenehm;}$$

  - Setze für die Variablen `schaden, aerger, unangenehm, aufschubErfahrung, anderer` Werte zwischen 1 und 10 ein;
  - Setze für `einfach` Werte zwischen -5 und +5 ein; < 0, wenn die Aufgabe mit der Zeit einfacher wird; > 0, wenn sie schwieriger wird
  - `tage` = Anzahl der Tage, bis die Aufgabe essentiell wichtig wird.

Falls doIt > 1, dann sollte die Aufgabe besser heute erledigt werden.

# Geek Logic (SML)

```
fun doIt(katastrophe, tage, unangenehm, einfach,
         anderer, aerger, aufschubErfahrung): real =
(50.0*katastrophe/(tage*tage) + aerger*aerger + unangenehm*einfach)
   /(aufschubErfahrung*anderer*unangenehm);
```

```
(*** Bad reinigen ***)

val Katastrophe = 3.0;
val Tage = 10.0 ;
val Unangenehm= 6.0 ;
val Einfach = 0.0 ;
val Anderer = 7.0 ;
val AErger = 5.0 ;
val AUFschubErfahrung = 7.0 ;
```

```
(*** Proseminarvortrag
     vorbereiten   ***)

val Katastrophe = 8.0;
val Tage = 10.0 ;
val Unangenehm= 4.0 ;
val Einfach = 4.0 ;
val Anderer = 1.0 ;
val AErger = 5.0 ;
val AUFschubErfahrung = 7.0 ;
```

```
doIt(Katastrophe, Tage, Unangenehm, Einfach,
     Anderer, AErger, AUFschubErfahrung);
```

# Zusammenfassung

- Ziel der Vorlesung ist die Einführung in grundlegende Prinzipien der funktionalen
  - Programmierung und Datenmodellierung
  - am Beispiel der Sprache SML
- Ein funktionales Programm kann verstanden werden als Sammlung von Gleichungen, die Funktionen (im mathematischen Sinne) beschreiben.
  - Theoretische Basis funktionaler Programmierung ist der λ-Kalkül (Lambda-Kalkül) von Alonzo Church
  - Bekannte funktionale Programiersprachen sind LISP, Scheme, SML, OCAML, Haskell, XSLT
  - SML ist entstanden aus Robin Milners Implementierungssprache ML für den LCF-Theorembeweiser; aktuelle Version ist SML ’97
- Vordefinierte Datentypen von SML sind u.a. `int, bool, real, char, string`
- Überladen: Derselbe Name (Bezeichner) wird zur Bezeichnung unterschiedlicher Operationen oder Funktionen verwendet, die vom System unterschieden werden.
- Jeder korrekt gebildete SML-Ausdruck besitzt einen Typ; jeder definierte Ausdruck hat (meistens) auch einen Wert. Neben Elementen der Standarddatentypen sind auch Funktionen Werte.
- Typ-Constraints (auch: Typisierungsausdrücke) ermöglichen, die fehlende Typinformation anzugeben.