Ludwig Maximilians Universität München
Institut für Informatik
Lehr- und Forschungseinheit für Datenbanksysteme

Skript zur Vorlesung
Informatik I
Wintersemester 2006

# Kapitel 3: Das Substitutionsmodell (zur Auswertung von rein funktionalen Programmen)

Vorlesung: Prof. Dr. Christian Böhm
Übungen: Elke Achtert, Arthur Zimek



http://www.dbs.ifi.lmu.de/Lehre/Info1

Database Systems

## Inhalt

1. Auswertung von Ausdrücken
2. Auswertung in applikativer und normaler Reihenfolge
3. Verzögerte Auswertung
4. Auswertung von Sonderausdrücken
5. Funktionale Variablen versus Zustandsvariablen
6. Funktionale Programmierung versus Imperative Programmierung

# Überblick

1. Auswertung von Ausdrücken
2. Auswertung in applikativer und normaler Reihenfolge
3. Verzögerte Auswertung
4. Auswertung von Sonderausdrücken
5. Funktionale Variablen versus Zustandsvariablen
6. Funktionale Programmierung versus Imperative Programmierung



# Arten von Ausdrücken

- Nicht alle Ausdrücke haben den selben Zweck!
- Vom Zweck eines Ausdrucks hängt ab, wie er ausgewertet wird.
- Konstanten- und Funktionsdeklarationen binden Werte (das sind Konstanten oder Funktionen) an Namen:

```
val zwei = 2;
fun quadrat (x: int) = x * x;
```

- Funktionsanwendungen wenden Funktionen auf Werte an:

```
quadrat(3 + zwei)
quadrat(3) + quadrat(2)
```

- Zunächst unterscheiden wir verschiedene Arten von Ausdrücken:
  - Funktionale Ausdrücke
  - Sonderausdrücke

# Funktionale Ausdrücke

- Funktionale Ausdrücke sind Konstanten und Funktionsanwendungen.
- Ein funktionaler Ausdruck kann atomar sein:
  - `3`
  - `2`
  - `true`
  - `false`
  - `zwei`
- Ein funktionaler Ausdruck kann zusammengesetzt sein:
  - `quadrat(3 + 2)`
  - `3 * 2`
  - `not false`
  - `not true <> true`



# Sonderausdrücke

- Sonderausdrücke sind die Ausdrücke, die keine funktionalen Ausdrücke sind.
- In SML sind das die mit folgenden vordefinierten Konstruktoren gebildeten Ausdrücke:
  - `val` und `fun`, die zur Wertedeklaration dienen
  - `if-then-else`, `case` und das *Pattern Matching* zur Fallunterscheidung
  - Die Boole´schen Operatoren `andalso` und `orelse`
- Die mit `andalso` bzw. `orelse` gebildeten Ausdrücke sind KEINE funktionalen Ausdrücke (warum sehen wir später)!

Im Laufe des Semesters werden wir weitere SML-Konstrukte zur Bildung von Sonderausdrücken kennen lernen.

# Die Auswertung von Ausdrücken als Algorithmus

- In welchem Formalismus soll die Auswertung von Ausdrücken spezifiziert werden?
  - Jeder ausreichend präzise und verständliche Formalismus ist geeignet.
- Warum muss man die Auswertung von Ausdrücken als Algorithmus formalisieren?
  - Weil es sich um eine (symbolische) Berechnung handelt (wie z.B. die Multiplikation zweier natürlicher Zahlen).
  - Weil die Auswertung von Ausdrücken auf einem Computer durchgeführt werden soll.



# Occam´sches Messer

- Das Grundprinzip des Algorithmusbegriffs besteht darin, nicht mehr Begriffe, Methoden und Annahmen als nötig einzuführen.
- Bereits im 14. Jh. wurde dieses Prinzip von dem englischen Franziskaner-Mönch und Philosoph, Wilhelm von Occam, als philosophisches und wissenschaftliches Prinzip formuliert:

  → Das so genannte *Occam´sche Messer*

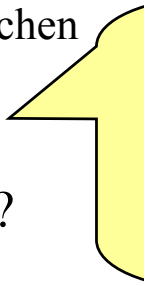
Schneide nutzlose Begriffe, Methoden oder Annahmen heraus.

*Nebenbei*: Der Roman „Der Name der Rose“ von Umberto Eco, dessen Hauptfigur von Occam inspiriert wurde, führt leicht verständlich in die Occam´sche philosophische Lehre ein.

# Der Algorithmusbegriff

- Der Algorithmusbegriff ist zur Spezifikation der Auswertung von Ausdrücken geeignet.
- Die Durchführung der Auswertung auf einem Computer ist NICHT der Hauptgrund, weswegen der Algorithmusbegriff verwendet wird.

  →Algorithmen und Programme werden zunächst für Menschen verfasst und nur nebenbei für Computer!
- Was wäre der Nutzen einer Auswertung von Ausdrücken, die Menschen nicht verstehen könnten?





# Die Auswertung von Ausdrücken

- Welche Art von Algorithmus eignet sich für die Auswertung von Ausdrücken?

  →Rekursive Funktionen spezifizieren einfach, prägnant und informell die Auswertung von Ausdrücken.

# Skizze des Auswertungsalgorithmus

Zur Auswertung eines Ausdrucks `A` gehe wie folgt vor:

1. Werte die Teilausdrücke von `A` aus.
2. Wende die Funktion, die sich als Wert des am weitesten links stehenden Teilausdrucks ergibt, auf die Werte an, die sich als Werte aus der Auswertung der restlichen Teilausdrücke ergeben.

> ! Diese Skizze eines Algorithmus ist rekursiv, weil sie die Auswertung von Teilausdrücken anfordert.
>
> Natürlich ist diese Skizze noch unvollständig und unpräzise.



# Annahmen des Auswertungsalgorithmus

- Für alle Funktionen gelte die Präfix- statt der Infixschreibweise, z.B.: `+(1, 2)` statt `1+2`
- Gewisse *Systemfunktionen*, die man nicht definieren muss, stehen zur Verfügung, z.B. die Addition oder die Multiplikation.
- Andere Funktionen können definiert werden: (z.B.)

```
fun quadrat (x) = *(x,x)
val quadrat = fn(x) => *(x,x)
```

Äquivalent!

- In der aktuellen Umgebung hat der Name „`quadrat`“ also die Funktion `fn(x) => *(x,x)` als Wert.

# Auswertungsalgorithmus (1)

Zur Auswertung eines Algorithmus A gehe wie folgt vor:

1. Falls A atomar ist, dann:
   a) Falls A vordefiniert ist, liefere den vordefinierten Wert von A (dieser kann auch eine Systemfunktion sein).
   b) Anderfalls (der Wert von A ist in der Umgebung definiert) sei A = W die Gleichung in der Umgebung, die den Wert von A definiert.
      Liefere W als Wert von A (W kann auch eine Funktion der Form $fn(F_1,\ldots,F_k)$ => R sein).



# Auswertungsalgorithmus (2)

2. Andernfalls (A ist zusammengesetzt) hat A die Form $B(A_1,\ldots,A_n)$ mit $n \geq 0$. Werte die Teilausdrücke $B, A_1, \ldots, A_n$ aus. Seien $W_1,\ldots,W_n$ die Werte der Teilausdrücke $A_1,\ldots,A_n$.
   a) Falls der Wert von B eine Systemfunktion ist, dann: Wende sie auf $(W_1,\ldots,W_n)$ an. Liefere den dadurch erhaltenen Wert als Wert von A.
   b) Falls der Wert von B eine Funktion der Form $fn(F_1,\ldots,F_n)$ => R ist, dann: Werte R in der erweiterten Umgebung aus, die aus der aktuellen Umgebung und den zusätzlichen Gleichungen $F_1 = W_1, \ldots, F_n = W_n$ besteht. Liefere den dadurch erhaltenen Wert als Wert von A (die Umgebung ist nun wieder die ursprüngliche).

# Betrachtung von Fall 2

- Muss der Teilausdruck `B` in Fall 2 tatsächlich erst ausgewertet werden?
- Betrachte die folgenden 3 Unterfälle:
  - Betrachte Fall 2.a
  - Betrachte Fall 2.b
  - `B` sei ein zusammengesetzter Ausdruck



# Unterfall 1

- Ein zusammengesetzter Ausdruck `A` beginne mit `abs`, `B` ist also der Ausdruck `abs.`
- Die Auswertung von `abs` liefert die einstellige Systemfunktion, die eine ganze Zahl als Argument nimmt und deren Absolutbetrag als Wert liefert (Fall 2.a).
- `abs` ist nur der Name dieser Systemfunktion, aber nicht die Funktion selbst.
- Also muss `B` zunächst ausgewertet werden.

! Systemfunktionen werden häufig durch die Maschinensprache des zu Grunde liegenden Computers zur Verfügung gestellt (siehe auch die Grundstudiums-Vorlesung „Informatik III“ und die Hauptstudiums-Vorlesung „Übersetzerbau“)

# Unterfall 2

- Sei `quadrat` wie vorher definiert, so dass der Name „`quadrat`“ in der aktuellen Umgebung die Funktion `fn(x) => *(x,x)` als Wert hat.
- `A` sei nun der zusammengesetzte Ausdruck `quadrat(2)`, `B` ist also der Ausdruck `quadrat`.
- `B` muss zunächst ausgewertet werden, um die Funktion `fn(x) => *(x,x)` zu erhalten (Fall 2.b), die dann auf die natürliche Zahl `2` angewandt wird.



# Unterfall 3

- `B` selbst sei ein zusammengesetzter Ausdruck.
- Sei `A` der Ausdruck

  `(if n > 0 then quadrat else abs)(5)`,

  also ist `B` der Ausdruck

  `(if n > 0 then quadrat else abs)`
- `B` muss zunächst ausgewertet werden, um die Funktion zu erhalten, die dann auf den Wert `5` angewandt wird.

→ Insgesamt definiert der beschriebene Auswertungsalgorithmus also eine Funktion, die als Eingabeparameter einen Ausdruck und eine Umgebung erhält und als Wert den Wert des Ausdrucks liefert.

# Unvollständigkeit des genannten Algorithmus

- Diese Spezifikation des Auswertungsalgorithmus ist nicht vollständig:
  - Sie berücksichtigt keine Programmfehler.
  - Sie berücksichtigt keine Typen (siehe Kapitel 6).
  - Sie kann einige Konstrukte von SML gar nicht behandeln.
  - Die Behandlung der Umgebung ist nur unpräzise erläutert

Die Ergänzung des Auswertealgorithmus ist nicht schwierig, wird in diesem Kapitel jedoch nicht weiter behandelt.



# Genauere Betrachtung der Fehler des Auswertealgorithmus

- Wenn A ein Name ist, der nicht deklariert wurde, tritt in Fall 1.b ein Fehler auf.
- Wenn die Anzahl n der aktuellen Parameter $A_1, \ldots, A_n$ mit der Stelligkeit des Wertes von B nicht übereinstimmt, tritt in Fall 2.a und 2.b ein Fehler auf.
- Die Umgebung, die als geordnete Liste verwaltet wird (vgl. Kapitel 2), wird in diesem Algorithmus nicht näher spezifiziert.
- Der Auswertungsalgorithmus in dieser Form kann folgende Konstruktoren nicht behandeln:
  - alle Sonderausdrücke
  - Funktionen, deren formale Parameter komplexe Pattern sind
  - einige andere Konstrukte

# Zweckmäßigkeit des genannten Algorithmus (1)

- Kann man den Algorithmus als rekursive Funktion spezifizieren, wenn er selbst u.a. zur Auswertung von rekursiven Funktionen dienen soll?

Dreht sich so eine Spezifikation nicht im Kreis?



# Zweckmäßigkeit des genannten Algorithmus (2)

- Die Annäherung an den Begriff „Algorithmus“ sollte zu einfacheren „Urbegriffen“ führen, mit denen komplexere Begriffe definiert werden können.
- Auf dem Weg zu solchen Definitionen darf man sich des informellen Verständnisses von komplexen Begriffen bedienen:

  Unsere Intuition von der Auswertung von (rekursiven) Funktionen hilft uns zu verstehen, wie (rekursive) Funktionen ausgewertet werden.

→ Die Durchführbarkeit des Algorithmus zeigt, dass wir uns mit seiner Spezifikation nicht im Kreis drehen.

# Beispiel einer Durchführung des Auswertungsalgorithmus (1)

- Seien folgende Deklarationen gegeben:

```
val zwei = 2;
fun quadrat (x) = *(x,x);
```

- Die Umgebung besteht also aus den beiden Gleichungen `quadrat = fn(x) => *(x,x)` und `zwei = 2`.
- Sei `A` der Ausdruck `quadrat(zwei)`, der in dieser Umgebung ausgewertet werden soll.



# Beispiel einer Durchführung des Auswertungsalgorithmus (2)

2. `A` ist zusammengesetzt: `B` ist `quadrat`, $A_1$ ist `zwei`, `n = 1`.
   Werte den Teilausdruck `B` aus;
   *Nebenrechnung*, in der `A` der Ausdruck `quadrat` ist:
   1. `A` ist atomar.
      b) `A` ist nicht vordefiniert.
      Als Wert von `quadrat` wird aus der Umgebung `fn(x) => *(x,x)` geliefert.

   *Ende der Nebenrechnung*; Wert von `B` ist `fn(x) => *(x,x)`.
   Werte den Teilausdruck $A_1$ aus;
   *Nebenrechnung*, in der `A` der Ausdruck `zwei` ist:
   1. `A` ist atomar.
      b) `A` ist nicht vordefiniert.
      Als Wert von `zwei` wird aus der Umgebung die natürliche Zahl `2` geliefert.

   *Ende der Nebenrechnung*; Wert von $A_1$ ist `2`.

# Beispiel einer Durchführung des Auswertungsalgorithmus (3)

2. b) Der Wert von B ist keine Systemfunktion, sondern eine Funktion
`fn(x) => *(x,x)`.

Die erweiterte Umgebung besteht aus der aktuellen Umgebung und der zusätzlichen Gleichung `x = 2`.

Werte `*(x,x)` in dieser erweiterten Umgebung aus;

*Nebenrechnung*, in der A der Ausdruck `*(x,x)` ist:

2. A ist zusammengesetzt.

B ist *, $A_1$ ist x, $A_2$ ist x, `n = 2`. Werte den Teilausdruck B aus;

*Nebenrechnung*, in der A der Ausdruck * ist:

1. A ist atomar.

a) A ist vordefiniert.

Wert von * ist die zweistellige Multiplikations-funktion, also eine Systemfunktion.

*Ende der Nebenrechnung*; Wert von B ist die Multiplikationsfunktion.



# Beispiel einer Durchführung des Auswertungsalgorithmus (4)

Werte den Teilausdruck $A_1$ aus;

*Nebenrechnung*, in der A der Ausdruck x ist:

1. A ist atomar.

b) A ist nicht vordefiniert.

Als Wert von x wird aus der (erweiterten) Umgebung die natürliche Zahl 2 geliefert.

*Ende der Nebenrechnung*; Wert von $A_1$ ist 2.

Genauso: Wert von $A_2$ ist 2.

a) Der Wert von B ist eine Systemfunktion, nämlich die Multiplikationsfunktion.

Wende sie auf `(2, 2)` an.

Der dadurch erhaltene Wert ist 4.

*Ende der Nebenrechnung*, Wert von `*(x,x)` ist 4.

Der dadurch erhaltene Wert von A ist also 4.

(die Umgebung ist nun wieder die ursprüngliche, ohne die Gleichung x=2)

# Vermerktechnik (1)

- Die Notation `fn(x) => *(x,x)` wird für eine vom Benutzer selbst definierte, einstellige Funktion verwendet, die der eigentliche Wert von `quadrat` ist.
- In Implementierungen wird statt dieser textuellen Notation eine Speicheradresse benutzt, an der sich ein „Funktionsdeskriptor“ befindet.
- Ein „Funktionsdeskriptor“ enthält die formalen Parameter, die Typen und den Rumpf der Funktion.
- Intern wird als „Wert“ für `quadrat` die Speicheradresse dieses Funktionsdeskriptors geliefert.
- Das SML-System nutzt einen Teil der dort vorhandenen Angaben, um die Ausgabe `fn: int -> int` (also den Typ von `quadrat`) zu ermitteln.



# Vermerktechnik (2)

- Das Symbol `fn` (oder die Speicheradresse für einen Funktionsdeskriptor) stellt also einen Vermerk dar.
- Diese Vermerktechnik ist notwendig, weil ein Programm nur Bezeichner (auch Symbole genannt) bearbeitet.
- Ein Begriff wie der Funktionsbegriff wird erst durch einen Algorithmus (wie der genannte Auswertungsalgorithmus) verwirklicht.
- Bis zu dieser Verwirklichung können nur symbolische Berechnungen stattfinden, d.h. eine Bearbeitung von Bezeichnern, wie `fn`.

# Vergleich mit der Mathematik

- Es gibt einen wesentlichen Unterschied zu Berechnungen in der Mathematik:
- Mathematiker führen Teilberechnungen, z.B. Teilbeweise, durch, die nicht immer durch einen ausformulierten Algorithmus spezifiziert sind, der penibel Schritt für Schritt durchgeführt wird.
- Trotzdem sind präzise ausformulierte Berechnungs- und Beweisalgorithmen in der Mathematik unverzichtbar.
- Für Beweise stellt die präzise Formulierung der Algorithmen eine der zentralen Aufgaben des Teilgebiets der mathematischen Logik dar.



# Das Substitutionsmodell

- Der Auswertungsalgorithmus kann auf einer höheren Abstraktionsebene so erläutert werden:

  Um einen Ausdruck A auszuwerten, werden alle Teilausdrücke von A durch ihre Definitionen ersetzt, wobei
  - die formalen Parameter von Funktionsdefinitionen durch die aktuellen Parameter der Funktionsanwendung ersetzt werden,
  - vordefinierte Konstanten gemäß ihrer Definition durch ihre Werte ersetzt werden
  - und vordefinierte Funktionen gemäß ihrer Definition ersetzt werden.
- Diese Beschreibung der Auswertung wird *Substitutionsmodell* genannt.

Das *Substitutionsmodell* ist unpräziser als der behandelte Algorithmus.

# Überblick

1. Auswertung von Ausdrücken
2. Auswertung in applikativer und normaler Reihenfolge
3. Verzögerte Auswertung
4. Auswertung von Sonderausdrücken
5. Funktionale Variablen versus Zustandsvariablen
6. Funktionale Programmierung versus Imperative Programmierung



# Auswertungsreihenfolge

- Der (präzise) Auswertungsalgorithmus legt die Reihenfolge der Auswertung der Teilausdrücke eines zusammengesetzten Ausdrucks fest.
- Das (abstrakte) Substitutionsmodell definiert diese Reihenfolge nicht näher.

# Auswertungsreihenfolge des Auswertungsalgorithmus (1)

- Betrachte den Ausdruck `quadrat(2+1)`:
  - Der Auswertungsalgorithmus „merkt“ sich mit dem Vermerk „`fn`“ (als Pseudo-Wert eines Funktionsnamens) die in der Umgebung definierte Funktion `quadrat`.
  - Dann wird der Teilausdruck `2+1` ausgewertet.
  - Erst dann wird die Funktion namens `quadrat` auf den so ermittelten Wert `3` von `2+1` angewandt, was zur Auswertung des Ausdrucks `3*3` führt.



# Auswertungsreihenfolge des Auswertungsalgorithmus (2)

- Der Auswertungsalgorithmus wertet zunächst die aktuellen Parameter (oder Operanden) einer Funktionsanwendung aus, bevor er die Funktion auf die Werte dieser Parameter anwendet.
- Diese Reihenfolge ist im Algorithmus dadurch vorgegeben, dass in Schritt 2 die Teilausdrücke $B, A_1, \ldots, A_n$ ausgewertet werden und erst dann der Wert von $B$ (der eine Funktion sein muss) auf die Werte von $A_1, \ldots, A_n$ angewandt wird.

> *Achtung:* Was der Auswertungsalgorithmus dabei nicht festlegt, ist, in welcher Reihenfolge die Teilausdrücke $B, A_1, \ldots, A_n$ ausgewertet werden, z.B. in der Reihenfolge des Aufschreibens von links nach rechts oder von rechts nach links oder anders.

# Beispiel Reihenfolge

- Im Falle des Ausdrucks `quadrat(quadrat(2))` kann diese Auswertungsreihenfolge so wiedergegeben werden:

```
quadrat(quadrat(2))
quadrat(2*2)
quadrat(4)
4*4
16
```

! Diese Reihenfolge ist aber auch möglich:

```
quadrat(quadrat(2))
quadrat(2)*quadrat(2)
(2*2)*(2*2)
4*4
16
```



# Auswertung in applikativer Reihenfolge

- Die Auswertung entsprechend dem vorherigen Auswertungsalgorithmus (Parameterauswertung vor Funktionsanwendung) hat die folgenden Namen:
  - Auswertung in applikativer Reihenfolge
  - Inside-out-Auswertung
  - Call-by-value-Auswertung
  - Strikte Auswertung

# Call by value – Call by reference

- Die Bezeichnung Call-by-value ist unter SML-Experten verbreitet.
- Für andere Programmiersprachen (wie Pascal und Modula) ist dieselbe Bezeichnung für eine andere Bedeutung gebräuchlich:
  - „call by value“ ist eine Form des Prozeduraufrufes, bei der die Werte der Aufrufparameter an die Prozeduren weitergegeben werden, nicht deren Speicheradresse.
  - Der Prozeduraufruf verändert die Parameter nicht.
  - Als Alternative von „call by value“ kennen Programmiersprachen wie Pascal das „call by reference“, bei dem die Speicheradressen, d.h. die Referenzen, der Aufrufparameter weitergegeben werden.



# Auswertung in normaler Reihenfolge

- Die Auswertung in anderer Reihenfolge (Funktionsanwendung vor Parameterauswertung) hat folgende Namen:
  - Auswertung in normaler Reihenfolge
  - Outside-in-Auswertung
  - Call-by-name-Auswertung

# Call by name

- Die Bezeichnung Call-by-name ist unter SML-Experten verbreitet.
- Andere Programmiersprachen gebrauchen dieselbe Bezeichnung für eine andere Bedeutung:
  - „call by name“ bezeichnet die Weitergabe eines Bezeichners ohne Festlegung eines Wertes oder einer Speicheradresse.
  - Es ist eine rein textuelle Zeichenfolgenersetzung.



# Applikative und normale Reihenfolge bei Nichtterminierung

- Die Auswertung in applikativer und in normaler Reihenfolge von einem Ausdruck liefert dasselbe Ergebnis, wenn alle Funktionsanwendungen des Ausdrucks terminieren.
- Bei nichtterminierenden Funktionsanwendungen muss das nicht sein.
- `null` sei eine einstellige Funktion, die für jede ganze Zahl den Wert `0` liefert; `f` sei eine einstellige nichtterminierende Funktion:

```
fun null(x: int)= 0;
fun f(x: int) = f(x+1);
```

- Die Auswertung in normaler Reihenfolge von `null(f(1))` liefert den Wert `0`.
- Die Auswertung in applikativer Reihenfolge terminiert nicht.

# Vorteil der applikativen Reihenfolge gegenüber der normalen

- An dem vorherigen Beispiel `quadrat(quadrat(2))` erkennt man einen Vorteil der Auswertung in applikativer Reihenfolge gegenüber der Auswertung in normaler Reihenfolge:

> Ein Parameter (oder Operand) wird bei einer Auswertung in applikativer Reihenfolge nur einmal ausgewertet, bei einer Auswertung in normaler Reihenfolge jedoch unter Umständen mehrmals.



# Überblick

1. Auswertung von Ausdrücken
2. Auswertung in applikativer und normaler Reihenfolge
3. Verzögerte Auswertung
4. Auswertung von Sonderausdrücken
5. Funktionale Variablen versus Zustandsvariablen
6. Funktionale Programmierung versus Imperative Programmierung

# Beispielauswertung in unterschiedlicher Reihenfolge

- Die einstellige konstante Funktion `null`, die zu jeder ganzen Zahl den Wert 0 liefert, kann in SML so deklariert werden:

```
fun null(x: int) = 0;
```

- Im Folgenden soll die Auswertung des Ausdrucks `null(quadrat(quadrat(quadrat(2))))` in
  - applikativer
  - und normaler

  Reihenfolge ausgewertet werden.



# Auswertung in applikativer Reihenfolge

```
null(quadrat(quadrat(quadrat(2))))
null(quadrat(quadrat(2*2)))
null(quadrat(quadrat(4)))
null(quadrat(4*4))
null(quadrat(16))
null(16*16)
null(256)
0
```

# Auswertung in normaler Reihenfolge

```
null(quadrat(quadrat(quadrat(2))))
0
```

- Wenn alle Parameter von einer Funktion „verwendet“ werden, ist die Auswertung der Parameter (oder Operanden) vor der Funktionsanwendung von Vorteil.
- Sonst führt diese Auswertungsreihenfolge (applikative Reihenfolge) zu nutzlosen Berechnungen.

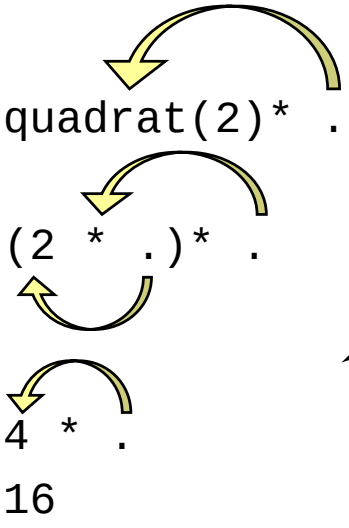


# Verzögerte Auswertung

- Die verzögerte Auswertung (*lazy evaluation*), auch *Call-by-need*-Auswertung genannt, bringt Aspekte der Auswertung in applikativer Reihenfolge und normaler Reihenfolge zusammen.
- Die Grundidee der verzögerten Auswertung ist:
  - Sie führt Funktionsanwendung vor der Auswertung der Parameter (oder Operanden) durch (vgl. Auswertung in normaler Reihenfolge).
  - Parameter werden aber nicht mehrmals ausgewertet, da bei der Funktionsanwendung alle bis auf ein Vorkommen eines Parameters durch einen Verweis auf ein einziges dieser Vorkommen ersetzt werden.

# Beispiel 1 zur verzögerten Auswertung

Betrachte die verzögerte Auswertung des folgenden Ausdrucks:
`quadrat(quadrat(2))`




Dank des Verweises gilt der Wert 4 für beide Operanden der letzten Multiplikation, sobald das innere Produkt 2*2 berechnet wird.



# Beispiel 2 zur verzögerten Auswertung

```
null(quadrat(quadrat(quadrat(2))))
0
```

- Die verzögerte Auswertung vermeidet:
  - die mehrfache Auswertung von Parametern (vgl. Auswertung in applikativer Reihenfolge).
  - die Auswertung von Parametern, die zur Auswertung einer Funktionsanwendung nicht notwendig sind (vgl. Auswertung in normaler Reihenfolge).
- Die verzögerte Auswertung hat also die Vorteile der beiden Grundansätze und keinen ihrer Nachteile.

# Verweis

- Was ist ein Verweis (auch Zeiger, Pointer oder Referenz genannt)?
- Oft wird gesagt: eine Adresse.
- Der Verwendung von Verweisen liegt ein Berechnungs-modell zugrunde:

  Bei der Auswertung (von Ausdrücken) wird der Ausdruck, auf den ein Verweis zeigt, an der Stelle des Ursprungs des Verweises eingefügt.



# Auswertungsreihenfolge von SML

- SML verwendet die Auswertung in applikativer Reihenfolge, da die verzögerte Auswertung folgende Nachteile hat:
  - Ziemlich komplizierte (also zeitaufwendige) Verwaltung von Verweisen.
  - Imperative (oder prozedurale) Befehle wie Schreibbefehle werden nur schwer zugelassen, weil der Zeitpunkt der Ausführung solcher Befehle für den Programmierer schwer vorhersehbar ist.
  - Höherer Speicherplatzbedarf (in manchen Fällen) als bei der Auswertung in applikativer Reihenfolge.

# Ausblick in die Forschung

- Effiziente Implementierungsmöglichkeiten der verzögerten Auswertung werden seit einigen Jahren mit hochinteressanten Ergebnissen untersucht.
- Auf dem Gebiet wird intensiv geforscht – siehe u.a.:
  - die Forschung um die Programmiersprache Haskell (http://www.haskell.org/)
  - das Buch Simon L. Peyton Jones: The implementation of functional programming languages, Prentice-Hall, ISBN 0-13-453333-X, ISBN 0-13-453325-9 Paperback, 1987).



# Überblick

1. Auswertung von Ausdrücken
2. Auswertung in applikativer und normaler Reihenfolge
3. Verzögerte Auswertung
4. Auswertung von Sonderausdrücken
5. Funktionale Variablen versus Zustandsvariablen
6. Funktionale Programmierung versus Imperative Programmierung

# Wertdeklarationen (`val und fun`)(1)

- Die Auswertung einer Deklaration wie `val N=A` fügt die Gleichung `N=A` zu der Umgebung hinzu.
- Als Spezialfall davon:
  - Die Auswertung einer Deklaration `val N = fn P=>A` fügt die Gleichung `N = fn P=>A` zu der Umgebung hinzu.
  - Eine Deklaration `val rec N = fn P=>A` wird ebenso behandelt, aber mit zusätzlichen Vorkehrungen, damit die Umgebung von `A` auch diese Gleichung für `N` enthält.



# Wertdeklarationen (`val und fun`)(2)

- Die Auswertung einer Deklaration `fun N P=A` fügt die Gleichung `N = fn P=>A` zu der Umgebung hinzu, mit den gleichen Vorkehrungen wie bei `val rec N = fn P=>A`.
- Wird eine Gleichung zu der Umgebung hinzugefügt, so wird sie auf Korrektheit überprüft:
  `val zwei=zwei` wird z.B. abgelehnt
- Gleichungen aus Deklarationen werden in der Praxis in einen anderen Formalismus übersetzt, bevor sie der Umgebung hinzugefügt werden.

# `if-then-else`

- Die Schreibweise von `if A1 then A2 else A3` entspricht nicht der üblichen Schreibweise von Funktionen.
- Bei der besonderen Schreibweise von `if-then-else-`Ausdrücken handelt es sich um „syntaktischen Zucker“.
- „`if A1 then A2 else A3`“ lässt sich genauso gut in Präfixschreibweise „`if_then_else(A1,A2,A3)`“ schreiben.

Das ist keinen korrekte SML-Syntax!



# Auswertung von `if-then-else` als funktionaler Ausdruck

- Nach dem Prinzip des Occam´schen Messers sollten `if-then-else-`Ausdrücke ähnlich wie herkömmliche Funktionsanwendungen ausgewertet werden.
- Zur Auswertung eines Ausdrucks „`if A1 then A2 else A3`“ müssten zunächst alle drei Teilausdrücke `A1,A2 und A3` ausgewertet werden (da SML auf der Auswertung in applikativer Reihenfolge beruht).

# Beispiel Fakultätsfunktion (1)

- Dieser Ansatz funktioniert nicht, wie dieses Gegenbeispiel zeigt:

```
n! = 1        falls n=0
n*(n-1)!      andernfalls
```

- Die Fakultätsfunktion ist auf den natürlichen Zahlen definiert, kann also als eine partielle Funktion auf den ganzen Zahlen gesehen werden.

- In SML kann die Fakultätsfunktion so implementiert werden:

```
fun fak(n)= if n=0 then 1 else n*fak(n-1);
fun fak(n)= if_then_else(n=0,1,n*fak(n-1));
```

(Präfixschreibweise)



# Beispiel Fakultätsfunktion (2)

- Für gewisse Argumente liefert die Funktion keinen Wert, ist also nicht total:

  `fak(0)` hat z.B. den Wert `1`,

  `fak(~1)` hat keinen Wert, da die Auswertung nicht terminiert.

- Würden zunächst die Teilausdrücke `A1,A2` und `A3` ausgewertet (so wie funktionale Ausdrücke in applikativer Reihenfolge), so müsste zur Auswertung von `fak(0)` der Ausdruck `fak(~1)` ausgewertet werden.
- Diese Auswertung terminiert nicht, obwohl `fak(0)` den Wert `1` hat.
- So macht die Auswertung keinen Sinn.

# Beispiel Fakultätsfunktion (3)

- Auswertung von `fak(0)` nach dem Substiutionsmodell:

```
fak(0)
if_then_else(0=0,1,0*fak(0-1))
if_then_else(true,1,0*fak(~1))
if_then_else(true,1,0*if_then_else(~1=0,1,~1*fak(~1-
1)))
if_then_else(true,1,0*if_then_else(false,1,~1*fak(~2)))
if_then_else(true,1,0*if_then_else
(false,1,~1*if_then_else(…)))
…
```



# Der Sonderausdruck `if-then-else`

- Bei einer Auswertung in applikativer Reihenfolge können keine rekursiven Definitionen ausgewertet werden, da zunächst immer alle darin vorkommenden Teilausdrücke ausgewertet werden müssen.
- Also müssen bestimmte Ausdrücke anders ausgewertet werden: *Sonderausdrücke.*
- Jede Programmiersprache mit Auswertung in applikativer Reihenfolge enthält Sonderausdrücke.

# Auswertung von `if-then-else` als Sonderausdruck (1)

- Das besondere Verfahren zur Auswertung eines Sonderausdrucks der Form:
  „`if A1 then A2 else A3`“ oder
  „`if_then_else(A1,A2,A3)`“ lautet:

> Werte von den drei Teilausdrücken `A1,A2,A3` zuerst nur `A1` aus. Hat `A1` den Wert `true`, dann (und nur dann) wird `A2` ausgewertet (und `A3` wird nicht ausgewertet). Der Wert von `A2` wird als Wert des gesamten Ausdrucks geliefert.
> Hat `A1` den Wert `false`, dann (und nur dann) wird `A3` ausgewertet (und `A2` wird nicht ausgewertet). Der Wert von `A3` wird als Wert des gesamten Ausdrucks geliefert.



# Auswertung von `if-then-else` als Sonderausdruck (2)

- Diese Spezifikation ist ein sehr präziser (und sehr einfacher) Algorithmus.
- SML wertet `if-then-else`-Ausdrücke in dieser Weise aus, da rekursive Definitionen sonst nicht möglich wären (obwohl sie wie im Fall der Fakultätsfunktion oft wünschenswert sind).

# Pattern Matching (1)

- Im Kapitel 2 haben wir das „Pattern Matching“ im Zusammenhang mit Deklarationen kennen gelernt:

```
val Ziffer =      fn     0 => true
                  |      1 => true
                  |      2 => true
                  |      3 => true
                  |      4 => true
                  |      5 => true
                  |      6 => true
                  |      7 => true
                  |      8 => true
                  |      9 => true
                  |      _ => false;
(fn true => E1    | false => E2)
```



# Pattern Matching (2)

- SML bietet das folgende Konstrukt an, das `if_then_else` in gewissem Sinn verallgemeinert und wie `if-then-else-`Ausdrücke in anderen Kontexten als Deklarationen verwendet werden kann:

```
case A     of    A1 => B1
              |    A2 => B2
                ...
              |    An => Bn
```

# case (1)

- Die Auswertung eines solchen Ausdrucks funktioniert so:
  - Zunächst wird nur der Ausdruck `A` ausgewertet.
  - Der Wert von `A` wird nacheinander mit den Mustern `A1,…,An` „gematcht“.
  - Ist `Ai` das erste Muster (*pattern*), das mit dem Wert von `A` „matcht“, so wird `Bi` ausgewertet und dessen Wert als Wert des `case`-Ausdrucks geliefert.
- Das Pattern Matching in Deklarationen wird ähnlich ausgewertet.

> ! Hier wird der Begriff „Pattern Matching“ nicht präzise beschrieben, es soll aber zunächst bei dieser intuitiven Erklärung belassen werden.



# case (2)

- Das `case`-Konstrukt kann als Grundlage zur Auswertung von anderen Sonderausdrücken dienen.
- z.B. kann ein Ausdruck „`if A1 then A2 else A3`“ auch so ausgewertet werden:
  - Werte von den drei Teilausdrücken `A1,A2,A3` zunächst gar keinen aus.
  - Konstruiere einen neuen Ausdruck:

    ```
    case A1 of true => A2   | false => A3
    ```

  - Werte diesen neuen Ausdruck aus und liefere seinen Wert als Wert des `if-then-else`-Ausdrucks.

# Zurückführen von Sonderausdrücken

- Andere Sonderausdrücke können in ähnlicher Weise auf `case` zurückgeführt werden.
- Dieses Zurückführen von Sonderausdrücken auf wenige andere Sonderausdrücke ist ein Beispiel für die Anwendung des Occam´schen Messers.
- In der Praxis werden Sonderausdrücke allerdings seltener durch Zurückführen auf andere Sonderausdrücke behandelt, sondern meistens durch maßgeschneiderte Auswertungsalgorithmen.



# Die Boole´schen Operatoren `andalso` und `orelse`

- Boole´sche Konjunktionen $A_1 \wedge A_2$ und Disjunktionen $A_1 \vee A_2$ können prinzipiell auf zwei verschiedenen Weisen ausgewertet werden:
  - Beide Ausdrücke werden ausgewertet, dann mit einer Tabelle verglichen.
  - Erst wird der eine Ausdruck ausgewertet u.U. dann der andere.

# 1. Ansatz

1. Die Teilausdrücke $A_1$ und $A_2$ werden zunächst ausgewertet und der Wert der Konjunktion oder Disjunktion wird entsprechend der folgenden Wahrheitstabelle aus den Werten von $A_1$ und von $A_2$ ermittelt:

| $A_1$ | $A_2$ | $A_1 \wedge A_2$ | $A_1 \vee A_2$ |
|---|---|---|---|
| `true` | `true` | `true` | `true` |
| `true` | `false` | `false` | `true` |
| `false` | `true` | `false` | `true` |
| `false` | `false` | `false` | `false` |



# 2. Ansatz

2. Fall „$\wedge$“:
   Zunächst wird nur der erste Teilausdruck $A_1$ ausgewertet.
   Ist der Wert von $A_1$ `true`, so wird auch $A_2$ ausgewertet und dessen Wert als Wert des Ausdrucks $A_1 \wedge A_2$ geliefert.
   Ist der Wert von $A_1$ `false`, so wird `false` als Wert des Ausdrucks $A_1 \wedge A_2$ geliefert (und $A_2$ wird nicht ausgewertet).

   Fall „$\vee$“:
   Zunächst wird nur der erste Teilausdruck $A_1$ ausgewertet.
   Ist der Wert von $A_1$ `true`, so wird `true` als Wert des Ausdrucks $A_1 \vee A_2$ geliefert (und $A_2$ wird nicht ausgewertet).
   Ist der Wert von $A_1$ `false`, so wird $A_2$ ausgewertet und dessen Wert als Wert des Ausdrucks $A_1 \vee A_2$ geliefert.

# Zurückführung des 2. Ansatz

- Den 2. Ansatz kann man wieder durch Zurückführen auf andere Sonderausdrücke realisieren.
- Zur Auswertung eines Ausdrucks der Form $A_1 \wedge A_2$ könnte man so vorgehen:

> Werte von den Teilausdrücken $A_1$,$A_2$ zunächst gar keinen aus. Konstruiere einen neuen Ausdruck
>
> `if` $A_1$ `then` $A_2$ `else false`
>
> Werte diesen neuen Ausdruck aus und liefere seinen Wert als Wert des ursprünglichen Ausdrucks.



# Auswertung von `andalso` und `orelse` in SML

- Auf welche der beiden möglichen Weisen sollten Kon- und Disjunktionen am besten ausgewertet werden?
- Verschiedene Programmiersprachen beantworten diese Frage verschieden.
- SML wählt wie viele andere Programmiersprachen auch den 2. Ansatz.

Die SML-Bezeichnung „`andalso`“ und „`orelse`“ statt „`and`“ und „`or`“ sollen unterstreichen, dass diese Operatoren nach dem 2. Ansatz ausgewertet werden.

# Auswertung von `andalso` in anderen Sprachen

- Wie kann man erkennen, wie eine Programmiersprache die Boole´sche Konjunktion auswertet?
- Dazu reicht ein Aufruf wie dieser (hier in SML):

```
fun endlose_berechnung(n):bool =
    endlose_berechnung(n+1);
false andalso endlose_berechnung(0);
```

→ Terminiert er, dann wertet die Programmiersprache die Konjunktion nach dem 2. Ansatz aus, andernfalls nach dem 1. Ansatz.



# Infixoperator-Deklarationen und Präzedenzen (1)

- SML bietet die Möglichkeit so genannter Infixoperator-Deklarationen:

```
infix /\;
fun A1 /\ A2 = if A1 then A2 else false;

infix \/;
fun A1 \/ A2 = if A1 then true else A2;
```

# Infixoperator-Deklarationen und Präzedenzen (2)

- Eine Infixoperator-Deklaration kann mit einer Präzedenz von `0` bis `9` ergänzt werden: z.B. `infix 6 /\;`
- Je größer die Präzedenz, desto stärker bindet der Operator.
- Fehlt die Präzedenz, dann gilt der voreingestellte (*default*) Wert `0`.
- Eine Infixoperator-Deklaration:
  - `infix n Op;` definiert den Operator `Op` als linksassoziativ.
  - `infixr n Op;` definiert den Operator `Op` als rechtsassoziativ.



# Beispiel `potenz´`

- Die Funktion `potenz´` aus Abschnitt 2.11 kann als rechtsassoziativer Infixoperator deklariert werden:

```
infixr 8 **;
fun a ** b =    if b = 0 then 1
                else  if gerade(b)
                      then quadrat(a**(b div 2))
                      else a*(a**(b-1));
```

# Der erweiterte Auswertungsalgorithmus (1)

- Bisher wurde für einige Klassen von Sonderausdrücken, jeweils ein Verfahren zu deren Auswertung beschrieben.
- Wenn wir alle diese Verfahren mit dem Sammelbegriff „Sonderalgorithmus“ bezeichnen und annehmen, dass die Sonderalgorithmen einfach die Werte der Bezeichner `if-then-else` usw. sind, kann der besprochene Algorithmus wie folgt um eine Behandlung von Sonderausdrücken erweitert werden:



# Der erweiterte Auswertungsalgorithmus (2)

- Zur Auswertung eines Ausdrucks A gehe wie folgt vor:

1. Falls A atomar ist, dann:
   a) Falls A vordefiniert ist, liefere den vordefinierten Wert von A (dieser kann auch ein Sonderalgorithmus sein).
   b) Andernfalls (der Wert von A ist in der Umgebung definiert) sei `A = W` die Gleichung in der Umgebung, die den Wert von A definiert. Liefere W als Wert von A (wobei W auch eine Funktion $fn(F_1,\ldots,F_k)=>R$ sein kann).

2. Andernfalls (A ist zusammengesetzt) hat A die Form $B(A_1,\ldots,A_n)$ mit $n \geq 0$. Werte den Teilausdruck B aus.
   a) Falls der Wert von B ein Sonderalgorithmus ist, dann:
   Wende ihn auf die Teilausdrücke $A_1,\ldots,A_n$ an. Liefere den dadurch erhaltenen Wert als Wert von A.

# Der erweiterte Auswertungsalgorithmus (3)

b) Andernfalls (der Wert von B ist kein Sonderalgorithmus), werte die Teilausdrücke $A_1, \ldots, A_n$ aus. Seien $W_1, \ldots, W_n$ die Werte der Teilausdrücke $A_1, \ldots, A_n$.

   i. Falls der Wert von B eine Systemfunktion ist, dann:

   Wende sie auf $(W_1, \ldots W_n)$ an. Liefere den dadurch erhaltenen Wert als Wert von A.

   ii. Falls der Wert von B eine Funktion der Form $fn(F_1, \ldots, F_n) => R$ ist, dann:

   Werte R in der erweiterten Umgebung aus, die aus der aktuellen Umgebung und den zusätzlichen Gleichungen $F_1=W_1, \ldots, F_n=W_n$ besteht. Liefere den dadurch erhaltenen Wert als Wert von A (die Umgebung ist nun wieder die ursprüngliche).



# Überblick

1. Auswertung von Ausdrücken
2. Auswertung in applikativer und normaler Reihenfolge
3. Verzögerte Auswertung
4. Auswertung von Sonderausdrücken
5. Funktionale Variablen versus Zustandsvariablen
6. Funktionale Programmierung versus Imperative Programmierung

# Überblick über Variablen

- Es gibt 2 Arten von Variablen:
  - Funktionale Variablen
  - Zustandsvariablen
- funktionale Variablen werden auch:
  - „logische Variablen“
  - oder kurz „Variablen“

  genannt, wenn der Kontext eindeutig macht, welche Art von Variablen gemeint ist.
- SML bietet sowohl funktionale als auch Zustandsvariablen an.

Alle bisher verwendeten Variablen waren funktionale Variablen.



# Funktionale Variablen (1)

- Hauptmerkmal der funktionalen Variable:

  Die Bindung eines Wertes an eine funktionale Variable kann nicht verändert werden.

- Man kann eine funktionale Variable als eine benannte Speicherzelle betrachten (oder einen benannten Speicherbereich), die zur Speicherung eines einzigen Wertes verwendet wird.

# Funktionale Variablen (2)

- Hier sieht man z.B. drei verschiedene solcher Speicherzellen:

```
-val z = 2;
val z = 2 : int

-fun quadrat(x : int) = x*x;

-val z = 5;
val z = 5 : int
```

- Die zweite Deklaration der Variable `z` ist ein Fall von „Wiederdeklaration eines Namens“ (oder „Wiederdeklaration einer Variable“) (vgl. Abschnitt 2.7).
- Dabei bleibt die erste Speicherzelle `z`, die den Wert `2` hat, erhalten und eine zweite Speicherzelle `z` bekommt den Wert `5`.



# Überschatten

- Namenskonflikte nach Wiederdeklarationen einer Variable werden durch eine einfache Regel gelöst, deren Wirkung auf Variablen „Überschatten“ heißt:

> Nach einer Wiederdeklaration einer Variablen mit Namen `N` gilt der Name `N` nur noch für die zuletzt deklarierte Variable mit Namen `N`.

- Liegt der Programmiersprache eine statische (oder lexikalische) Bindung zu Grunde (vgl. 2.7), dann sind Deklarationen, in deren definierenden Teilen der Namen `N` vorkommt und die vor der Wiederdeklaration ausgewertet wurden, von der Wiederdeklaration der Variable mit Namen `N` unbeeinflusst.

# Referenztransparent

- Überschatten und die statische (oder lexikalische) Bindung zusammen machen eine funktionale Programmiersprache „referenztransparent“:

> Der Wert eines Ausdrucks (dieser Sprache) wird nicht verändert, wenn „das Gleiche durch das Gleiche“, d.h. ein Teilausdruck durch seine Definition, ersetzt wird.

- Eine Sprache ist genau dann referenztransparent, wenn syntaktisch gleiche Ausdrücke in der gleichen Umgebung auch die gleichen Werte haben.



# Zustandsvariablen

- Grundidee einer Zustandvariable:

> Der Inhalt einer benannten Speicherzelle (oder eines Speicherbereichs) ist veränderbar.

- Vgl. eine Zustandsvariable mit einer etikettierten Schublade:

  Der Name der Variable entspricht dem Etikett, der Schubladeninhalt lässt sich verändern.

# Anwendungsbeispiel Bankkonto

- Eine Zustandsvariable `kto` habe als Inhalt den Wert `100` (Euro).
- Operationen ermöglichen, z.B.:
  - den (Zu-)Stand des Kontos `kto` um `25` Euro auf `125` Euro zu erhöhen (`einzahlen 25`),
  - um `55` Euro auf `70` Euro zu verringern (`abheben 55`).

Mit funktionalen Variablen ist eine solche Veränderung von Inhalten nicht möglich.

→ Zustandsvariabeln, d.h. Variablen mit veränderbaren Inhalten, sind zur Modellierung gewisser Rechenvorgänge nützlich und nötig.



# Zustand, Zustandsänderung und Zuweisung

- Da Zustandsvariablen veränderbar sind, hängen ihre Inhalte vom Zeitpunkt ab, zu dem sie ermittelt werden.
- Ein „*Zustand*“ eines Systems, das auf Zustandsvariablen beruht, ist eine Menge von Bindungen (Variablenname, Variableninhalt), die zu einem Zeitpunkt gelten.
- Eine „*Zustandsänderung*“ ist eine Veränderung des Inhaltes einer Zustandsvariablen, also auch ihres Zustandes.
- Zustandsänderungen erfordern eine Operation, mit der der Inhalt einer Zustandsvariablen verändert werden kann.
- Diese Operation heißt üblicherweise „*Zuweisung*“.
- In vielen Programmiersprachen (wie SML) wird sie „:=“ notiert.

# Zwei Arten von Zustandsvariablen

- Operationen auf Zustandsvariablen können auf zwei verschiedene Weisen ausgedrückt werden:
  - „Zustandsvariablen mit expliziter Dereferenzierung“
  - „Zustandsvariablen ohne explizite Dereferenzierung“

Zustandsvariablen mit expliziter Dereferenzierung werden auch „*Referenzen*“ genannt.



# Referenzen und Dereferenzierung

- Bei „*Referenzen*“ steht der Variablenname stets für eine (symbolische) Speicheradresse (Schubladenetikett).
- Man braucht ein Sprachkonstrukt, um den Inhalt einer Schublade zu bezeichnen.
- In SML:
  - Zustandsvariablen sind Referenzen
  - Der Operator „!“ ist das Sprachkonstrukt

Bsp.: Ist v eine SML Referenz, so bezeichnet !v den Inhalt der Referenz.

- Die Operation, die in SML mit „!“ ausgedrückt wird, heißt „*Dereferenzierung*“.

# Beispiel `kto` (1)

- In SML wird `kto` mit anfänglichem Wert `100` so deklariert:
```
- val kto = ref 100;
```
- Den Kontostand kann man so erhalten:
```
- !kto;
val it = 100 : int
```
- *Achtung:* Es muss zwischen den Bezeichnungen „*Wert*“ und „*Inhalt*“ unterschieden werden:
  - Der Ausdruck `!kto` hat den Wert `100`.
  - Der Ausdruck `kto` dagegen hat nicht den Wert `100`.
  - Der Wert von `kto` ist das Etikett der Schublade, der Wert von `!kto` ist das, was in der Schublade drin ist.
- Das sieht man auch an den Typen:
  - `!kto` hat den Typ `int`
  - `kto` hat den Typ `int ref`



# Beispiel `kto` (2)

- Mit dem Dereferenzierungsoperator „`!`“ und der Zuweisung „`:=`“ (ausgesprochen: *becomes*) wird der Konto(zu)stand so verändert:

```
-kto := !kto + 25;
val it = () : unit

-!kto;
val it = 125 : int

-kto := !kto - 55;
val it = () : unit

-!kto;
val it = 70 : int
```

# Sequenzierung

- Der Zeitpunkt einer Zustandsveränderung ist wichtig für den Zustand. Also muss die Reihenfolge von Zustandsveränderungen festgelegt werden.
- Das Konstrukt „ `;` “ drückt in SML eine Reihenfolge oder „*Sequenz*“ aus.
- Der Ausdruck `(A1;A2)` wird so ausgewertet:
  - Zuerst wird `A1` ausgewertet, sein Wert wird ignoriert.
  - Danach wird `A2` ausgewertet und sein Wert wird als Wert des Ausdrucks `(A1,A2)` geliefert.
- Zwischen den Klammern können beliebig viele durch „ `;` “ getrennte Ausdrücke stehen:
  - Sie werden in der gegebenen Reihenfolge ausgewertet.
  - Nur der Wert des letzten wird zurückgeliefert.



# Zustandsvariablen ohne explizite Dereferenzierung

- Es gibt Zustandvariablen, die keine explizite Dereferenzierung brauchen.
- Je nach Kontext, in dem ein Variablenname `v` vorkommt, steht er für eine Referenz oder für den Wert, der Inhalt der Speicheradresse `v` ist.
- Sei `kto` eine Zustandsvariable ohne explizite Dereferenzierung.
- In vielen Programmiersprachen wird der Konto(zu)stand dann so verändert:

```
kto := kto + 25;
kto := kto - 55;
```

Links von „ `:=` “ bezeichnet `kto` eine Speicheradresse, rechts einen Wert. Die Dereferenzierung auf der rechten Seite ist implizit.

# Gleichheit von Zustandsvariablen mit expliziter Dereferenzierung

- Für Referenzen bedeutet Gleichheit die Gleichheit der Referenzen (→ Gleichheit der Etiketten).
- Beispiel:

```
- val v = ref 5;
val v = ref 5 : int ref

- val w = ref 5;
val w = ref 5 : int ref

- v = w;
val it = false : bool

- !v = !w;
val it = true : bool
```



# Gleichheit von Zustandsvariablen ohne explizite Dereferenzierung

- Für Zustandsvariablen ohne explizite Dereferenzierung bezieht sich die Gleichheit auf die Inhalte.
- Beispiel (in einer Phantasiesprache):

```
- declare v = pointer to 5;
v = pointer to 5 : pointer to int

- declare w = pointer to 5;
w = pointer to 5 : pointer to int

- v = w;
val it = true : bool
```

Viele Sprachen mit expliziter Dereferenzierung verwenden die Bezeichnung „`pointer`“ („Zeiger“) für (symbolische) Speicheradressen.

# Alias-Problem

- Referenzen und Zeiger ermöglichen, dieselbe Zustandsvariable in verschiedener Weise zu benennen. In SML z.B.:

```
- val w = ref 5;
val w = ref 5 : int ref

- val z = w;
val z = ref 5 : int ref
```

- Was mit funktionalen Variablen kein Problem ist, kann mit Zustandsvariabeln zur Unübersichtlichkeit führen :

```
- w := 0;
val it = () : unit

- w;
val it = ref 0 : int ref
```

Da `z` dieselbe Referenz wie `w` bezeichnet, verändert die Zuweisung `w := 0` nicht nur `w`, sondern auch `z` (genauer: den Inhalt von `z`).



# Zyklische Referenzierung (1)

- Betrachte:

```
- val a = ref (fn(x : int) => 0);
val a = ref fn : (int -> int) ref

- !a(12);
val it = 0 : int
```

`!a` ist die Funktion, die jede ganze Zahl auf `0` abbildet.

# Zyklische Referenzierung (2)

- Betrachte außerdem:

```
- val b = ref (fn(x : int) =>
           if x=0 then 1 else x*!a(x-1));

- !b(0);
val it = 1 : int

- !b(3);
val it = 0 : int

- a := !b;
val it = () : unit

- !b(0);
val it = 1 : int

- !b(3);
val it = 6 : int
```

`!b` ist eine Funktion, die `0` auf `1` und jede andere ganze Zahl auf `0` abbildet.



# Zyklische Referenzierung (3)

- Nach der Zuweisung `a := !b` ist `!b` die rekursive Fakultätsfunktion (total auf den natürlichen Zahlen).
- In Zusammenhang mit komplexen Datenstrukturen (Kapitel 8) können zyklische Referenzierungen sehr nützlich sein.

# Zustandsvariablen in SML: Referenzen (1)

- Ist `v` eine SML-Referenz eines Objekts vom Typ `t`, so ist `t ref` der Typ von `v`.
- Der Referenzierungsoperator von SML ist „`ref`“:

```
- val v = ref 5;
val v = ref 5 : int ref

- val w = ref (fn(x : int) => 0);
val w = ref fn : (int -> int) ref
```

- Zusammen mit dem Dereferenzierungsoperator „`!`“lässt sich der folgende Ausdruck bilden und auswerten:

```
- !w(!v);
val it = 0 : int
```



# Zustandsvariablen in SML: Referenzen (2)

- Der Zuweisungsoperator von SML ist „`:=`“.
- Der Wert einer Zuweisung ist „`()`“ (*unity*).
- SML liefert Referenzen, druckt sie aber nicht aus. Anstelle einer Referenz druckt SML das Symbol `ref` gefolgt von ihrem Inhalt:

```
- val v = ref 5;
val v = ref 5 : int ref

- v;
val it = ref 5 : int ref

- val w = ref (fn(x : int) => 0);
val w = ref fn : (int -> int) ref

- w;
val it = ref fn : (int -> int) ref
```

# Zustandsvariablen in SML: Referenzen (3)

- Die Referenz einer Referenz ist selbstverständlich möglich:

```
- val w = ref (fn(x : int) => 0);
val w = ref fn : (int -> int) ref

- val z = ref w;
val z = ref (ref fn) : (int -> int) ref ref

- !(!z)(9);
val it = 0 : int
```

- Die Gleichheit für Referenzen in SML ist „ = “.



# Überblick

1. Auswertung von Ausdrücken
2. Auswertung in applikativer und normaler Reihenfolge
3. Verzögerte Auswertung
4. Auswertung von Sonderausdrücken
5. Funktionale Variablen versus Zustandsvariablen
6. Funktionale Programmierung versus Imperative Programmierung

# Funktionale Programmierung vs. Imperative Programmierung

- Die Programmierung mit Zustandsvariablen nennt man imperative Programmierung.
- Die imperative Programmierung stellt ein Programmierparadigma dar, das sich wesentlich von der funktionalen Programmierung unterscheidet.
- Programmiersprachen wie SML bieten die Möglichkeit, beide Programmierparadigmen zu vermischen.
- Das Hauptparadigma solcher Sprachen ist immer nur eines von beiden!



# Überschatten versus Zustandsänderung

- *Überschatten* und *Zustandsänderungen* sind zwei unterschiedliche Techniken.
- Überschatten verändert den Wert einer bereits existierenden Variable nicht, sondern verwendet deren Name für eine neue Speicherzelle.
- Eine Zustandsänderung erstellt keine neue Variable, sondern verändert den Inhalt einer bereits vorhandenen Speicherzelle.

# Funktion versus Prozedur (1)

- Eine *Funktion* kann auf Argumente angewandt werden und liefert einen Wert als Ergebnis, ohne dabei Zustandsveränderungen zu verursachen. Eine Funktion ist also referenztransparent.
- Die Bezeichnung *Prozedur* ist ein Oberbegriff, der sowohl Funktionen umfasst als auch Programme, die Zustandsveränderungen verursachen.
- Eine Prozedur, die keine Funktion ist, die also Zustands-veränderungen verursacht, liefert je nach Programmiersprache entweder überhaupt keinen Wert als Ergebnis oder einen uninteressanten Wert wie `()` in SML.



# Funktion versus Prozedur (2)

- In vielen Sprachen kann man Prozeduren definieren, die:
  - keine Funktionen sind, weil sie Zustandsveränderungen verursachen,
  - aber auch Werte liefern, als seien sie Funktionen.

→ Schlechter Programmierstil!

- Man stelle sich ein Programm `ln` vor, das den Logarithmus zu einer Basis `b`=*e* berechnet (wobei *e* die Euler´sche Zahl ist), das beim Aufruf `ln(`*e*`)` nicht nur den Wert `1.0` liefert, sondern nebenbei noch den Wert der Basis `b` verändern würde!

> Prozeduren, die Werte liefern, die also in Ausdrücken an derselben Stelle wie Funktionen vorkommen können, werden häufig – fälschlich bzw. irreführend – Funktionen genannt.

# Unzulänglichkeit des Substitutionsmodells

- Zur Behandlung von Zustandsvariablen ist das Substitutionsmodell unzulänglich.
- Mit Zustandsvariablen und Zustandsveränderungen wird die Referenztransparenz durchbrochen - der Wert eines Ausdrucks hängt nun vom Auswertungszeitpunkt ab.

→ Das Substitutionsmodell ergibt keinen Sinn mehr.



# Beispiel `kto`

- Obwohl die Umgebung gleich bleibt, haben die syntaktisch gleichen Ausdrücke `!kto` und `!kto` verschiedene Werte:

```
- val kto = ref 100;
val kto = ref 100 : int ref

- kto := !kto + 50;
val it = () : unit

- kto;
val it = ref 150 : int ref

- kto := !kto + 50;
val it = () : unit

- kto;
val it = ref 200 : int ref
```

! Um Zustandsvariablen beschreiben zu können, ist ein anderes, wesentlich kompliziert-eres Berechnungsmodell nötig, als das Substitutionsmodell: Das „*Umgebungsmodell*“.

# Rein funktionale Programme und Ausdrücke

- Funktionale Programme und Ausdrücke ohne Zustandsvariablen werden „rein funktional“ genannt.
- Mit Zustandsvariablen erweist sich der Beweis von Programmeigenschaften (z.B. Terminierung) als viel schwieriger, weil zusätzlich zu den Deklarationen die Veränderungen des Programmzustands berücksichtigt werden müssen.
- Dafür müssen zeitliche Programmabläufe berücksichtigt werden, wozu so genannte „temporallogische“ Formalismen verwendet werden (siehe die Lehrveranstaltungen im Hauptstudium zu den Themen „Temporallogik“ und „Model Checking“).



# Nebeneffekte

- Zustandsveränderungen, die sich aus der Auswertung von nicht rein funktionalen Ausdrücken oder Programmen ergeben, werden „Nebenwirkungen“ oder „Nebeneffekte“ (*side effects*) genannt.

# Reihenfolge der Parameterauswertung

- Der Auswertungsalgorithmus und das Substitutionsmodell, die am Anfang dieses Kapitels eingeführt wurden, legen die Reihenfolge der Auswertung der aktuellen Parameter $A_1, \ldots, A_n$ eines Ausdrucks $B(A_1, \ldots, A_n)$ NICHT fest.
- Aktuelle Parameter können von links nach rechts (also in der Reihenfolge $A_1$ bis $A_n$) oder von rechts nach links (also von $A_n, A_{n-1}, \ldots, A_1$) ausgewertet werden.
- Bei rein funktionalen Programmen beeinflusst die Reihenfolge der Auswertung der aktuellen Parameter das Ergebnis nicht.
- Anders ist es, wenn einige der aktuellen Parameter nicht rein funktional sind, weil ihre Auswertung Nebeneffekte hat, so dass der Zustand am Ende der Parameterauswertung von der Reihenfolge dieser Auswertung abhängt.