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

Skript zur Vorlesung
Informatik I
Wintersemester 2006

Kapitel 4: Prozeduren zur Abstraktionsbildung

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

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

Database Systems

Inhalt

1. Die „Prozedur“ als Kernbegriff der Programmierung
2. Prozeduren zur Bildung von Abstraktionsbarrieren: Lokale Deklarationen
3. Prozeduren versus Prozesse
4. Ressourcenbedarf – Größenordnungen
5. Beispiel: Der größte gemeinsame Teiler

Überblick

1. Die „Prozedur“ als Kernbegriff der Programmierung
2. Prozeduren zur Bildung von Abstraktionsbarrieren: Lokale Deklarationen
3. Prozeduren versus Prozesse
4. Ressourcenbedarf – Größenordnungen
5. Beispiel: Der größte gemeinsame Teiler

Prozeduren zur Programmzerlegung

- Zur Lösung eines Problems lohnt es sich, Teilprobleme zu erkennen, zu formalisieren und getrennt vom Hauptproblem zu lösen.
- Aus diesem natürlichen Ansatz ist der Begriff „Prozedur“ entstanden, der ein Kernbegriff der Programmierung ist: *Prozeduren sind Teilprogramme, die Teilaufgaben lösen.*
- Jede Programmiersprache ermöglicht die Programmierung von Prozeduren, auch die so genannten „niederen“ Maschinensprachen.

Vorteile von Prozeduren

1. Prozeduren ermöglichen die Zerlegung eines Programms in übersichtliche Teilprogramme.
2. Prozeduren ermöglichen die Mehrfachverwendung von identischen Programmteilen an verschiedenen Stellen eines Programms.
3. Prozeduren ermöglichen die Verwendung von so genannten „lokalen Variablen“ mit präzise abgegrenzten Geltungsbereichen.
4. Prozeduren sind austauschbare Programmkomponenten.

`flaecheSchnittRechteck` (Übungsblatt 2)

```
fun schnittRechteck(
  ((untenX1,untenY1),(obenX1,obenY1)),
  ((untenX2,untenY2), (obenX2,obenY2)) ) =

  ((Real.max(untenX1,untenX2),Real.max(untenY1,untenY2)),
   (Real.min(obenX1,obenX2),Real.min(obenY1,obenY2)) );

fun flaecheRechteck( (untenX,untenY),(obenX,obenY) ) =
  if untenX <= obenX andalso untenY <= obenY
  then (obenX - untenX) * (obenY - untenY)
  else 0.0;

fun flaecheSchnittRechteck(a,b)=
  flaecheRechteck( schnittRechteck(a,b) );
```

1.Übersichtliche Teilprogramme

- Durch die Verwendung der Funktionen `schnittRechteck` und `flaecheRechteck` wird die Spezifikation der Funktion `flaecheSchnittRechteck` übersichtlicher.
- Jedes einzelne Teilprogramm kann leichter als das Gesamtprogramm spezifiziert, verfasst und auf Korrektheit überprüft werden.

2. Mehrfachverwendung von identischen Programmteilen

- Die Funktionen `Real.max` und `Real.min` werden je 2x in der Definition der Funktion `schnittRechteck` verwendet.
- Die Erkennung von mehrfach verwendbaren Teilen erleichtert sowohl die Spezifikation als auch die Implementierung und die Wartung von Programmen.

3. Verwendung von „lokalen Variablen“

- In unserem Beispiel verwenden wir zwar keine lokalen Variablen, aber im Verlauf dieses Kapitels werden diese noch ausführlich behandelt.
- Die Verwendung von lokalen Variablen als Namen für konstante Zwischenwerte in Berechnungen trägt oft dazu bei, Berechnungen verständlicher zu machen.
- Variablennamen, die in einer Prozedur lokal für konstante Werte oder für Prozeduren benutzt werden, stehen zu (anderen) Verwendungen außerhalb dieser Prozedur frei:
  - Die Prozedur, in der lokale Variablen deklariert werden, ist der „Geltungsbereich“ dieser lokalen Variablen.
  - Dadurch reicht es aus, nur über die Namen von „globale Variablen“ Buch zu führen, um Namenskonflikte zu vermeiden.
  - So wird die Erstellung von komplexer Software und die Zusammenarbeit von mehreren Personen zur Erstellung derselben Software erheblich erleichtert.

4. Austauschbare Programmkomponenten

- Wie `schnittRechteck` oder `flaecheRechteck` implementiert sind, ist für die Definition der Funktion `flaecheSchnittRechteck`, die diese Funktionen verwendet, unwichtig.
- Jede partiell korrekte bzw. total korrekte Implementierung, die der natürlich-sprachlichen Spezifikation dieser beiden Funktionen entspricht, kann verwendet werden:

  → Sie ergibt eine partiell korrekte bzw. total korrekte Implementierung von `flaecheSchnittRechteck`.
- So kann während der Programmentwicklung z.B. eine „einfache“ Implementierung von Prozeduren verwendet werden, die zu späteren Zeitpunkten durch „bessere“ Implementierungen ersetzt wird, ohne dass solche Änderungen die Korrektheit des Gesamtprogramms in Frage stellen (mehr dazu in den Hauptstudiumsvorlesungen über Software-Entwicklung).

Funktion versus Prozedur (1)

- NICHT alle Prozeduren sind Funktionen (siehe Abschnitt 3.6)!
- Funktionen liefern ein Ergebnis und haben keine Nebeneffekte.
- Prozeduren, die keine Funktionen sind, verursachen Nebeneffekte und liefern keine oder uninteressante Werte wie（）in SML.
  - Die Programmiersprache ist rein funktional (siehe Kapitel 3): dann sind alle Prozeduren Funktionen.
  - Die Programmiersprache ist nicht rein funktional: dann sind manche Prozeduren keine Funktionen.

Funktion versus Prozedur (2)

- Oft wird von Funktionen anstelle von Prozeduren gesprochen, wenn es um Prozeduren (mit Nebeneffekten) geht, die einen Wert liefern.
  → „Funktionsprozedur“
- Eigenschaften von Prozeduren sind nicht immer formal untersucht und bewiesen.
- Für Funktionen ist dafür das Substitutionsmodell seiner Einfachheit wegen hervorragend geeignet.
- Um Eigenschaften von Nebeneffekte verursachenden Prozeduren zu beweisen, sind viel kompliziertere Ansätze nötig.

Definition von Funktionen und Prozeduren in SML

- Alternative Schreibweisen zur Definition von Funktionen oder Prozeduren ohne Pattern Matching:

```
val rec fak = fn n => if n=0 then 1 else
  n*fak(n-1);
fun fak(n) = if n=0 then 1 else n*fak(n-1);
```

- Alternative Schreibweisen zur Definition von Funktionen oder Prozeduren mit Pattern Matching:

```
val rec fak = fn 0 => 1
              | n => n*fak(n-1);
fun fak(0) = 1
  | fak(n) = n*fak(n-1);
```


Beispiel Uhrzeit

- Eine Prozedur, die von der Hardware die Uhrzeit erhält und als Wert liefert, verursacht keinen Nebeneffekt.
- Sie ist keine Funktion, weil sie nicht referenztransparent ist:
  - Der Wert, den sie liefert, ist bei gleichen aktuellen Parametern nicht immer der selbe.
- Die Unterscheidung zwischen Prozeduren, die keine Funktionen sind, und Funktionen je nach dem, ob die Prozeduren Nebeneffekte verursachen, ist nicht ganz zutreffend.

Fälle wie die Uhrzeit-liefernde Prozedur sind seltene Grenzfälle, so dass diese Unterscheidung ihre Nützlichkeit behält.

Überblick

1. Die „Prozedur“ als Kernbegriff der Programmierung
2. Prozeduren zur Bildung von Abstraktionsbarrieren: Lokale Deklarationen
3. Prozeduren versus Prozesse
4. Ressourcenbedarf – Größenordnungen
5. Beispiel: Der größte gemeinsame Teiler

Lokale Deklarationen (1)

- Prozeduren sind u.a. zur Definition von Zwischenergebnissen und Teilberechnungen nützlich.
- Lokale Deklarationen ermöglichen innerhalb einer Prozedur die Verwendung von Variablen als Namen für
  - konstante Werte oder
  - Funktionen oder
  - Prozeduren,

  die nur innerhalb dieser Prozedur verwendet werden können.
- Lokale Deklarationen sind in (fast) allen modernen „höheren“ Programmiersprachen möglich.

Lokale Deklaration (2)

- Maschinensprachen ermöglichen keine echten lokalen Deklarationen, da sie nicht sicherstellen, dass die Geltungsbereiche geschützt sind.
- In SML gibt es zwei syntaktische Möglichkeiten, Namen lokal zu deklarieren:
  - Mit dem Ausdruck „`let`“
  - Mit dem Ausdruck „`local`“

Eine 3. Möglichkeit besteht bei der Deklaration von Funktionen: Die formalen Parameter sind Namen, die nur lokal im Rumpf der deklarierten Funktion gelten.

Lokale Deklarationen mit „`let`“ (1)

- Man verwendet `let`-Ausdrücke, um Deklarationen lokal zu einem Ausdruck zu deklarieren, der selbst keine Deklaration ist.
- Betrachte die folgende Funktion:

  f: $\mathrm{N} \rightarrow \mathrm{N}$

  $x \rightarrow (3x + 1)^2 + (5x + 1)^2 + (3x)^3 + (5x)^3$
- Eine Implementierung in SML sieht so aus:

```
fun hoch2(x : int) = x*x;

fun hoch3(x : int) = x*x*x;

fun f(x)  = hoch2(3*x+1) + hoch2(5*x+1) + hoch3(3*x)
          + hoch3(5*x);
```

Lokale Deklarationen mit „let“ (2)

- Kommen die Funktionen `hoch2` und `hoch3` nur in der Definition von `f` vor, so kann man diese Funktionen lokal zu `f` deklarieren:

```
fun f(x) = let fun hoch2(x : int) = x*x
               fun hoch3(x : int) = x*x*x
           in
               hoch2(3*x+1)+hoch2(5*x+1)+hoch3(3*x)
               +hoch3(5*x)
           end;
```

- Die Syntax des definierenden Teils (oder Rumpfs) einer Funktions- oder Prozedurdeklaration mit lokalen Deklarationen ist:

```
let … in … end;
```


Andere Syntax für mehrere lokale Deklarationen

- Zwischen „`let`“ und „`in`“ dürfen sich mehrere, auch durch „`;`“ getrennte, Deklarationen befinden:

```
fun f(x) = let fun hoch2(x : int) = x*x;
               fun hoch3(x : int) = x*x*x
           in
           hoch2(3*x+1)+hoch2(5*x+1)+hoch3(3*x)
           +hoch3(5*x)
           end;
```

Warum SML die Verwendung von „`;`“ zwischen lokalen Variablen zulässt, wird am Ende dieses Abschnitts erläutert.

Lokale Deklaration in Definitionen von anonymen Funktionen

- Unter Verwendung einer anonymen Funktion (SML-Konstrukt `fn`, gesprochen *lambda* (λ)) kann die Funktion `f` in SML so implementiert werden:

```
val f = fn(x) =>
          let fun hoch2(x : int) = x*x
              fun hoch3(x : int) = x*x*x
          in
              hoch2(3*x+1)+hoch2(5*x+1)+hoch3(3*x)
              +hoch3(5*x)
          end;
```

- In anonymen Funktionen kann `let` genauso benutzt werden wie in benannten.

Verbesserungen des Programmbeispiels (1)

- Statt der lokalen Funktion `hoch2` kann eine Funktion `plus1hoch2` verwendet werden.
- Das führt zu einem verständlicheren und kompakteren Programm:

```
fun f(x) = let fun plus1hoch2(x : int) =
                   let   fun hoch2(x) = x*x
                   in    hoch2(x+1)
                   end;
               fun hoch3(x : int) = x*x*x
           in
               plus1hoch2(3*x)+plus1hoch2(5*x)
               +hoch3(3*x)+hoch3(5*x)
           end;
```

Verschachtelung von lokalen Deklarationen ist möglich.

Verbesserungen des Programmbeispiels (2)

- Die mehrfache Berechnung von `3*x` und `5*x` kann mit lokalen Variablen für Zwischenergebnisse vermieden werden:

```
(*)fun f(x) = let fun plus1hoch2(x : int) = (x+1)*(x+1)
                  fun hoch3(x : int) = x*x*x
                  val x3 = 3*x
                  val x5 = 5*x
              in
                  plus1hoch2(x3)+plus1hoch2(x5)
                  +hoch3(x3)+hoch3(x5)
              end;
```

Manche Übersetzter erkennen in einigen Fällen mehrfache Berechnungen und führen eine ähnliche Änderung durch!

Lokale Deklarationen mit „`local`“ (1)

- Um Deklarationen lokal zu Deklarationen zu deklarieren, verwendet man `local`-Ausdrücke:

  f: N → N

  x → $(3x + 1)^2 + (5x + 1)^2 + (3x)^3 + (5x)^3$ wird ähnlich zu `(*)` deklariert:

```
local    fun plus1hoch2(x : int) = (x+1)*(x+1)
         fun hoch3(x : int)      = x*x*x
in
         fun f(x)=
               let   val x3 = 3*x
                     val x5 = 5*x
               in
                     plus1hoch2(x3)+plus1hoch2(x5)
                     +hoch3(x3)+hoch3(x5)
               end
end;
```

Lokale Deklarationen mit „`local`“ (2)

- Die Syntax eines `local`-Ausdrucks ist:
  `local…in…end.`
- Die lokalen Deklarationen von `x3` und `x5` können NICHT zwischen `local` und dem ersten `in` stehen, weil dieser Bereich außerhalb des Geltungsbereiches des Parameters `x` liegt:
  Zwischen `local` und dem ersten `in` ist `x` unbekannt!
- Zwischen `in` und `end` könnte man noch eine weitere Funktion `f´` deklarieren, die ebenfalls die lokalen Definitionen `plus1hoch2` und `hoch3` verwendet.
- Bei einer Konstruktion wie in `(*)` ist es NICHT möglich, lokale Deklarationen für mehrere Funktionen gemeinsam zu verwenden!

Unterschied zwischen `let` und `local` (1)

- `let` ermöglicht Deklarationen lokal zu Ausdrücken, die keine Deklarationen sind:

```
let       val zwei = 2
          val drei = 3
in
          zwei + drei
end;
```

- `let` ermöglicht KEINE Deklarationen lokal zu Ausdrücken, die selbst Deklarationen sind:

```
let       val zwei = 2
          val drei = 3
in
          val fuenf = zwei + drei
end;
```

FALSCH!!!
Das gilt auch für Funktionsdeklarationen:
z.B. `fun f(x) = x + zwei + drei`

Unterschied zwischen `let` und `local` (2)

- In solchen Fällen kann `local` verwendet werden:

```
local    val zwei = 2
         val drei = 3
in
         val fuenf = zwei + drei
end;
```

Entsprechend sind bei Funktionsdeklarationen nur die Funktionen deklariert.

- Damit ist die Variable `fuenf` mit Wert `5` deklariert.
- `zwei` und `drei` sind nicht deklariert; sie haben nur als internes Hilfsmittel bei der Deklaration von `fuenf` gedient.
- `local` ermöglicht wiederum keine Deklarationen lokal zu Ausdrücken, die keine Deklarationen sind:

```
local    val zwei = 2
         val drei = 3
in             zwei + drei
end;
```

FALSCH!!!

Blockstruktur und Überschatten

Dadurch die gem komplex über we Bezeich

- Programmiersprachen, in denen Variablen als Namen für
  - konstante Werte
  - Funktionen
  - Prozeduren

  lokal deklariert werden können, besitzen die so genannten „Blockstruktur“ und werden „Blocksprachen“ genannt.
- Ein `let-` oder `local-`Ausdruck stellt einen so genannten „Block“ dar.
- In Blocksprachen erfolgt die Bindung von Werten an Variablen statisch (vgl. Abschnitt 2.7).

Beispiel (1)

- Ein Programmierer deklariert einen Namen `N` lokal zu einem Programmteil, den er implementiert:

```
- fun f(x)=    let   val a = 2
               in    a*x
               end;
val f = fn : int -> int
```


Beispiel (2)

- Derselbe Name `N` kann von einem anderen Programmierer lokal zu einem anderen Programmteil mit einer ganz anderen Bedeutung verwendet werden:

```
fun g(x)=     let   val a = 2000
              in    a*x
              end;
val g = fn : int -> int

- f(1);
val it = 2 : int

- g(1);
val it = 2000 : int
```

Festlegung der Geltungsbereiche von Namen – Einführung

- Der Geltungsbereich eines Namens (oder einer Variablen) in Blocksprachen erfolgt nach diesen einfachen Prinzipien:

  1. Der Geltungsbereich eines Namens `N` ist der Block, der die Deklaration von `N` enthält, mit Ausnahme aller in dem Block vorkommenden Blöcke, die Deklarationen desselben Namens `N` enthalten.

  2. Kommt ein Name `N` in einem Block `B` vor und ist `N` in `B` nicht deklariert, so gilt:

     (a) `N` muss in einem Block `B'` deklariert sein, der `B` umfasst.

     (b) Wenn mehrere Blöcke, die `B` umfassen, Deklarationen für `N` enthalten, so gilt die Deklaration desjenigen Blocks, der `B` am engsten umfasst.

Festlegung der Geltungsbereiche unter Verwendung der Umgebung

- Die Verwaltung einer Umgebung als eine geordnete Liste von Gleichungen der Gestalt `Name = Wert` ermöglicht eine einfache Implementierung der vorangehenden Regeln:
  - Beim Eintritt in einen neuen Block, d.h. bei der Auswertung eines neuen Ausdrucks, führt jede Deklaration eines Namens `N` für einen Wert `W` zu einem Eintrag `N = W` am Anfang der Umgebung.
  - Beim Austritt aus einem Block, d.h. bei der Beendigung der Auswertung eines Ausdrucks, werden alle Einträge der Gestalt `N = W` gelöscht, die während der Auswertung des Ausdrucks in die Umgebung eingefügt wurden.
  - Um den Wert eines Namens zu ermitteln, wird die Umgebung von Anfang an durchlaufen. Die zuerst gefundene Gleichung `N = W` gilt als Definition von `N`.

    So gilt als Wert eines Namens `N` der Wert, der bei der „letzten“ (also „innersten“) Deklaration von `N` angegeben wurde. Dadurch werden ältere Deklarationen eines Namens `N` überschattet (siehe Abschnitt 2.7).

Überschatten durch verschachtelte lokale Deklarationen (1)

- Lokale Deklarationen ermöglichen das Überschatten von Deklarationen:

```
- fun f(x) =    let   val a = 2
                      fun g(x) = let val a = 3
                                 in
                                       a * x
                                 end
                in
                      a * g(x)
                end;
val f = fn : int -> int

- f(1);
val it = 6 : int
```

Hier muss man nicht mit Überschatten arbeiten, zur Veranschaulichung macht es allerdings Sinn.
Um Missverständnisse bei Lesern eines Programms zu vermeiden sollte man nicht unüberlegt überschatten!

Überschatten durch verschachtelte lokale Deklarationen (2)

- Die Anwendung des Substitutionsmodells auf den Ausdruck `f(1)` liefert die Erklärung für den Wert dieses Ausdrucks:

```
f(1)
a * g(1)
2 * g(1)
2 * (a * 1)
2 * (3 * 1)
2 * 3
6
```

- Hier können sowohl die Regeln zur Festlegung der Geltungsbereiche von Namen als auch der Algorithmus zur Verwaltung der Umgebung verwendet werden.
- Mit der Einführung von lokalen Variablen verliert das Substitutionsmodell einen Teil seiner Einfachheit.

Auswertung von `val`-Deklarationen

- Die Auswertung einer Deklaration `val N = A` geschieht in einer Umgebung `U`.
- Zunächst wird der Ausdruck `A` in dieser Umgebung `U` ausgewertet.
- Erst nach der Auswertung des Ausdrucks `A` in `U` wird die Umgebung `U` (an ihrem Anfang) um die Gleichung `N = A` erweitert.
- Zur Auswertung des Rumpfes (oder definierenden Teils) einer Deklaration gilt die „alte Umgebung“ vor Berücksichtigung dieser Deklaration.

Eintrag einer Funktionsdeklaration in die Umgebung (1)

- Zur Auswertung einer Funktionsdeklaration

```
(*)      val N = fn P => W
```

  wird eine Gleichung

```
(**)     N = fn P => W
```

  zu der Umgebung (am Anfang) hinzugefügt.

Dies ergäbe keinen Sinn, weil `W` der Rumpf der Funktion namens `N` ist.

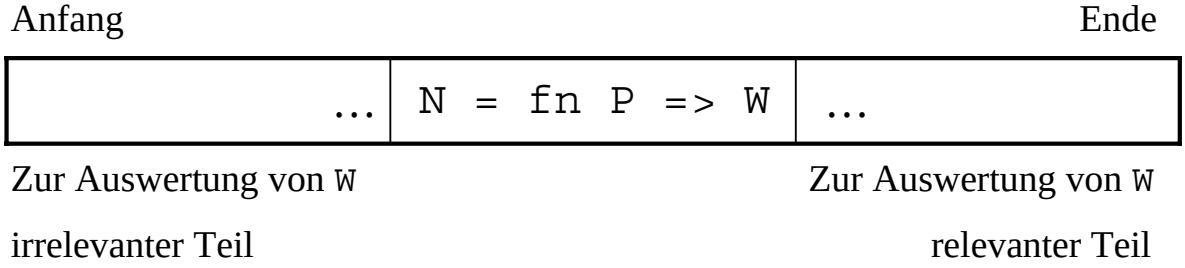
- Zum Zeitpunkt der Auswertung der Funktionsdeklaration `(*)`:
  - wird der Ausdruck `W` nicht ausgewertet.
  - sind keine aktuellen Parameter bekannt, also auch keine Anwendung der Funktion namens `N` durchzuführen.
- Zur (späteren) Auswertung einer Anwendung der Funktion namens `N` wird `W` ausgewertet.
- Welche Umgebung soll für diese Auswertung berücksichtigt werden?

Eintrag einer Funktionsdeklaration in die Umgebung (2)

- Die Umgebung ist eine geordnete Liste, so dass sowohl nach wie vor der Gleichung `(**) N = fn P => W`, die den Wert des Namens `N` liefert, weitere Gleichungen vorkommen können.
- Weitere Gleichungen kommen vor `(**)` vor, wenn weitere („neuere“) Deklarationen nach der Deklaration von `N` stattgefunden haben.
- Die Blockstruktur (und das daraus folgende Prinzip der statischen Bindung) verlangt, dass nur die Gleichungen, die nach `(**)` vorkommen, also die „älter“ als die Deklaration von `N` sind, zur Auswertung von `W` berücksichtigt werden.

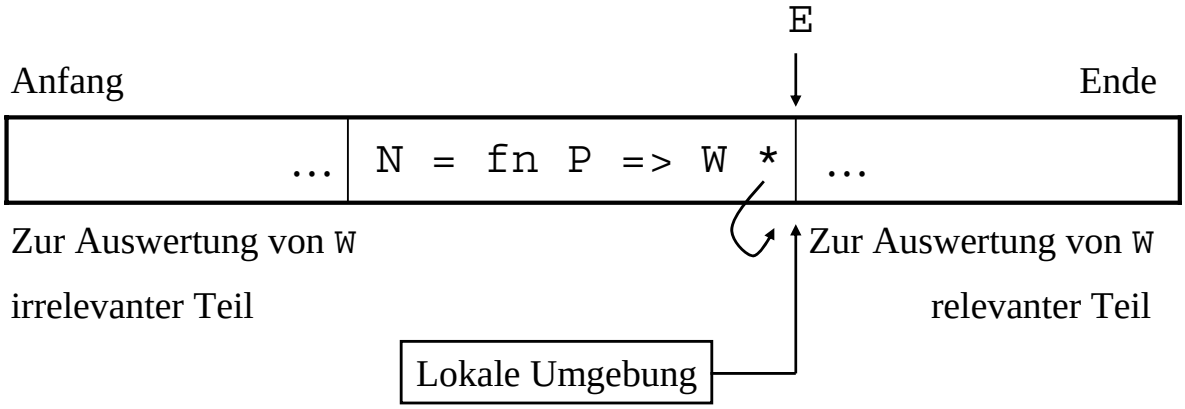
Eintrag einer Funktionsdeklaration in die Umgebung (3)

- Die Umgebung wird vom Anfang her gelesen oder erweitert:



Auswertung einer Funktions-anwendung – Lokale Umgebung (1)

- Zur Auswertung einer Anwendung der Funktion namens `N` müssen auch die aktuellen Parameter der Funktionsanwendung berücksichtigt werden.
- Der für die Auswertung von `W` relevante Teil wird um eine „*lokale Umgebung*“ erweitert:



Auswertung einer Funktions-anwendung – Lokale Umgebung (2)

- `E` gibt die Stelle an, ab der die Umgebung gelesen wird.
- Die Umgebung, die während der Auswertung einer Anwendung der Funktion namens `N` berücksichtigt wird, besteht aus der lokalen Umgebung gefolgt von dem Teil der (nichtlokalen, auch global genannten) Umgebung, die zur Auswertung von `W` relevant ist.
- Jede Prozedurdeklaration kann lokale Deklarationen beinhalten.
- Also kann während einer Auswertung eine Verkettung von mehreren lokalen Umgebungen entstehen.

Diese Verkettung spiegelt die Schachtelung von Ausdrücken wider, in denen Parameter von Funktionsanwendungen oder lokale Variablen deklariert werden.

Sequenzielle Auswertung von lokalen `let`-Deklarationen

- Aufeinanderfolgende Deklarationen werden der Reihe nach ausgewertet:

```
- val x = 2;
val x = 2 : int
- let   val x3 = 3 * x;
        val x5 = 5 * x3
  in
        1 + x5
  end;
val it = 31 : int
```

Auch wenn es einen Namenskonflikt gibt, z.B.

```
...
- let val x = 3 * x;
      val x = 5 * x
  in
      1 + x
  end
...
```

bleibt die Auswertung durch die Reihenfolge gleich.

Nichtsequenzielle Auswertung von lokalen Deklarationen

- Zuerst werden alle Ausdrücke in den Rümpfen der Deklarationen in derselben äußeren Umgebung ausgewertet.
- Dann werden alle berechneten Werte gleichzeitig an ihre Namen gebunden.

```
- val   x = 2;
val     x = 2 : int
- let   val   x = 3 * x
        and   x5 = 5 * x
  in
        x * x5
  end;
val it = 60 : int
```

Das `and`-Konstrukt bewirkt eine nicht-sequenzielle Auswertung von lokalen Deklarationen in SML.
Es kann auch in `local`-Ausdrücken verwendet werden.

An die lokale Variable `x5` wird der Wert `10` gebunden!
(`x5` hätte den Wert `30`, wenn `and` durch `val` ersetzt würde)

Wechselseitige Rekursion

- Zwei oder mehrere Funktionen heißen „*wechselseitig rekursiv*“, wenn sie sich wechselseitig aufrufen.
- Die Rümpfe aller Deklarationen müssen in derselben Umgebung ausgewertet werden.
- Deshalb ist das `and`-Konstrukt für die Deklaration von wechselseitig rekursiven Funktionen notwendig.

Beispiel zur wechselseitigen Rekursion

```
val rec ungerade  = fn 0 => false
                  |  1 => true
                  |  n => gerade(n-1)
and
        gerade    = fn 0 => true
                  |  1 => false
                  |  n => ungerade(n-1);
```

Beispiel mit `fun` zur wechselseitigen Rekursion

```
fun ungerade(0) = false
  | ungerade(1) = true
  | ungerade(n) = gerade(n-1)
and
    gerade(0) = true
  | gerade(1) = false
  | gerade(n) = ungerade(n-1);
```


Auswertung von `val-rec`-Deklarationen

- Eine Funktionsdeklaration der Form

  ```
  val rec N = fn P => W
  ```

  wird wie eine Funktionsdeklaration der Form

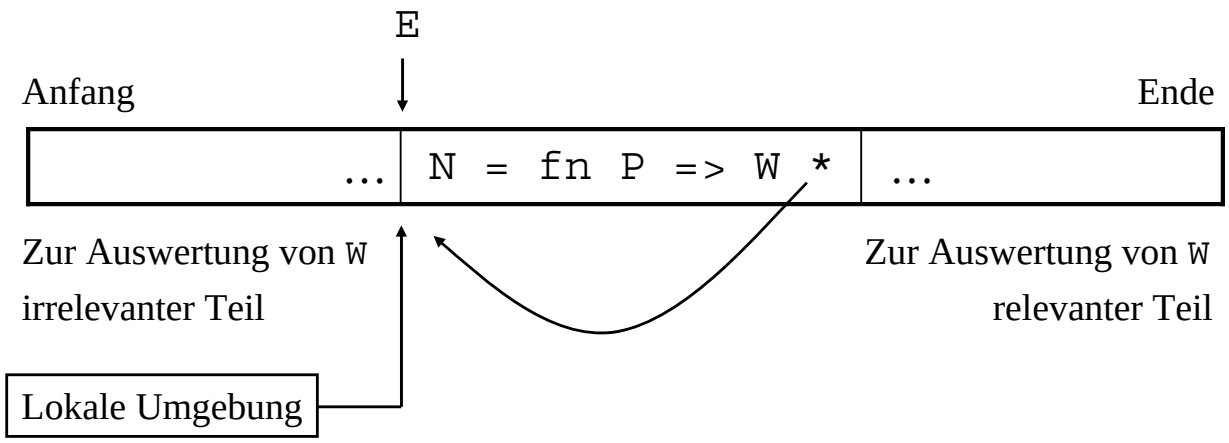
  ```
  val N = fn P => W
  ```

  behandelt, mit dem Unterschied, dass der Verweis am Ende der lokalen Umgebung *vor* statt hinter die Gleichung

  ```
  N = fn P => W
  ```

  in der globalen Umgebung zeigt.

Graphische Darstellung der Auswertung



- `E` gibt die Stelle an, ab dem die Umgebung während der Auswertung einer Anwendung der rekursiven Funktion namens `N` gelesen wird.

Auswertung von `fun`-Deklarationen

- Ein Funktionsdeklaration der Form:

```
fun N P = W
```

  wird genauso ausgewertet wie die Funktionsdeklaration:

```
val rec N = fn P => W
```

- Das Konstrukt `fun` ist „syntaktischer Zucker“ für die letztere Schreibweise.