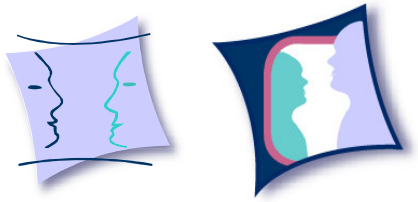
Technische Universität Darmstadt



Telecooperation/RBG

Grundlagen der Informatik I
Thema 4: Auswertungsreihenfolge und Lexikalisches Scoping

Prof. Dr. Max Mühlhäuser
Dr. Guido Rößling

Nochmal: Applikative vs. Normale Auswertungsreihenfolge

- Zur Erinnerung:
  - *Applikative Reihenfolge*: Erst Operator und alle Operanden auswerten, dann substituieren
  - *Normale Reihenfolge*: Operator auswerten, dann die (unausgewerteten) Operanden für die formalen Argumente des Operators substituieren

- Wir hatten gesagt:
  - Die Reihenfolge der Auswertung spielt für das Ergebnis keine Rolle, **wenn es ein Ergebnis gibt**
  - Diesen letzten Punkt wollen wir nun beleuchten

Nochmal: Applikative vs. normale Auswertungsreihenfolge

- Beispiel: “Eigenbau” eines if-statements als Prozedur mit Hilfe von “cond”

```
(define (my-if test then-exp else-exp)
  (cond
    [test then-exp]
    [else else-exp]))
```

  - Scheint zu funktionieren:
    `(my-if true 3 5)` → `3`
    `(my-if false 3 5)` → `5`

- Betrachten wir nun folgendes Programm:

```
;; ! : N  ->  N
;; computes the faculty function
(define (! n)
  (if (zero? n) 1 (* n (! (pred n)))))
```

  - Ersetzen wir nun das **if** durch **my-if**

Nochmal: Applikative vs. normale Auswertungsreihenfolge

- Version der Funktion mit **my-if**

```
;; ! : N  ->  N
;; computes the faculty function
(define (! n)
  (my-if (zero? n) 1 (* n (! (pred n)))))
```

- Ein Aufruf wie (! 2)
  - terminiert nicht bei applikativer Auswertungsreihenfolge,
  - bei normaler Reihenfolge jedoch schon

- Wieso?

Nochmal: Applikative vs. normale Auswertungsreihenfolge

- Viele Sonderformen in Scheme müssten keine Sonderformen sein, wenn Scheme die normale Auswertungsreihenfolge benutzen würde
  - z.B. `if`, `cond`, `and`, `or`
  - Das würde die Semantik der Sprache vereinfachen
  - Normale Auswertungsreihenfolge ermöglicht zudem sehr elegante Programmiertechniken (z.B. “streams”)
- Wieso benutzt Scheme also applikative Auswertung (wie die meisten anderen Programmiersprachen)?
  - Einige Features wie Zuweisungen oder E/A (später…) zerstören die Konfluenz (die Auswertungsreihenfolge beeinflusst das Ergebnis)
  - In diesem Fall ist applikative Auswertungsreihenfolge vorzuziehen, weil transparenter ist, wann ein Ausdruck ausgewertet wird
  - Es gibt allerdings moderne Techniken (z.B. sogenannte “Monaden”), die Zuweisungen, E/A etc. ermöglichen und trotzdem die Unabhängigkeit von der Ausführungsreihenfolge erhalten
    - Das ist ein Thema der Vorlesung “Konzepte der Programmiersprachen”

Blockstruktur

- Wir haben gesehen, dass Hilfsprozeduren wertvoll sind
- Deren Verwendung verursacht aber auch Probleme:
  1. Wenn die Anzahl der Prozeduren zu groß wird, verliert man schnell den Überblick
  2. Jede Prozedur „kostet“ einen Namen: der Namensraum für Prozeduren wird immer kleiner
     - Gefahr von Verwechselungen oder Namenskonflikten wächst
  3. Sehr viele Daten müssen explizit als Parameter an Hilfsprozeduren weitergegeben werden
- Daher gibt es die Möglichkeit, lokale Namen zu definieren und an Werte oder Prozeduren zu binden
- Bevor wir die Scheme Konstrukte dazu diskutieren, werden wir zunächst die bisher verwendete Untermenge von Scheme formaler definieren …

Intermezzo:

Syntax & Semantik von Scheme

Syntax

- Ähnlich wie die natürlichen Sprachen haben auch die Programmiersprachen ein Vokabular und eine Grammatik
  - Das Vokabular ist eine Sammlung der „Wörter“', aus denen wir „Sätze“' in unserer Sprache bilden können.
  - Ein Satz in einer Programmiersprache ist ein Ausdruck oder eine Funktion
  - Die Grammatik der Sprache sagt uns, wie wir ganze Sätze aus Wörtern bilden.

- Der Ausdruck **Syntax** bezieht sich auf Vokabular und Grammatik von Programmiersprachen

Semantik

- Nicht alle grammatikalisch richtigen Sätze sind sinnvoll
  - Weder in Deutsch noch in Programmiersprachen.
  - „Die Katze ist schwarz“' ist ein sinnvoller Satz.
  - „Der Ziegel ist ein Auto“ macht wenig Sinn, auch wenn der Satz grammatikalisch richtig ist.

- Um herauszufinden ob ein Satz sinnvoll ist, müssen wir die **Bedeutung** oder die **Semantik** der Wörter und Sätze verstehen
  - Für Programmiersprachen gibt es verschiedene Wege, um den Sinn von einzelnen Sätze/Ausdrücken zu erklären.
  - Den Sinn von Scheme-Programmen diskutieren wir mit einer Erweiterung der bekannten Gesetze aus Arithmetik und Algebra (Substitutionsmodell)

Das Vokabular von Scheme (Syntax)

<var>	=	x | a | alon| ...
<fct>	=	area-of-disk | perimeter | ...
<con>	=	true | false |
		'a | 'doll | 'sum | ...
		1 | -1 | 3/5 | 1.22 | ...
<prm>	=	+ | - | ...

- Variablen <var>: Namen von Werten
- Funktionen <fct>: Namen von Funktionen
- Konstanten <con>: boolean, Symbole, numerische Konstanten
- Primitive Operationen <prm>: Die Grundfunktionen, die Scheme von Anfang an zur Verfügung stellt

- **Notation**:
  - Zeilen zählen einfache Beispiele auf, getrennt durch ein „|“
  - Punkte bedeuten, dass es noch mehr Dinge derselben Art in der Kategorie gibt

Beginning Student Scheme: Grammatik (Syntax)

```
<def>   =   (define (<fct> <var> ...<var>) <exp>)
          | (define <var> <exp>)
          | (define-struct <var0> (<var-1> ...<var-n>))
<exp>   =   <var>
          | <con>
          | (<prm> <exp> ...<exp>)
          | (<fct> <exp> ...<exp>)
          | (cond [<exp> <exp>] ... [<exp> <exp>])
          | (cond [<exp> <exp>] ... [else <exp>])
          | (and <exp> <exp>)
          | (or <exp> <exp>)
          | (if <exp> <exp> <exp>)
```

Kann auch leer sein

Korrekte Anzahl Argumente

Diese Grammatik wurde vereinfacht; *nicht alle daraus erzeugbaren* Ausdrücke sind gültig!

Zwei Kategorien von Sätzen:
Definitionen (<def>) und Ausdrücke (<exp>)

Grammatik von Scheme: Beispiele

- Beispiele für Ausdrücke:
  - `'all`: Symbol, also ein Ausdruck
  - `x`: jede Variable ist ein Ausdruck
  - `(f x)`: eine Funktionsanwendung, weil `x` eine Variable ist

- Die folgenden Sätze sind *keine* korrekten Ausdrücke:
  - `(f define)`: stimmt teilweise mit der Form einer Funktionsanwendung überein, aber es benutzt `define` als sei es eine Variable.
  - `(cond x)`: kann kein korrekter `cond`-Ausdruck sein, weil an zweiter Stelle eine Variable steht, und nicht ein geklammertes Paar von Ausdrücken
  - `()`: Grammatik verlangt, dass nach jeder linken Klammer etwas anders kommt als eine rechte Klammer

Grammatische Terminologie

- Eine Definition **(<def>)** besteht aus:
  - **Header**: Die zweite Komponente einer Definition, z.B. die nicht leere Sequenz von Variablen
  - **Parameter** einer Funktion: Die Variablen, die der ersten Variable im Header folgen

    (**define** (<function-name> <parameter> ...<parameter>) <body>)
  - **Body**: Die Ausdruck-Komponente einer Definition

- Eine Anwendung besteht aus
  - **Funktion**: Die erste Komponente
  - **Argumente** (**tatsächliche Argumente**): die übrigen Komponenten

    (<function-name> <argument> ...<argument>)
- Ein **cond**-Ausdruck besteht aus **cond**-Zeilen (**cond**-Bedingungen) die jeweils aus zwei Ausdrücken bestehen:
  - **Frage** (**Bedingung**) und **Antwort**

    (**cond** [<question> <answer>] <**cond-clause**> ...)

Syntax des `local`-Ausdrucks

```
(local ((define PI 3)) (* PI 5 5))
```

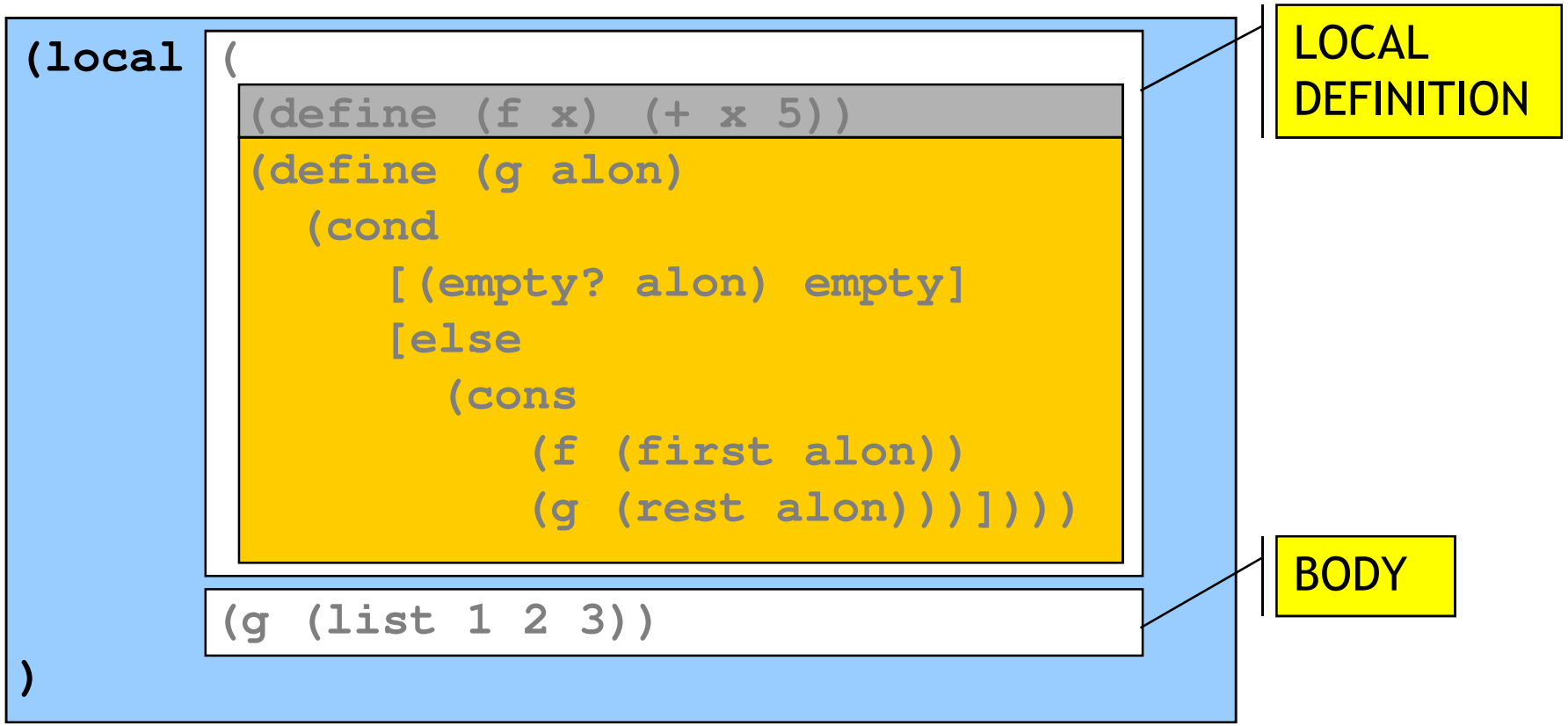
```
<exp>    =   ...
             |   ( local (<def-1> ...<def-n>) <exp> )
```

- Local Definition: Parametrisierte Sequenz nach dem Schlüsselwort `local`
  - Definitionen werden „LOCALLY DEFINED“ Variablen, Funktionen oder Strukturen genannt.
    - Definitionen in dem Definitions-Fenster werden „TOP-LEVEL DEFINITIONS“ genannt.
  - Jeder Name kann höchstens einmal auf der linken Seite vorkommen, sei es in einer Variablen- oder Funktionsdefinition.
  - Der Ausdruck einer Definition wird der RIGHT-HAND SIDE Ausdruck genannt.
- Body: Ausdruck (<exp>) nach den Definitionen

Ende Intermezzo:

Syntax & Semantik von Scheme

Blockstruktur: Beispiel



- Zwei lokal definierte Prozeduren: **f** und **g**
- **g** ruft **f** auf
- Body ruft **g** auf

Eigenschaften von lokalen Definitionen

- Der Rumpf der lokalen Definitionen kann auf weiter außen definierte Namen zugreifen
  - Hilfsprozedur **`plus-y-sqr`** kann auf **`y`** zugreifen
    - Bei nicht-lokaler Definition müsste **`y`** stets explizit übergeben werden
    - Lokale Definitionen können also Parameter einsparen

```
(define (f x y z)
  (local (
     (define (square n) (* n n))
     (define (plus-y-sqr q) (square (+ q y))))
     (+ (plus-y-sqr x) (plus-y-sqr z))))
```

$(x+y)^2 + (z+y)^2$

- Von außen sind die lokal definierten Namen nicht sichtbar
  - An anderen Stellen könnte lokal noch mal derselbe Name definiert werden

Eigenschaften von lokalen Definitionen

- Ein `local` Konstrukt ist ein Ausdruck, und daher universell einsetzbar
  - z.B. als Operator, Operand oder Rumpf einer Prozedur
  - Abgeschlossenheitseigenschaft
- Insbesondere können lokale Definitionen beliebig geschachtelt werden (sogenannte Blockstruktur)
  - z.B. im Rumpf einer lokalen Definition
  - Skalierbarkeit

$(x+y)^2 + (z+y)^2$

```
(define (f x y z)
  (local (
    (define (plus-y-sqr q)
       (local (
         (define (square n) (* n n)))
         (square (+ q y)))))

  (+ (plus-y-sqr x) (plus-y-sqr z))))
```

Eigenschaften von lokalen Definitionen

- Eine wichtige Eigenschaft lokaler Definitionen ist **lexikalisches Scoping**
- **Scoping** ist die Strategie, nach der ein Name einer Definition zugeordnet wird
- Lexikalisches Scoping bedeutet, dass immer die in der Schachtelungsstruktur nächste Definition benutzt wird
  - So können wir den Parameter von **square** auch **z** nennen
  - Trotzdem wird **z** im Rumpf von **square** nicht mit dem Parameter **z** von **f** verwechselt

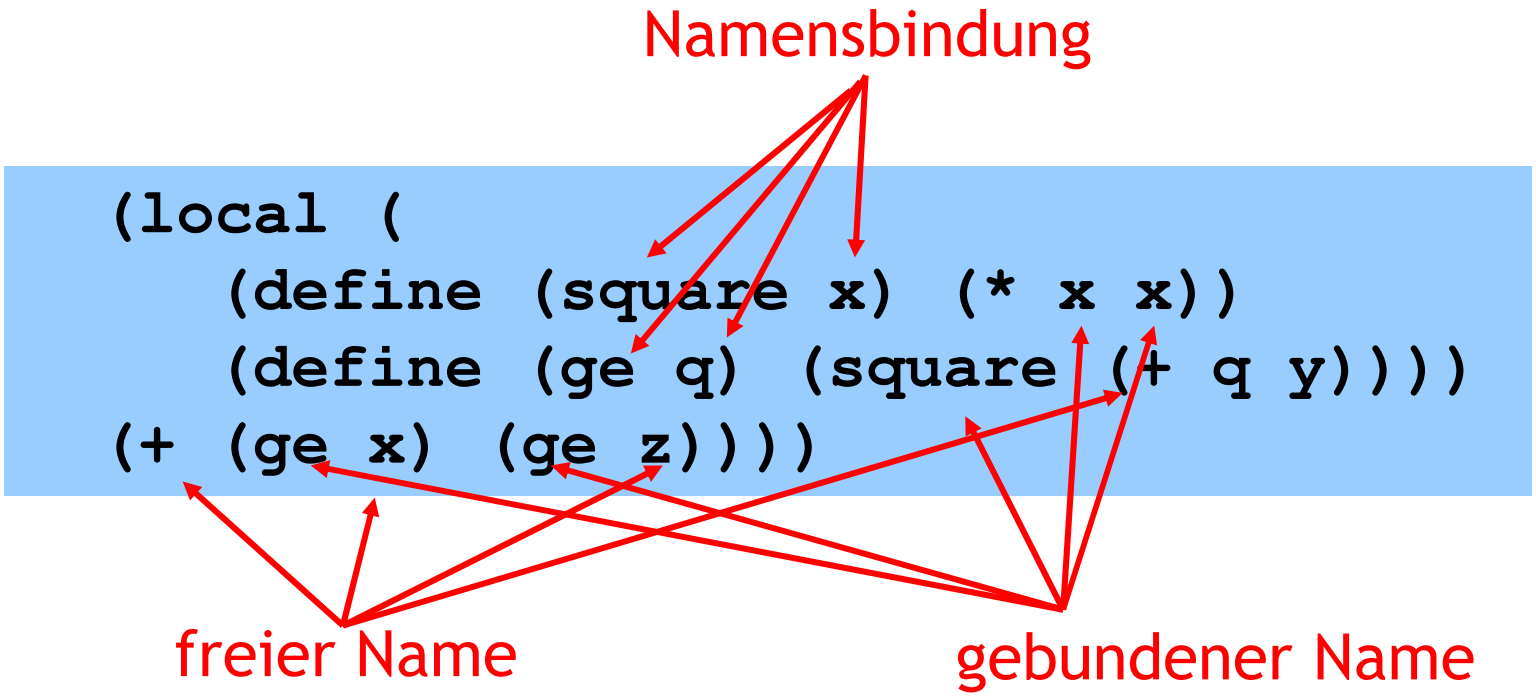
```
(define (f x y z)
  (local (
     (define (square z) (* z z))
     (define (plus-y-sqr q) (square (+ q y))))
  (+ (plus-y-sqr x) (plus-y-sqr z))))
```

Überdeckt äußeres **z** innerhalb von **square**

$(x+y)^2 + (z+y)^2$

Lexikalisches Scoping

- Um lexikalisches Scoping zu verstehen, ist es wichtig, zwischen **Namensbindungen**, **gebundenen Namen** und **freien Namen** zu unterscheiden
  - „frei“ und „gebunden“ sind immer relativ zu einem Ausdruck



Lexikalisches Scoping

- Der ***Scope* einer Namensbindung** ist der textuelle Bereich, in dem sich ein Auftreten des Namens auf diese Namensbindung bezieht
  - Top-Level Definitionen haben ***globalen Scope***
  - Der **Scope eines Prozedurparameters** ist der Körper der Prozedur
  - Der **Scope einer lokalen Definition** ist der Ausdruck (letzter Operand) in der `local` Definition

- Wie wir gesehen haben, kann es „Löcher“ im Scope einer Namensbindung geben
  - Durch Überdeckung der Namensbindung durch eine andere Namensbindung desselben Namens

Testen Sie lexikalisches Scoping mit dem „Check Syntax“ Feature von DrScheme

Zur Erinnerung: Auswertungsregel für Prozeduren

- Anwendungsregel für Prozeduren

  - Die Prozedur ist eine **primitive Prozedur** → führe die entsprechenden Maschineninstruktionen aus.

  - Die Prozedur ist eine **zusammengesetzte Prozedur**
    - Werte den Rumpf der Prozedur aus
    - ***Ersetze dabei jeden formalen Parameter mit dem entsprechenden aktuellen Parameterwert***, der bei der Anwendung angegeben wird.

```
(define (f x ) (* x x))   (f 5 )
```

Ersetzung der Parameter und Lexikalischer Scope

- Bei der Ersetzung der Parameter durch die aktuellen Werte darf nicht einfach blind ersetzt werden
  - Nur das Auftreten der Parameternamen **im Scope der Parameterdefinition** darf ersetzt werden!
  - Genauer: *Die Vorkommen des Namens, die im Prozedurkörper **frei** sind, müssen ersetzt werden*
- Beispiel:
  - Keine Ersetzung von **x** durch **2** in **square**
    - Kein freies Vorkommen von **x** in **square**

```
(f 2 3 4) →  (define (f x y z)
               (local (
                  (define (square x) (* x x))
                  (define (plus-y-sqr q) (square (+ q y))))
               (+ (plus-y-sqr x) (plus-y-sqr z))))
```

Auswertung von `local` Blöcken im Substitutionsmodell

- Wir müssen unser Substitutionsmodell erweitern, um lokale Blöcke auswerten zu können
  - Wir haben bisher keine Regel für lokale Blöcke

- **Idee**: Lokale Definitionen werden auf Top-Level "gezogen"
  - Bei diesem Prozess müssen "neue" Namen vergeben werden
  - Er muss bei **jeder** Auswertung der **local** Definition gemacht werden
    - Kann nicht statisch erfolgen

Auswertung von `local` Blöcken: Beispiel

```
(define y 20)
(+ y
   (local ((define y 10)
           (define z (+ y y)))
          z))
```

```
(define y 20)
(+ y
   (local ((define y1 10)
           (define z1 (+ y1 y1)))
          z1))
```

```
(define y 20)
(define y1 10)
(define z1 (+ y1 y1))
(+ y z1)
```

Schritt 1: Umbenennung der lokalen Namen mit "neuen" Namen

Schritt 2: Herausziehen der lokalen Definitionen auf die oberste Ebene

Die weitere Auswertung erfolgt nach dem bisherigen Substitutionsmodell

Auswertung von `local` Blöcken

- Wieso ist es wichtig, dass die lokalen Definitionen bei jeder Auswertung neu (mit neuen Namen) auf die Top-Level Ebene gezogen werden?
  - Zur Veranschaulichung verfolgen Sie die Auswertung des folgenden Programms mit dem DrScheme Stepper:

```
(define (fac n)
  (local ((define m (- n 1)))
    (if (zero? n)
        1
        (* n (fac m)))))

(fac 5)
```

Dr. G. Rößling
Prof. Dr. M. Mühlhäuser
RBG / Telekooperation

Benutzung von `local`

- Richtlinien für die Benutzung von `local`:
  - Wenn Sie bei der Entwicklung einer Prozedur nach dem Designrezept feststellen, dass Sie Hilfsprozeduren benötigen, die Sie lediglich innerhalb der zu schreibenden Prozedur benötigen, definieren Sie diese auch nur innerhalb eines **local** Blocks.
  - Nutzen Sie aus, dass Sie innerhalb des **local** Blocks Zugriff auf lexikalisch weiter außen liegende Namen (z.B. Prozedurparameter) haben.
  - Wenn Sie innerhalb eines Prozedurkörpers einen Wert mehrfach ausrechnen, definieren Sie einen lokalen Namen mit diesem Wert und benutzen ihn an allen Stellen, wo sie diesen Wert zuvor berechneten.

Benutzung von `local`: Beispiel

```
(define (make-rat n d)
    (make-xy
       (/ n (gcd n d))
       (/ d (gcd n d))))
```

```
(define (make-rat n d)
  (local ((define t (gcd n d)))
    (make-xy
       (/ n t)
       (/ d t)))))
```