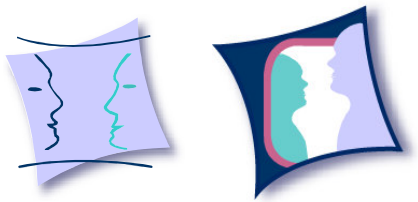
Technische Universität Darmstadt



Telecooperation/RBG

Grundlagen der Informatik 1
Thema 1: Grundelemente der Programmierung

Prof. Dr. Max Mühlhäuser
Dr. Guido Rößling

Was ist Programmierung?

Schauen wir mal, was einige der „großen Köpfe“ der Informatik dazu sagen:

„To program is to understand“
Kristen Nygaard

„Programming is a Good Medium for Expressing Poorly Understood and Sloppily Formulated Ideas“
Marvin Minsky, Gerald J. Sussman

Strukturierungsmechanismen einer Programmiersprache

- Eine mächtige Programmiersprache ist mehr als ein Hilfsmittel um einen Computer anzuweisen, Aufgaben durchzuführen.

- Sie dient auch als Rahmen, innerhalb dessen wir unsere Ideen über die Problemdomäne organisieren

Wenn wir eine Sprache beschreiben, sollten wir die Hilfsmittel beachten, die sie uns zum Kombinieren von einfachen Ideen anbietet, um komplexere Ideen zu bilden.

Strukturierungsmechanismen einer Programmiersprache

- Jede mächtige Programmiersprache hat **drei Mechanismen, um Prozessideen zu strukturieren**:
  - **Primitive Ausdrücke**
    - Repräsentieren die einfachsten Einheiten der Sprache.
  - **Kombinationsmittel**
    - Zusammengesetzte Elemente werden aus einfacheren Einheiten konstruiert.
  - **Abstraktionsmittel**
    - Zusammengesetzte Elemente können benannt und weiter als Einheiten manipuliert werden.

Strukturierungsmechanismen in der Elektronik

- **Primitive**
  - Widerstände, Kondensatoren, Induktivitäten, Spannungsquellen, …
- **Kombinationsmittel**
  - Richtlinien für das Verdrahten der Schaltkreise
  - Standardschnittstellen (z.B. Spannungen, Strömungen) zwischen den Elementen
- **Abstraktionsmittel**
  - “Black box” Abstraktion – denke über einen Unter-Schaltkreis als eine Einheit: z.B. Verstärker, Regler, Empfänger, Sender, …

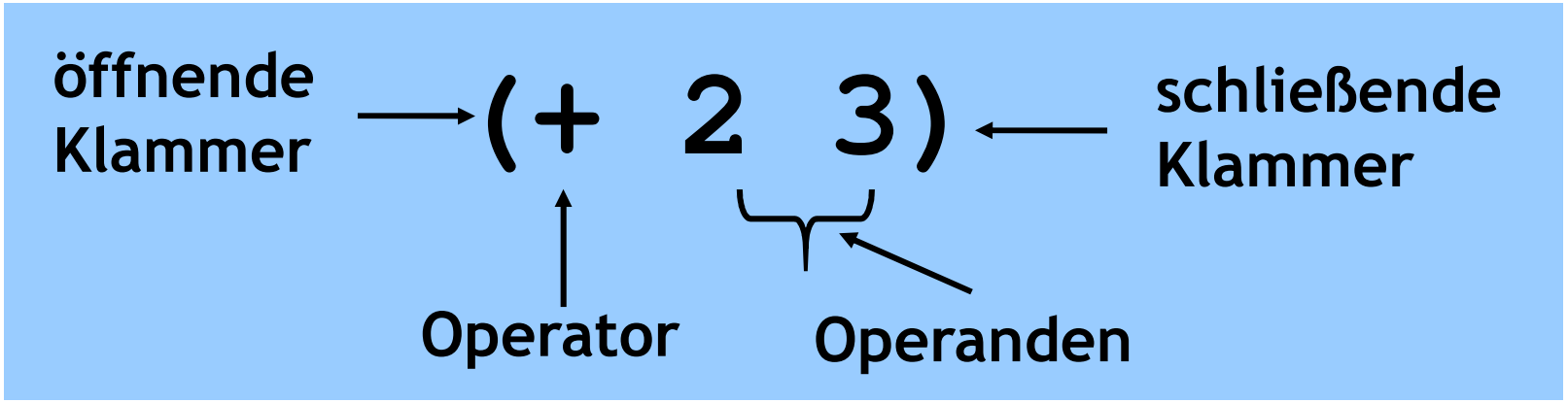
Sprachelemente – Die Primitiven

- **Zahlen**
  - Beispiele: 23, -36
  - Zahlen sind selbstauswertend: Die Werte der Ziffern sind die Zahlen, die sie bezeichnen.

    `23` → *23*

    `-36` → *-36*
- **Boolesche Werte**
  - `true` und `false`
  - ebenfalls selbstauswertend
- **Namen für eingebaute Prozeduren**
  - Beispiele: `+, *, /, -, =, …`
  - Was ist der Wert von so einem Ausdruck?
    - Der Wert von + ist eine Prozedur, die Zahlen addiert
      - Das werden wir später als „Higher-Order Procedures“ kennenlernen
  - Auswertung durch Nachschlagen des dem Namen zugewiesenen Wertes.

Sprachelemente – Kombinationen



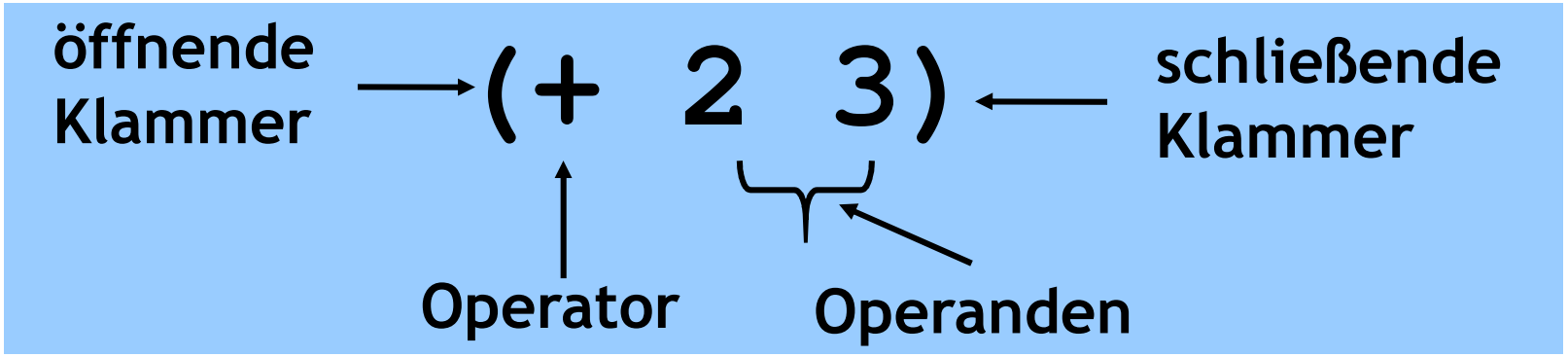
Präfixdarstellung

Der Wert einer Kombination wird bestimmt durch die Ausführung der (durch den Operator) angegebenen Prozedur mit den Werten der Operanden.

Sprachelemente – Kombinationen

- **Zusammengesetztes Element**:
  - Eine Sequenz von Ausdrücken, eingeschlossen in Klammern.
  - Die Ausdrücke selbst sind primitiv oder wiederum zusammengesetzt
- Beispiel:
  - Numerische Ausdrücke können mit Ausdrücken kombiniert werden, die primitive Prozeduren repräsentieren (+ oder *), um einen *zusammengesetzten Ausdruck* zu erstellen.

stellt die Anwendung der Prozedur auf die Zahlen dar



Sprachelemente - Kombinationen

- Kombinationen können verschachtelt werden – Regeln einfach rekursiv anwenden

```
(+ 4 (* 2 3)) = (4 + (2 * 3)) = 10

(* (+ 3 4) (- 8 2))
     = ((3 + 4) * (8 - 2))
     = 42
```

- Eine Kombination bedeutet immer eine Anwendung einer Prozedur
  - Klammern können nicht eingefügt oder weggelassen werden, ohne die Bedeutung des Ausdrucks zu ändern.

Sprachelemente - Abstraktionen

1. **Erstelle** eine komplexe Sache aus primitiveren Sachen,
2. **Benenne** sie,
3. Behandle sie als eine **primitive** Sache.

- Einfachstes Abstraktionsmittel: **`define`**

```
(define score (+ 27 3))
(define PI 3.14)
```

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

Sprachelemente - Abstraktionen

- **Sonderformen**: Geklammerte Ausdrücke, die mit einem der wenigen Scheme-**Schlüsselwörter** starten.

- Beispiel: **`define`**
  - Mit **`define`** kann man einen Wert an einen Namen binden
    Beispiel: `(define score (+ 27 3))`
  - Die **`define`** **Sonderform** wertet den zweiten Ausdruck nicht aus (in dem Beispiel: **`score`**)
  - Sie assoziiert diesen Namen mit dem Wert des dritten Ausdruckes in einer Umgebung.

- Der Rückgabewert einer Sonderform ist nicht spezifiziert.

Namensgebung und Umgebung

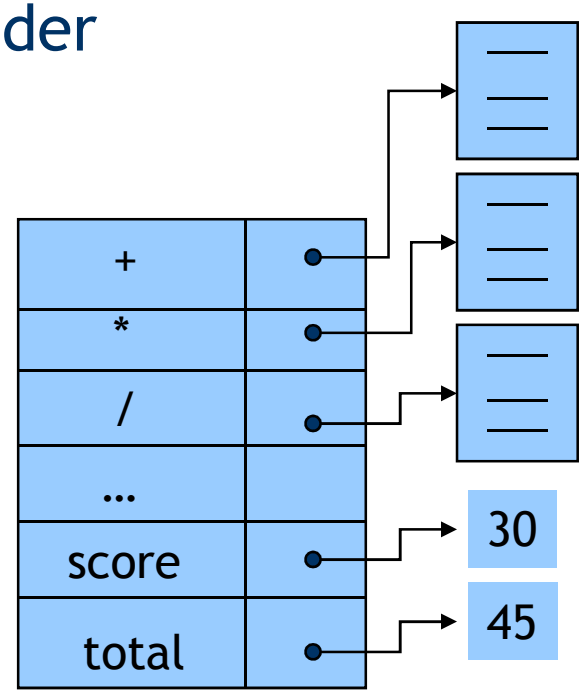
- Ein wichtiger Aspekt einer Programmiersprache ist die Möglichkeit, Objekte über Namen zu referenzieren.
  - Ein Name identifiziert eine *Variable*, deren *Wert* ein Objekt ist.

- **Umgebung**: der Interpreter unterhält eine Art Speicher, um Name-Objekt Paare zu verwalten.
  - Assoziiere die Werte mit den Symbolen
  - Hole die Werte später über die Symbole

Namensgebung und Umgebung

- Um den Wert auszurechnen, der durch den Namen repräsentiert wird, reicht es, ihn in der Umgebung aufzulösen.
  - Beispiel: Die Evaluierung von `score` ist `30`

```
(define score (+ 27 3))
(define total (+ 30 15))
(* 100 (/ score total))
```



  - **Beachte**: wir haben das (implizit) schon gemacht für die Symbole der arithmetischen Operatoren `+`, `*`, usw. (d.h. den Wert erhalten, der durch diese Namen referenziert wird)

Auswertungsregel

- Selbst-auswertend (self-rule) → gib den Wert zurück
  Die Werte der Ziffern sind die Zahlen, die sie bezeichnen.
- Eingebauter Operator → gebe die Sequenz der Maschineninstruktionen zurück, die die entsprechenden Operationen durchführen.
- Name (name-rule) → gebe den Wert zurück, der in der Umgebung mit diesem Namen assoziiert wurde.
- Sonderform → mache was besonderes.
- Kombination →

  i. *Rechne die Unterausdrücke* (in beliebiger Reihenfolge) *aus*

  ii. *Wende die Prozedur, die der Wert des am weitesten links liegenden Unterausdruckes ist (der Operator), auf die Argumente an – die Werte der restlichen Unterausdrücke (Operanden).*

  Beispiel einer Kombination: `(+ 4 (* 2 3))`

Auswertungsregel

- Die Auswertungsregel ist rekursiv
  - Bei einer Kombination ruft die Regel sich selbst auf
  - Jedes Element muss ausgewertet werden, bevor die Gesamtauswertung einer Kombination abgeschlossen werden kann

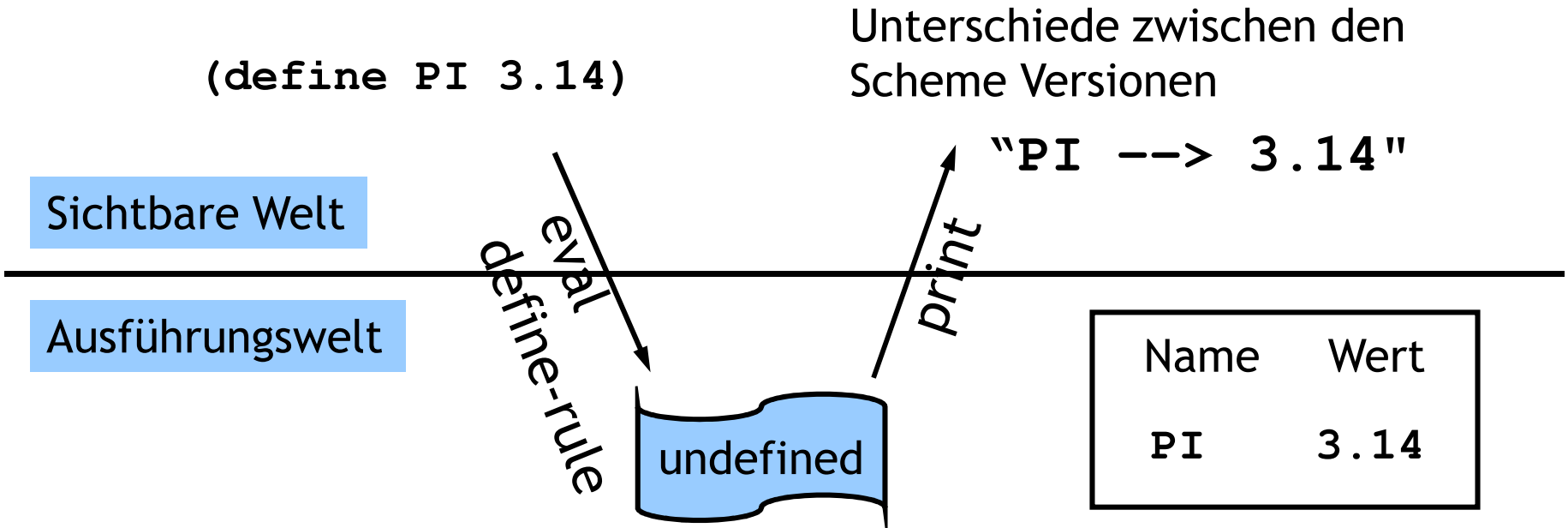
- Die Auswertung der folgenden Kombination erfordert die Anwendung der Auswertungsregel auf vier unterschiedliche Kombinationen.

```
( *  (+ 2 (* 4 6) )  (+ 3 5 7) )
```

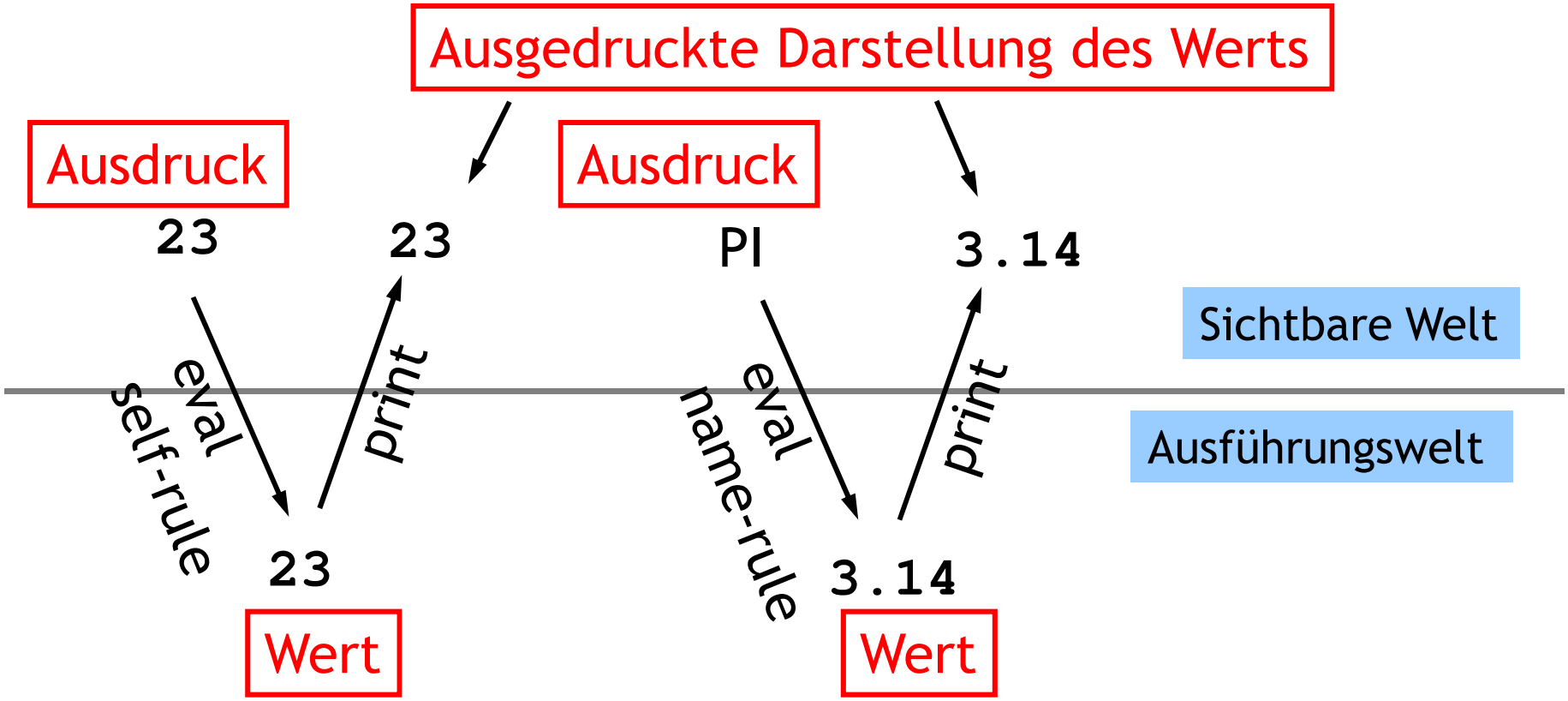
Lesen-Auswerten-Ausgeben-Schleife

`define`-Regel:

- Werte nur den zweiten Operanden aus
- Der Name des ersten Operanden ist an den berechneten Wert gebunden
- Gesamtwert des Ausdrucks ist undefiniert



Lesen-Auswerten-Ausgeben-Schleife des Interpreters



Namensregel:
Schlage in der aktuellen Umgebung den Wert unter dem Namen nach

Finden gemeinsamer Muster

- Haben folgende Kombinationen etwas gemeinsam?

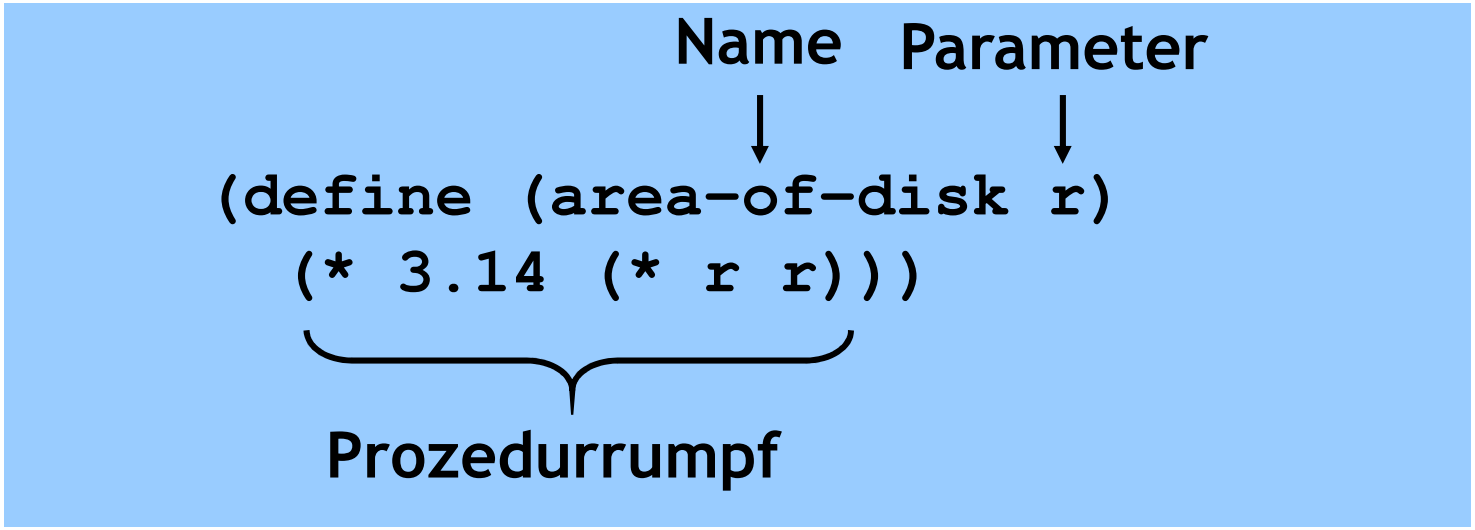
```
(* 3.14 (* 5 5))
(* 3.14 (* 23.2 23.2))
(* 3.14 (* r r))
```

Alle sind Beispiele einer Kreisflächenberechnung

- Wie generalisieren wir?
  - Wie drücken wir die Idee von “Fläche eines Kreises” aus?

Definition neuer Prozeduren

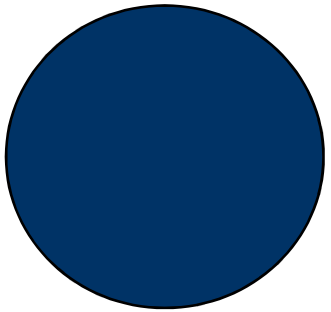
- Um wiederholende Muster festzuhalten, benutzen wir **Prozeduren**
  - Analog zu einer Funktionsdefinition in der Mathematik
  - Deshalb verwenden wir auch manchmal den Namen Funktion
- Die **`define` Sonderform** wird benutzt, um neue Prozeduren zu erstellen/definieren



Definition neuer Prozeduren

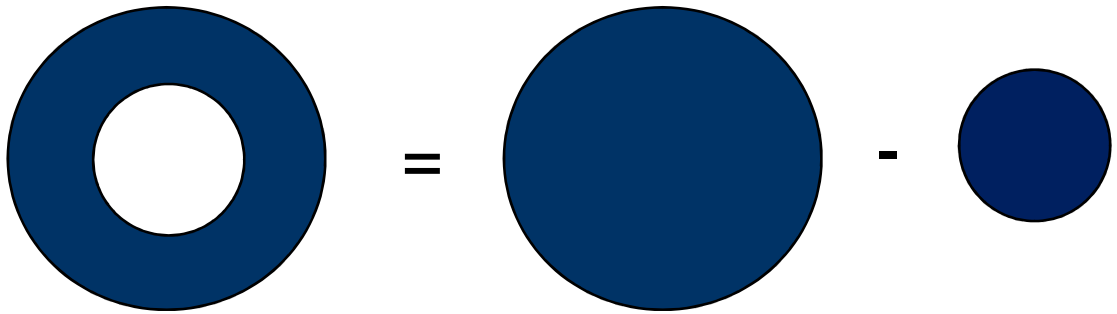
- Sobald eine Prozedur definiert wurde, kann sie wie eine primitive Prozedur (wie +, * etc.) benutzt werden
  - Kreisfläche:

```
(area-of-disk 5)
  → ergibt 78.5
```



  - Fläche eines Rings:

```
(- (area-of-disk 5) (area-of-disk 3))
  → ergibt (78.5 - 28.26) = 50.24
```

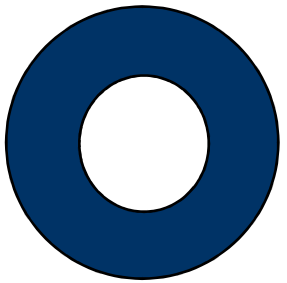


Definition neuer Prozeduren

- Bereits definierte Prozeduren können zu neuen, mächtigeren Prozeduren kombiniert werden

  - Berechnen der Fläche eines Rings

```
(define (area-of-ring outer inner)
  (- (area-of-disk outer)
     (area-of-disk inner)))
```



  - Anwendung der neuen Prozedur

```
  (area-of-ring 5 3)
= (- (area-of-disk 5) (area-of-disk 3))
= (- (* 3.14 (* 5 5))   (* 3.14 (* 3 3)))
= … = 50.24
```

Informelle Beschreibungen

- Typische Programmspezifikation
  - Üblicherweise nicht in Form mathematischer Ausdrücke, die in Programme umzuwandeln sind
  - Sondern informelle Problembeschreibungen
    - Enthalten irrelevante oder mehrdeutige Informationen
- Beispiel:

"Die Firma XYZ bezahlt ihren Angestellten 12 € pro Stunde. Ein Angestellter arbeitet zwischen 20 und 65 Stunden pro Woche. Entwickeln Sie ein Programm, welches den Lohn eines Angestellten aus der Anzahl der Arbeitsstunden berechnet"

Problemanalyse

```
(define (lohn stundenanzahl)
  (* 12 stundenanzahl))
```

Fehler

- Ihre Programme werden Fehler enthalten
  - Das ist nichts Schlimmes
  - Lassen Sie sich durch Fehler nicht verwirren/entmutigen
- Mögliche Fehler:
  - Keine korrekte Klammerung, z.B. `(* 3 (5)`
  - Operator eines Prozeduraufrufs ist keine Prozedur, z.B.

    ```
    (10)
    (10 + 20)
    ```
  - Typfehler, andere Laufzeitfehler, z.B.

    ```
    (+ 3 true)
    (/ 3 0)
    ```
- Probieren Sie aus, was bei fehlerhaften Eingaben passiert und versuchen Sie die Fehlermeldungen zu verstehen!

Design von Programmen

- Das Design von Programmen ist nicht trivial
- Folgendes Rezept soll Ihnen helfen, Ihre ersten Programme zu schreiben
  - Schritt-für-Schritt Beschreibung, was zu tun ist
  - Später wird dieses Rezept verfeinert

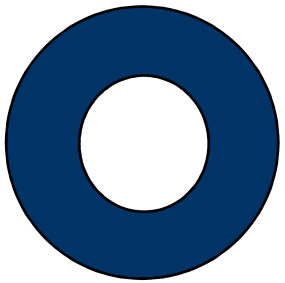
Jede Programmentwicklung besteht aus wenigstens vier Aktivitäten:

1. Verstehen, was der Zweck des Programms ist
2. Programmbeispiele ausdenken
3. Den Programmkörper implementieren
4. Testen

Design von Programmen

“If you can't write it down in English, you can't code it.“
Peter Halpern

- Verstehen, was der Zweck des Programms ist
  - Berechnung der Fläche eines Rings:



  - Berechnung der Fläche eines Ringes mit einem äußeren Radius ‘outer’ und einem inneren Radius ‘inner’
  - Berechnen kann man das durch die Fläche des Kreises mit Radius ‘outer’ *minus* die Fläche des Kreises mit Radius ‘inner’
  - …

Design von Programmen

1. Verstehen, was der Zweck des Programms ist
   - Vergabe eines sinnvollen Namens
   - Definition eines Vertrags
     - Welche Daten werden konsumiert und produziert?
   - Formulierung einer kurzen Beschreibung des Sinns des Programms
   - Hinzufügen des Programmkopfes

   Beispiel:

```
;; area-of-ring :: number number -> number
;; Berechnet die Flaeche eines Rings
;; mit Radius "outer", dessen
;; Loch den Radius "inner" hat
(define (area-of-ring outer inner) … )
```

Design von Programmen

2. Programmbeispiele ausdenken

- Hilft die Ein-/Ausgaberelation zu charakterisieren
- Es ist oft einfacher, Abstraktes anhand von Beispielen zu verstehen
- Beispiele helfen, den Berechnungsprozess eines Programms zu verstehen und logische Fehler zu entdecken

In unserem Beispiel:

```
;; area-of-ring :: number number -> number
;; Berechnet die Flaeche eines Rings…
;; dessen Loch den Radius "inner" hat

;; Beispiel: (area-of-ring 5 2) ergibt 65.94

(define (area-of-ring outer inner) … )
```

Design von Programmen

3. Den Programmkörper implementieren

- Ersetzung des "…" Platzhalters mit einem Ausdruck
- Ist die Eingabe-Ausgabe Relation eine mathematische Formel, können wir diese meist direkt übersetzen
- Bei einer informellen Beschreibung müssen wir uns den Berechnungsprozess klarmachen
  - Die Beispiele aus Schritt 2 können helfen

In unserem Beispiel:

```
;; area-of-ring :: number number -> number
;; Berechnet die Flaeche eines Rings…
;; dessen Loch den Radius "inner" hat

;; Beispiel: (area-of-ring 5 2) ergibt 65.94

(define (area-of-ring outer inner)
  (- (area-of-disk outer) (area-of-disk inner)))
```

Design von Programmen

- 4. Testen: Nutzen Sie die Funktion von Scheme!
- (check-expect *actual expected*)
  - Vergleicht den "actual"-Wert exakt mit dem "expected"-Wert.
  - Für Gleitkommazahlen problematisch, da Ergebnis nicht unbedingt exakt ist (markiert durch „#i“ vor dem Wert)
  - Beispiel: (check-expect (* 2 2) 4)
- (check-within *test expected delta*)
  - Prüfung, ob der Testwert (meist Gleitkommazahl) korrekt ist mit Abweichung delta, etwa delta = 0.0001.
  - Beispiel: (check-within (area-of-ring 5 2) 65.9 0.5)
- (check-error *test message*)
  - Prüft, ob der Aufruf die erwartete (durch (error „message“) ausgelöste) Fehlermeldung liefert
- Fehlgeschlagene Tests werden in einem separaten Fenster angezeigt.

"Testing can show the presence of bugs, but not their absence."

Edsger W. Dijkstra

"Beware of bugs in the above code; I have only proved it correct, not tried it"

Donald E. Knuth

Hilfsprozeduren

- Wann/wofür sollte man Hilfsprozeduren verwenden?
- Beispiel:

Schreiben Sie ein Programm, das den Profit eines Kinobesitzers in Abhängigkeit vom Ticketpreis berechnet.

- *Bei 5€ pro Ticket kommen 120 Leute.*
- *Pro 0,10€ Rabatt kommen 15 Leute mehr.*
- *Jede Aufführung kostet 180€.*
- *Jeder Teilnehmer kostet 0,04€.*

Hilfsprozeduren: Schlechtes Design

```
;; How NOT to design a program
(define (profit price)
  (- (* (+ 120
           (* (/ 15 .10)
              (- 5.00 price)))
        price)
     (+ 180
        (* .04
           (+ 120
              (* (/ 15 .10)
                 (- 5.00 price)))))))
```

Hilfsprozeduren: Gutes Design

```
;; How to design a program
(define (profit ticket-price)
  (- (revenue ticket-price)
     (cost ticket-price)))

(define (revenue ticket-price)
  (*  (attendees ticket-price) ticket-price))

(define (cost ticket-price)
  (+ 180
     (* .04 (attendees ticket-price))))

(define (attendees ticket-price)
  (+ 120
     (* (/ 15 .10) (- 5.00 ticket-price))))
```

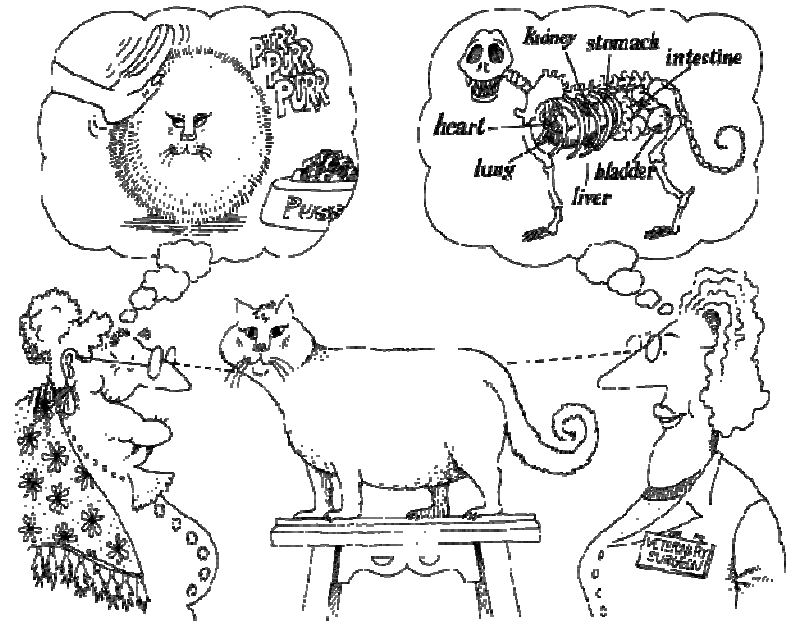
Daumenregel für Hilfsprozeduren

Definieren Sie Hilfsprozeduren für jede Abhängigkeit zwischen

- Quantitäten der Problembeschreibung oder
- Quantitäten, die bei den Beispielberechnungen entdeckt wurden.

Prozeduren als Black-Box-Abstraktionen

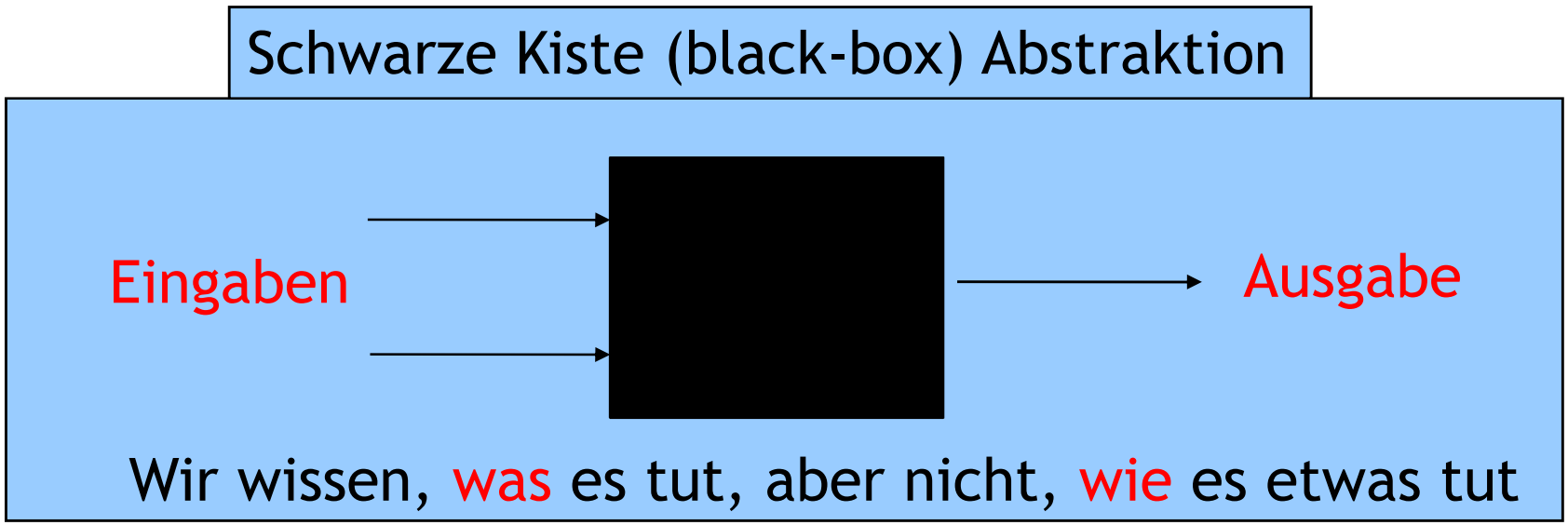
- Abstraktion dient dazu, Komplexität zu verstecken
- Details werden versteckt, die nicht zum Verständnis des Sachverhalts beitragen (abhängig vom Betrachter…)



Abstraction focuses upon the essential characteristics of some object, relative to the porspective of the viewer.

Prozeduren als Black-Box-Abstraktionen

- In der Vorlesung werden wir mehrere fundamentale Arten der Abstraktion kennen lernen
- Eine **prozedurale Abstraktion** ist eine davon:
  - **area-of-ring** berechnet die Fläche eines Rings
  - Der Benutzer muss sich nicht um die Details von **area-of-ring** kümmern



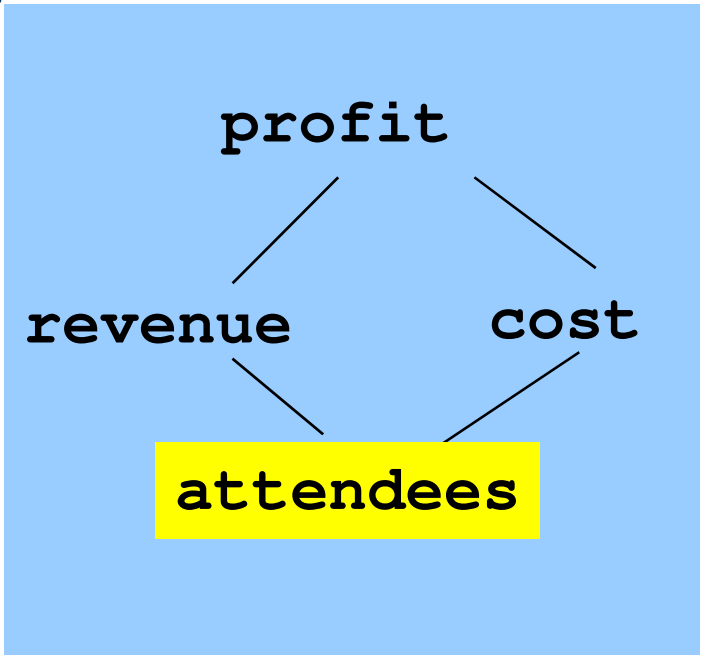
Prozeduren als Black-Box-Abstraktionen

- Ein Berechnungsproblem wird oft in natürliche, kleinere Teilprobleme aufgeteilt.
  - **Beispiel:**
    - Ringfläche → Zweifache Berechnung einer Kreisfläche
  - Für Teilprobleme werden Prozeduren geschrieben
    - **area-of-disk**, … primitive Prozeduren …

Prozeduren als Black-Box-Abstraktionen

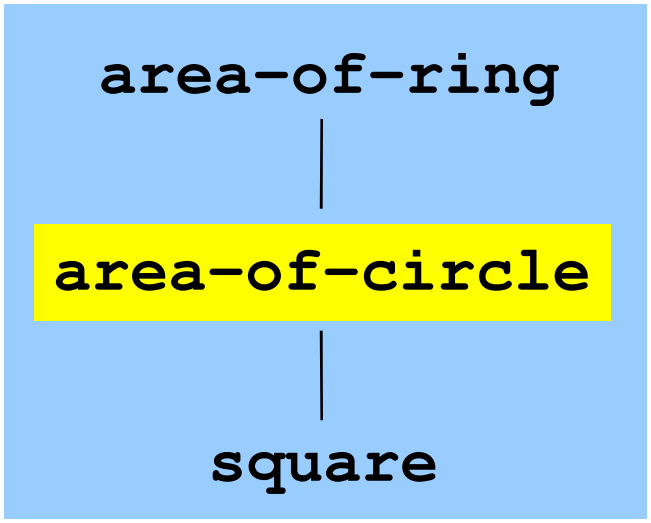
- Die Prozedur **`attendees`** kann als eine ‚black-box' betrachtet werden.
  - Sie soll die Anzahl der Teilnehmer berechnen.
  - Wir sind nicht interessiert zu wissen, wie sie die Berechnung durchführt.
  - Diese Details können vernachlässigt werden.

- **`attendees`** ist eine prozedurale Abstraktion für **`revenue/cost`**.



Prozeduren als Black-Box-Abstraktionen

- Eine benutzerdefinierte Prozedur wird über den Namen aufgerufen, genauso wie die primitiven Prozeduren.
- Wie die Prozedur arbeitet, bleibt versteckt.



Auf diesem Abstraktionsniveau ist jede Prozedur, die Quadrate berechnet, so gut wie jede andere.

```
(define (square x) (* x x))
```

```
(define (square x) (* (* x 10) (/ x 10)))
```

Konstantendefinitionen

Richtlinie für Variablendefinitionen:

Taucht eine Konstante häufig in einem Programm auf, sollten wir ihr einen Namen geben.

- Bessere Lesbarkeit
- Bessere Wartbarkeit
  - Bei Änderungen muss nur an einer Stelle etwas geändert werden
- In unserem Beispiel:

```
(define PI 3.14)
```

- Benötigt man eine bessere Approximation, ist nur eine Änderung erforderlich

```
(define PI 3.14159)
```

Bedingte Ausdrücke

Zwei Formen:

1) `(if <test>`
`<then-expr>`
`<else-expr>)` nicht optional in Scheme

Beispiel:

```
(define (absolute x)
   (if (< x 0)
       (- x)
       x))
```

Bedingte Ausdrücke

Zwei Formen:

2) `(cond [<test1> <expr1>]`
`[<test2> <expr2>]`
`. . .`
`[else <last-expr>])` optional

Beispiel:

```
(define (absolute x)
   (cond [(> x 0) x]
         [(= x 0) 0]
         [else (- x)]))
```

Boolesche Funktionen

`(and <expr_1> <expr_2> . . . <expr_N>)`

- **`<expr_i>`** **`(i = 1..N)`** werden in dieser Reihenfolge ausgewertet
- Es wird **`false`** zurück gegeben, falls irgendein Ausdruck als **`false`** ausgewertet wird, sonst wird **`true`** zurückgegeben.
- Wenn einer der Ausdrücke nicht **`true`** oder **`false`** ergibt, gibt es einen Fehler.
- Durch Shortcut-Regel werden manche Ausdrücke nicht ausgewertet.

```
(and (= 4 4) (< 5 3))   → false
(and true (+ 3 5)) → Fehler: and: question result is not true or false: 8
(and false (+ 3 5)) → Shortcut-Regel: false
```

Boolesche Funktionen

`(or <expr1_> <expr_2> . . . <expr_N>)`

- **`<expr_i> (i = 1..N)`** werden in dieser Reihenfolge ausgewertet.
- Es wird **`true`** nach dem ersten Wert zurückgegeben, der zu **`true`** ausgewertet wird.
- Es wird **`false`** zurückgegeben, falls alle Ausdrücke als **`false`** ausgewertet werden.
- Wird ein Wert ausgewertet und ergibt nicht **`true`** oder **`false`**, gibt es einen Fehler

Boolesche Funktionen

(boolean=? <expr1> <expr2>)

- **expr1, expr2** werden in dieser Reihenfolge ausgewertet.
- Es wird **true** zurückgegeben, falls **expr1** und **expr2** beide **true** ergeben oder beide **false** ergeben
- Es wird **false** zurückgegeben, falls die Operanden einen unterschiedlichen booleschen Wert haben
- Wird ein Operand ausgewertet und ergibt nicht **true** oder **false**, gibt es einen Fehler

(not <expr>)

- Gibt **true** zurück, wenn **<expr>** **false**
- Gibt **false** zurück, wenn **<expr>** **true**

Design konditionaler Prozeduren

- Wie ändert sich unser Designprozess?

Jede Programmentwicklung besteht aus wenigstens vier Aktivitäten
1. Verstehen, was der Zweck des Programms ist
2. Programmbeispiele ausdenken
3. Den Programmkörper implementieren
4. Testen

- Neue Phase: Datenanalyse
  - Welche unterschiedlichen Situationen gibt es?
- Beispiele
  - Mindestens ein Beispiel für jede Situation
- Implementierung des Programmkörpers
  - Erst Skelett der cond/if Ausdrücke definieren, dann die einzelnen Fälle implementieren
- Testen
  - Tests sollten alle Situationen abdecken

Symbole

- Bis jetzt hatten wir Zahlen und Booleans als primitive Werte kennengelernt
- Oft wollen wir symbolische Informationen speichern
  - Namen, Wörter, Richtungen
- Ein Symbol in Scheme ist eine Sequenz von Zeichen, angeführt von einem einfachen Anführungszeichen:
  - `'the 'dog 'ate 'a 'cat! 'two^3 'and%so%on?`
  - Nicht alle Zeichen sind erlaubt (z.B. keine Leerzeichen)
- Nur eine Operation auf diesem Datentyp: **`symbol=?`**
  - `(symbol=? 'Hallo 'Hallo)` → `true`
  - `(symbol=? 'Hallo 'ABC)` → `false`
  - `(symbol=? 1 2)` → `Fehler`
- Symbole sind atomar (wie auch Zahlen, Booleans)
  - Symbole können nicht zerlegt werden

Symbole: Beispiel

```
(define (reply s)
  (cond
    [(symbol=? s 'GoodMorning) 'Hi]
    [(symbol=? s 'HowAreYou?) 'Fine]
    [(symbol=? s 'GoodAfternoon) 'INeedANap]
    [(symbol=? s 'GoodEvening) 'BoyAmITired]
    [else 'Error_in_reply:unknown_case] ))
```

Symbole vs. Strings

- Viele von Ihnen kennen vielleicht den Datentyp **String**
- Symbole sind zu unterscheiden von Strings
  - Symbole: Werden benutzt für **symbolische Namen**
    - Atomar
    - Keine Manipulation
    - Sehr effizienter Vergleich
    - Gewisse Einschränkungen, welche Zeichen dargestellt werden können
  - Strings: Werden benutzt für **Textdaten**
    - Manipulation möglich
      - (z.B. suchen, zusammensetzen etc.)
    - Vergleiche sind teuer
    - Beliebige Zeichen(ketten) darstellbar
  - Strings sind auch in Scheme verfügbar
    - Mit doppelten Anführungszeichen zu erzeugen
    - Mit `string=?` zu vergleichen
  - Wir werden Strings zunächst ignorieren

Erinnerung: Die Auswertungsregel

Bisher hatten wir nur eingebaute Operatoren als Prozeduren. Wie werden Prozeduren ausgewertet, die vom Programmierer definiert sind?

- selbst-auswertend → …
- eingebauter Operator → …
- Name → …
- Sonderform → …
- Kombination →
  i. *Rechne die Unterausdrücke* (in beliebiger Reihenfolge) *aus*
  ii. *Wende die Prozedur, die der Wert des am weitesten links liegenden Unterausdruckes ist (der Operator), auf die Argumente an – die Werte der restlichen Unterausdrücke (Operanden).*

Auswertungsregel erweitert

- Auswertungsregel 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)
```

Das Substitutionsmodell

- **Fiktive Namen** (formale Parameter/Variablen):
  - Erlauben die Definition von allgemeinen Prozeduren, die in unterschiedlichen Situationen wiederverwendet werden können.
- Um die Prozedur auszuwerten, müssen effektive Werte an die fiktiven Namen gebunden werden.
- Auch bekannt von arithmetischen Formeln:

$f(a, b) = a^2 + b^2$

Deklaration der formalen Parameter

Definition der Funktion mit Hilfe von formalen Parametern

f (3, 2)

Benutzung der allgemeinen Funktion, um ein Spezialproblem zu lösen.

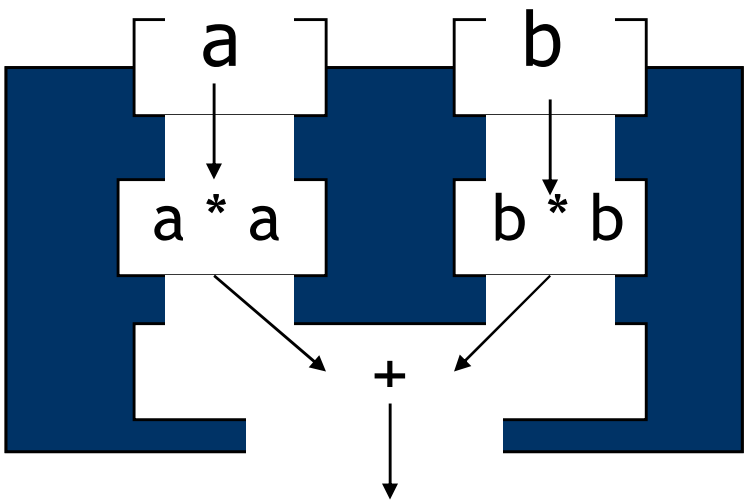
Das Substitutionsmodell

Substitution der fiktiven Variablen mit effektiven Werten während der Ausführung

```
(define b 2)
... (f 3 2) ...
... (f b 2) ...
...
```

Aufrufumgebung

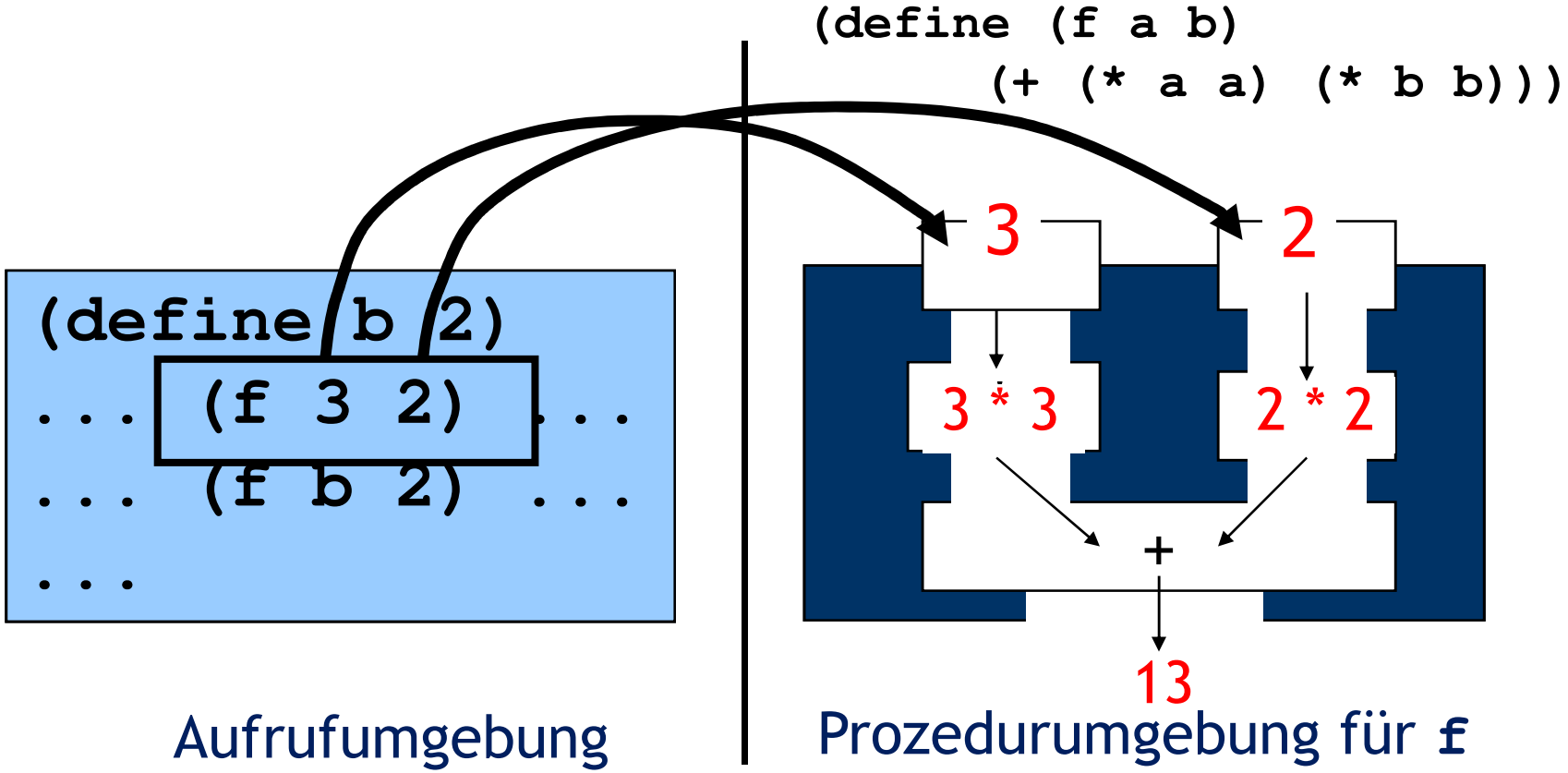
```
(define (f a b)
        (+ (* a a) (* b b)))
```



Prozedurumgebung für `f`

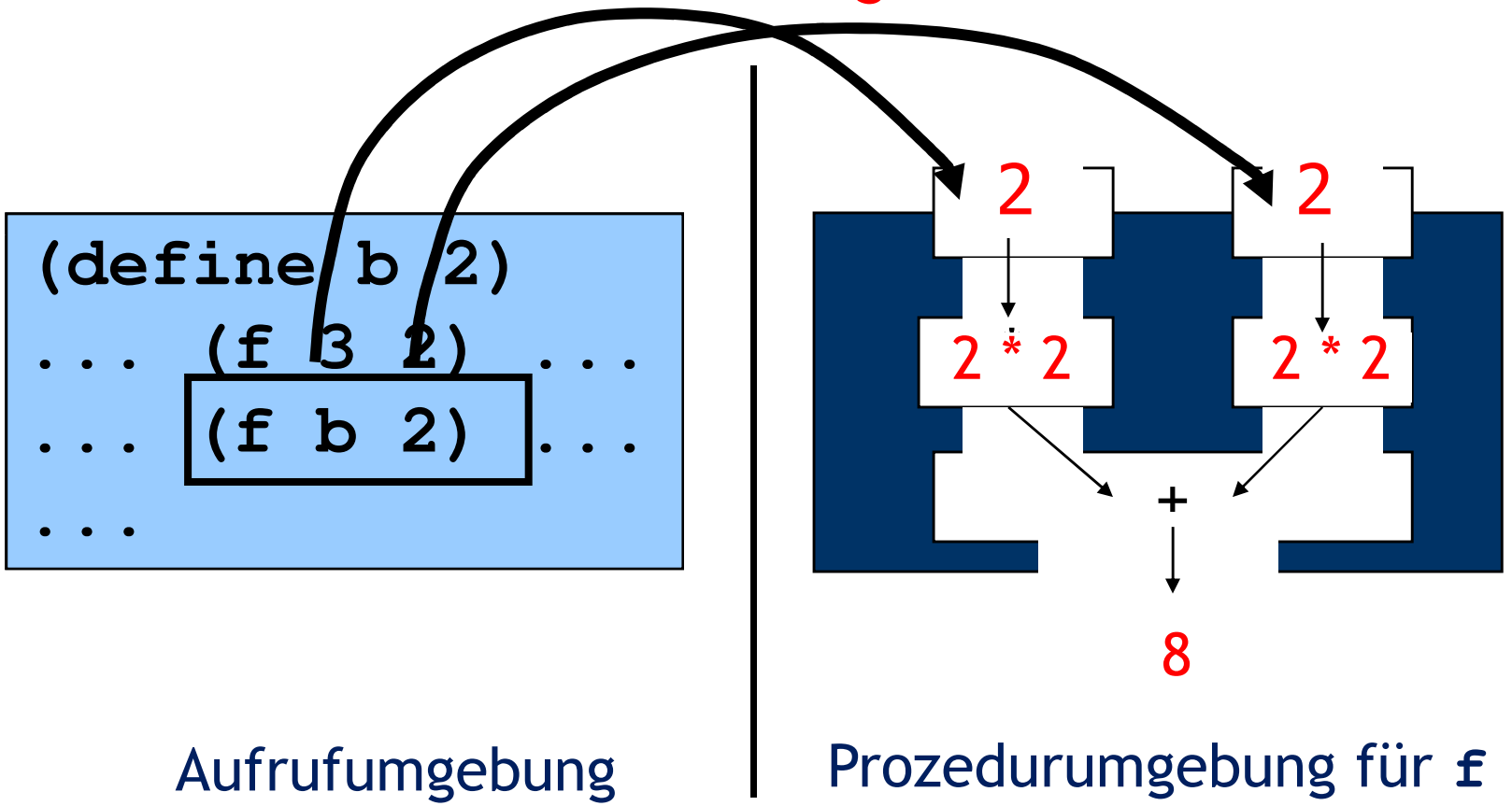
Das Substitutionsmodell

Substitution der fiktiven Variablen mit effektiven Werten während der Ausführung



Das Substitutionsmodell

Substitution der fiktiven Variablen mit effektiven Werten während der Ausführung



Das Substitutionsmodell

- Das Substitutionsmodell soll Ihnen helfen, über die **Bedeutung** (Semantik) eines Programms nachzudenken
  - Es gibt keine Auskunft darüber, wie der Interpreter tatsächlich arbeitet.
  - Typischerweise wertet ein Interpreter nicht den Prozeduraufruf durch textuelle Manipulation des Rumpfs aus.

- Einfaches Modell, um formal über den Auswertungsprozess nachzudenken.
  - Es erlaubt Ihnen, Programme auf einem Stück Papier auszuführen
  - Detailliertere Modelle folgen später

Details des Substitutionsmodells: Applikative Reihenfolge

```
(define (square x) (* x x)))
(define (average x y) (/ (+ x y) 2))

(average 5 (square 3))
(average 5 (* 3 3))
(average 5 9)
```

Werte zuerst die Operanden aus, führe dann die Ersetzung durch (applikative Reihenfolge)

```
(/ (+ 5 9) 2)
(/ 14 2)
```

Falls der Operator eine einfache Prozedur ist, ersetze diesen mit dem Resultat der Operation

```
7
```

Details des Substitutionsmodells: Normale Reihenfolge

```
(define (square x) (* x x))
(define (average x y) (/ (+ x y) 2))

(average 5 (square 3))
(/ (+ 5 (square 3)) 2)
(/ (+ 5 (* 3 3)) 2)
(/ (+ 5 9) 2)
(/ 14 2)
7
```

Normale Reihenfolge

Applikative vs. normale Auswertungsreihenfolge

- *Applikativ*: Erst Operator und alle Operanden auswerten, dann substituieren
- *Normal*: Operator auswerten, dann die (unausgewerteten) Operanden für die formalen Argumente des Operators substituieren

- Wichtige, nicht-triviale Eigenschaft: Das Ergebnis hängt nicht von der Auswertungsreihenfolge ab (**Konfluenz**)
  - Allerdings kann es sein, dass bei ungeschickter Auswertungsreihenfolge die Auswertung nie terminiert
  - Wir werden später Sprachfeatures (Zuweisungen, Ein-/Ausgabe) betrachten, die diese Eigenschaft zerstören
  - Die beiden Strategien können sich erheblich in der Anzahl der benötigten Schritte unterscheiden
    - Argument wird nicht benötigt → normale Reihenfolge besser
    - Argument wird mehrfach benötigt → applikative Reihenfolge besser

Rückblick: Die Sprache Scheme

- **Dinge, die ein Scheme Programm ausmachen:**
  - selbst auswertende `23, true, false`
  - Namen `+, PI, pi`
  - Kombinationen `(+ 2 3) (* pi 4)`
  - Sonderformen `(define PI 3.14)`
- **Syntax**
  - Kombination: `(oper-expression other-expressions …)`
  - Sonderform: Ein besonderes Schlüsselwort als erster Unterausdruck
- **Semantik**
  - Kombinationen: Werte Unterausdrücke in beliebiger Reihenfolge aus.
    Wende den Operator auf die Operanden an
    Substitution bei benutzerdefinierten Prozeduren
  - Sonderformen: Jede hat eine eigene Semantik

Zusammenfassung

Wir haben kennengelernt:

- Die einfachsten Elemente (Daten/Prozeduren) von Scheme
- Kombination als ein Mittel der Zusammensetzung von einfacheren Elementen in komplexeren Elementen
- Benennen von Kombinationen, um sie weiter als Elemente in andere Kombinationen einzusetzen
- Eigene Prozeduren definieren – Prozessschablone – und sie als elementare Elemente von Kombinationen einsetzen

_____ Scheme _____

- Probleme in kleinere klar umrissene Aufgaben zerlegen
- Prozeduren als Black-Box Abstraktionen
- Semantik eines Prozeduraufrufs als Substitutionsprozess