Struktur und Implementation von Computerprogrammen

Nicolas Neuß
Institut für Angewandte und Numerische Mathematik
Universität Karlsruhe
Englerstraße 2, 76128 Karlsruhe
neuss@math.uni-karlsruhe.de

Erstellt: 17. Juli 2008

URL für die Vorlesung:
http://www.mathematik.uni-karlsruhe.de/ianm3/lehre/compprog2008s/

Inhaltsverzeichnis

1 Einführung

1.1 Etwas Geschichte

- Charles Babbage (ca. 1830), Ada Lovelace (ca. 1840), Alonzo Church (1936), Alan Turing (1936), Konrad Zuse (ca. 1940), John von Neumann (1945)
- Programmiersprachen:
  - Lambda-Kalkül (Church, 1936)
  - Plankalkül (Zuse, 1942-1946)
  - Maschinensprachen (1940-heute)
  - Hochsprachen: Fortran (Backus, ...(IBM), 1954-heute), Lisp (McCarthy, ..., 1956-heute), Algol (verschiedene Autoren, 1960-...), ...
- Scheme (Steele&Sussman 1975): Mitglied der Lisp-Sprachfamilie

1.2 Programmierparadigmen

- Imperative Programmierung (unstrukturiert, strukturiert, prozedural, modular)
- Funktionale Programmierung
- Objektorientierte Programmierung
- Datenstromorientierte Programmierung
- Logische Programmierung, Constraintprogrammierung
- Reaktive Programmierung
- Aspektorientierte Programmierung
- ...

1.3 Software Engineering

Programmieren ist inhärent schwierig, und so ist die Software-Welt voll von Schlagworten, die eine „Silberkugel“ (gegen den Komplexitätswerwolf) sein möchten. Zum Beispiel:

- Wasserfallmodell

- Agile Software-Entwicklung: XP (Extreme Programming, Pair programming)
- Rational Unified Process, UML
- ...

Bemerkung: Obwohl diese Techniken teilweise schon in der Schule unterrichtet werden, nutzen sie wenig ohne eine erhebliche Erfahrung im Programmieren unter verschiedenen Paradigmen.

1.4 Philosophie dieses Kurses

- Wir lernen *Programmieren*
- in verschiedenen *Paradigmen*
- mit der *einfachstmöglichen* Programmiersprache,
- die aber gleichzeitig eine der *flexibelsten* und *ausdrucksstärksten* ist,
- so dass wir unsere Sprache *beliebig ausbauen* können.

1.5 Hello, World!

Der Vergleich des bekannten "Hello, World"-Programms in Java und Scheme/Lisp ist bezeichnend. (Man beachte, dass Java heutzutage oft für die Anfängerausbildung eingesetzt wird!)

Programm: (hello-world.java)

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Programm: (hello-world.scheme)

```
"Hello, World!"
```

1.6 Interpreter und Compiler

Definition: Ein Interpreter liest die Befehle eines in einer Hochsprache geschriebenen Programms und führt sie durch Aufruf passender (in Maschinensprache geschriebener) Routinen aus.

Definition: Ein Compiler übersetzt das in einer Hochsprache geschriebene Programm in Maschinensprache, die dann ausgeführt werden kann.

Bemerkungen:

- Die Grenze ist fließend, z.B. Bytecode- oder Just-in-Time-Kompilierung.
- Die Unterscheidung Compiler/Interpreter ist orthogonal zur Unterscheidung interaktiv/nicht interaktiv (z.B. SBCL=interaktiver Common-Lisp-Compiler).

1.7 Geplantes Programm

- Funktionale Programmierung
  - Auswertungsregeln
  - Newton-Verfahren
  - Kodierung
  - Verschlüsselung
  - Symbolisches Rechnen
- Objektorientierte Programmierung
  - Message-passing
  - Generische Funktionen
- Der Lambda-Kalkül
- Reaktive Programmierung
  - Schaltkreissimulation
  - Parallele Prozesse
  - Datenströme
- Metaprogrammierung
  - Scheme in Scheme
  - Logikprogrammierung

1.8 Referenzen

[1] H. Abelson, G. Sussman: *Structure and Interpretation of Computer Programs*, MIT Press

[2] `http://mitpress.mit.edu/sicp`

[3] `http://www.plt-scheme.org` (PLT, DrScheme)

2 Abstraktion mit Funktionen

2.1 Programmelemente

Jede höhere Programmiersprache hat

- Primitive Ausdrücke
- Kombinationsmöglichkeiten zur Erzeugung zusammengesetzter Ausdrücke
- Abstraktionsmöglichkeiten, durch die zusammengesetzte Objekte benannt und als neue Einheiten behandelt werden können.

Frage: Wie sehen diese Elemente in Scheme aus?

2.2 Ausdrücke

Es gibt in Scheme folgende Ausdrücke (*expressions*):

- Primitive Ausdrücke (*primitive expressions*)
  - Zahlen, Zeichenketten, ...
  - Symbole, die als Bezeichner für ein Objekt dienen (Variablen)
- Zusammengesetzte Ausdrücke (*compound expressions*) der Form

  ```
  (operator operand-1 ... operand-n)
  ```

  Man unterscheidet:
  - Funktionsauswertungen, z.B. `(+ 42 1)`
  - Sonderformen, z.B. `(define pi 3.1415926)`

Bemerkungen:

- Scheme hat ein sehr ausgefeiltes Zahlsystem. Dies beinhaltet
  - Ganze Zahlen beliebiger Größe: 1,-2,132412340887987, ...
  - Rationale Zahlen, z.B. 3/4
  - Komplexe Zahlen, z.B 1-2i
  - Gleitkommazahlen, z.B. 2.7182818

- Die Positionierung des Operators an erster Stelle nennt man Präfix-Notation. Sie ist auch in der Mathematik und anderen Computersprachen vorherrschend (nur für Arithmetik wird dort meist die Infix-Notation verwendet).
- Jeder Unterausdruck eines zusammengesetzten Ausdrucks kann wieder zusammengesetzt sein, z.B.

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

- Um zusammengesetzte Ausdrücke lesbar zu machen, sollten sie geeignet indentiert werden, z.B.

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

- Zur sinnvollen Lisp-Programmierung braucht man daher unbedingt einen Editor, der bei Indentierung und Klammerung Hilfestellung leistet. (Bei anderen Computersprachen ist das auch wichtig, aber nicht essentiell.)

2.2.1 Funktionsauswertungen

Wenn <operator> kein Symbol ist, welches eine Sonderform einleitet, so wird der zusammengesetzte Ausdruck

```
(<operator> <operand-1> ... <operand-n>)
```

immer folgendermaßen ausgewertet:

> a) Werte alle Unterausdrücke <operator>, <operand-1>, ... aus. Die Werte der Operanden nennt man auch Argumente der Funktion.
>
> b) Wende den Wert des Operator-Ausdrucks auf die Argumente an!

Beispiele: `(+ 2 3 4)`, `(* (+ 2 3) (/ 5 6))`, ...

Bemerkung: Die obige Regel ist *rekursiv*! Ein Ausdruck kann daher eine Baumstruktur bilden, die rekursiv abgearbeitet wird.

2.2.2 Das Bezeichnen von Objekten

Die Einführung neuer Namen für irgendwelche Werte geschieht in Scheme mit

```
(define <name> <expression>)
```

Dies führt dazu, dass man im weiteren Programm den Wert von <expression> durch die Angabe von <name> erhalten kann.

Beispiel:

```
(define pi 3.1415926)
(define r 0.1)
(* pi r r)
```

Sprechweise: Der Variablen <name> wird der Wert von <expression> zugewiesen.

Bemerkung:

- (define ...) ist *keine Funktionsauswertung*, weil der zweite Operand (nämlich der Name) *nicht ausgewertet* wird!
- Man nennt (define ...) daher eine Sonderform.
- Sonderformen sollte man als Bestandteil der Syntax einer Programmiersprache verstehen.

2.2.3 Die Definition neuer Funktionen

Mit einer besonderen Form von define können auch neue Funktionen definiert und bezeichnet werden:

```
(define (<name> <formale-parameter>)
   <funktionscode>)
```

Sprechweise: Wir nennen solche selbstdefinierten Funktionen auch zusammengesetzte Funktionen (*compound procedures*).

Beispiel: Quadrieren

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

Auswertung zusammengesetzter Funktionen Zusammengesetzte Funktionen werden ausgewertet, indem der Funktionsaufruf (<name> <arg1> ... <argn>) durch eine Version von <funktionscode> ersetzt wird, bei der die formalen Parameter durch die entsprechenden Argumente ersetzt worden sind.

Beispiel: Ein Aufruf (square (+ 3 5)) wird wie folgt ausgewertet:

```
(square (+ 3 5)) => (square 8) => (* 8 8) => 64
```

Bemerkungen:

- Zusammengesetzte Funktionen wie square sind von ihrer Anwendung her nicht von eingebauten Funktionen (wie etwa +) unterscheidbar.

- Das Konstrukt `(define (<name> <formal-parameters>) <code>)` ist eigentlich nur syntaktischer Zucker für das einfache

  ```
  (define <name> (lambda (<formal-parameters>) <code>))
  ```

  (Die Sonderform `lambda` erzeugt dabei Funktionsobjekte.)

2.2.4 Bedingte Ausdrücke

Der einfachste bedingte Ausdruck ist durch die Sonderform `if` gegeben:

```
(if <bedingung>
    <dann-ausdruck>
    <sonst-ausdruck>)
```

Die Auswerteregel ist dabei, dass `<bedingung>` ausgewertet wird, und je nach dem Ergebnis (`#t` für wahr und `#f` für falsch) der gesamte `if`-Ausdruck entweder durch `<dann-ausdruck>` oder `<sonst-ausdruck>` ersetzt wird.

Beispiel:

```
(if (< 2 0 1)
    "ansteigend"
    "nichtansteigend")
=> (if #f "ansteigend" "nichtansteigend")
=> "nichtansteigend"
```

Bemerkung: In Bedingungen darf in vielen Fällen nicht den Regeln der Funktionsauswertung gefolgt werden. Beispielsweise könnte

```
(if (= x 0)
    0
    (/ 1 x))
```

auf einen Fehler laufen, falls immer alle Operanden ausgewertet würden!

Definition: Die Modellierung der Arbeitsweise von Funktionen und Ausdrücken mittels Ersetzen (Substitution), was wir hier und bei der Definition der Funktionsauswertung kennengelernt haben, nennt man das Substitutionsmodell.

Bemerkung: Durch Syntaxtransformationen (später im Kapitel über den Lambda-Kalkül werden wir das genauer besprechen) kann man mit Hilfe von `if` verschiedene weitere nützliche Sonderformen definieren:

- Eine Bedingung mit mehreren Ausgängen (der erste wahre wird genommen):

  ```
  (cond (<predicate-1> <consequence-1>)
        ...
        (<predicate-n> <consequence-n>))
  ```

- `(and <predicate-1> ... <predicate-n>)`: nur wahr, wenn alle wahr sind.
- `(or <predicate-1> ... <predicate-n>)`: nur falsch, wenn alle falsch sind.

2.3 Verzögerte Auswertung

Erinnerung: In Scheme (so wie in den meisten anderen Programmiersprachen) werden die Operanden ausgewertet, bevor sie verwendet werden, also

```
(square (+ 3 5)) => (square 8) => (* 8 8) => 64
```

Beobachtung: Alternativ hätte man auch die formalen Parameter sofort durch die Operanden ersetzten könnnen und erst so spät wie möglich auswerten können:

```
(square (+ 3 5)) => (* (+ 3 5) (+ 3 5)) => (* 8 8) => 64
```

Dies nennt man verzögerte Auswertung (lazy evaluation).

Bemerkungen:

- Diese Art der Auswertung liefert zwar für die bisher betrachteten Programme dasselbe Ergebnis, braucht aber im Normalfall mehr Operationen.
- In Sprachen mit verzögerter Auswertung (z.B. Haskell) kann auch die if-Bedingung als Funktion definiert werden.

2.4 Übungen

Aufgabe 1: (0 Punkte)
Installieren Sie DrScheme und experimentieren Sie! Zum Beispiel mit den Ausdrücken:

```
1, (/ 3 5), (define a 3), a, (define b (+ a 1)), b,
(+ a b (* a b), (= a b), (if (> a b) a b), ...
```

HINWEIS: DrScheme bietet Sprachebenen (Anfänger, Fortgeschrittener, ...). Leider folgt die Einstufung einem anderen Buch, so dass für unser Programm am besten schon die Einstufung Fortgeschrittener (oder sogar Experte?) gewählt werden sollte.

Aufgabe 2: (1 Punkt)
Übersetzen Sie den mathematischen Ausdruck $\frac{1+2(3+4)}{(5+6)(7+8)}$ in (gut indentierte) Präfix-Form und berechnen Sie den Wert.

Aufgabe 3: (1 Punkt)
Schreiben Sie eine Funktion, die drei Argumente akzeptiert und die Summe der beiden größten berechnet.

Aufgabe 4: (1 Punkt)
Werten Sie aus:

```
(define (p) (p))
(define (test x y)
  (if (= x 0)
      0
      y))
(test 0 (p))
```

Was passiert? Warum? Was wäre das Ergebnis, wenn man einen Interpreter mit verzögerter Auswertung vor sich hätte?

Aufgabe 5: (1 Punkt)
Ist folgende Funktion äquivalent zu `if`?

```
(define (new-if predicate if-clause else-clause)
  (cond
    (predicate if-clause)
    (else else-clause)))
```

Aufgabe 6: (1 Punkt)
Geben Sie der folgenden Funktion einen passenderen Namen als `???`:

```
(define (??? x)
  ((if (< x 0) - +)
   x))
```

2.4.1 Quadratwurzelberechnung

Wissen: Die Berechnung der Quadratwurzel aus einer positiven Zahl $y \in \mathbb{R}$ kann mittels des Newton-Verfahrens durchgeführt werden. Die Iterationsvorschrift lautet:

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{y}{x_k}\right)$$

Satz: Die durch obige Iterationsvorschrift gegebene Folge (x_k) konvergiert für alle Startwerte $x_0 > 0$ gegen $\sqrt{y}$. Die Konvergenz ist sehr schnell (*quadratisch*) wenn x_k nahe bei $\sqrt{y}$ ist, ansonsten kann sie sehr langsam sein.

Programm: (square-root-1.scm)

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

(define (improve x y)
  (average x (/ y x)))

(define (average a b)
  (/ (+ a b) 2))
```

```
(define (iterate x y)
  (if (good-enough? x y)
      x
      (iterate (improve x y) y)))

(define (good-enough? x y)
  (< (abs (- y (* x x))) 0.000001))

(define (square-root y)
  (iterate 1.0 y))
```

Bemerkungen:

- Das Problem zerfällt auf natürliche Weise in Teilprobleme, die man im Programm wiederfindet.
- Die einzelnen Funktionen könnte man im Programm recht einfach durch andere Bausteine ersetzen. Z.B. könnte man `improve` um eine Dämpfungsstrategie erweitern oder durch ein Newton-Verfahren für allgemeinere Funktionen ersetzen.
- Die einzelnen Funktionen haben lange Namen, die gleichzeitig das Programm dokumentieren. Man beachte, dass Namen in Scheme/Lisp auch - und ? enthalten können (genauer: alle Zeichen außer Klammern und Leerzeichen).
- Der Startwert 1.0 garantiert eine Rechnung in Gleitkomma-Arithmetik.
- Der rekursive Aufruf von `iteration` ersetzt eine Schleife in einer anderen Sprache.

2.4.2 Prozeduren/Funktionen als Black-Box-Abstraktionen

Beobachtung: Von jeder Funktion ist jeweils nur der Funktionskopf für die Außenwelt interessant, die interne Arbeitsweise interessiert nicht! So würde es die Arbeitsweise des Programms nicht ändern, wenn `square` als

```
(define (square x)
  (exp (* 2 (log x))))
```

definiert worden wäre. Funktionen sind Black-Box-Abstraktionen.

2.4.3 Interne Namen

Beobachtung: Insbesondere hängt die Arbeitsweise einer Funktion nicht von den Namen der formalen Parameter ab. So hätte die Definition

```
(define (square y)
  (* y y))
```

denselben Effekt wie die Version mit `x`.

Bezeichnung: In einen Funktionsrumpf bezeichnet man eine Variable, die von einem formalen Parameter herrührt, als gebunden (*bound*), jede andere Variable als frei (*free*). Die Ausdrücke, für die eine Bindung Gültigkeit hat, bezeichnet man als Definitionsbereich (scope) dieser Variablendefinition.

Bemerkung: Allerdings dürfte man nicht `*` als formalen Parameter verwenden, weil diese „Variable“ schon definiert ist und innerhalb von `square` gebraucht wird. Ein Binden von freien Variablen bezeichnet man als capturing.

2.4.4 Interne Definitionen

Problem: Von unseren Funktionen interessiert letztendlich nur eine einzige, nämlich `square-root`. Dass andere Funktionen wie `iterate` und `good-enough?` ebenfalls sichtbar sind, ist ein Nachteil, weil es die Black-Box-Abstraktion durchbricht und die Arbeitsweise unserer Wurzelberechnung unnötig offengelegt.

Abhilfe: Es ist in Scheme möglich, innerhalb der Funktion weitere Definitionen einzuführen, welche Namen dort lokal binden und nach außen nicht sichtbar sind.

Programm: (square-root-2.scm)

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

(define (average a b)
  (/ (+ a b) 2))

(define eps 0.0000001)

(define (square-root y)
  ;; lokale Routinen
  (define (improve x)
    (average x (/ y x)))

  (define (good-enough? x)
    (< (abs (- y (* x x))) eps))

  (define (iterate x)
    (if (good-enough? x)
        x
        (iterate (improve x))))

  ;; Aufruf der Iteration
  (iterate 1.0))
```

Bemerkung:

- Diese Verschachtelung von Definitionen nennt man auch Blockstruktur. Sie wurde mit der Sprache Algol 60 eingeführt. Es gibt sie in vielen höheren Sprachen (in C/C++ allerdings nicht für Funktionsdefinitionen).
- Man beachte, dass Kommentare in Lisp/Scheme durch ein oder mehrere Semikolons eingeleitet werden und dann bis zum Ende der Zeile gehen. Man verwendet meist ein Semikolon nach Code in einer Zeile, zwei innerhalb einer Funktionsdefinition ohne vorherstehenden Code, und drei außerhalb von Funktionen.
- Man beachte auch, dass sich einige Funktionen vereinfacht haben, weil der Wert, für den die Wurzel berechnet wird, nicht mehr übergeben werden muss.
- In DrScheme kann man sich die Bindungen der Variablen durch Pfeile anzeigen lassen, was hier ganz hübsch ist.

2.5 Prozeduren und die zugehörigen Rechenprozesse

Frage: Die direkte Umsetzung der rekursiven Fakultätsdefinition $n! = n \cdot (n-1)!$ liefert folgenden Code:

```
(define (factorial n)
   (if (= n 0)
       1
       (* n (factorial (- n 1)))))
```

Was geschieht bei dessen Ausführung?

Antwort: Wenn man `(trace factorial)` eingibt, so erhält man folgende Ausgabe:

```
> (factorial 5)
|(factorial 5)
| (factorial 4)
| |(factorial 3)
| | (factorial 2)
| | |(factorial 1)
| | | (factorial 0)
| | | 1
| | |1
| | 2
| |6
| 24
|120
```

Dies bedeutet, dass bei der Auswertung eine Kette von wartenden Operationen $*$ entsteht, die Speicher verbraucht.

Alternative: Man kann $n!$ aber auch berechnen, indem die Produktbildung als Argument einer iterierenden Funktion auftritt:

```
(define (fact-iter n f)
  (if (= n 0)
      f
      (fact-iter (- n 1) (* n f))))
```

Beobachtung: Dies führt dann zu der Ausgabe

```
> (fact-iter 5 1)
|(fact-iter 5 1)
|(fact-iter 4 5)
|(fact-iter 3 20)
|(fact-iter 2 60)
|(fact-iter 1 120)
|(fact-iter 0 120)
|120
```

Wir sehen, dass hier keine Operationen warten müssen! Der Aufruf der Funktion an Endposition kann direkt geschehen und wird von Scheme optimiert (*tail-call optimisation*)!

Bezeichnung: Die erste Version nennt man einen rekursiven Rechenprozess, und weil der Speicher- und Rechenaufwand linear mit dem Argument wächst, einen linear rekursiven Prozess. Die zweite Version nennt man einen iterativen Prozess (bzw. newclinear iterativen Prozess).

Bemerkung:

- Auf Maschinensprachebene bedeutet die Optimierung des Endaufrufs im wesentlichen das Ersetzen eines Unterroutinenaufrufs durch einen Sprung.
- Die Optimierung von Endaufrufen wird von vielen anderen Sprachen gar nicht oder nur mit passenden Optimierungsparametern geliefert. Diese Sprachen besitzen dann spezielle Schleifenkonstrukte, welche eine ähnliche Funktionalität liefern.
- Scheme hat zwar auch spezielle Schleifenbefehle, diese sind aber wieder nur Syntaxtransformationen von solchen endrekursiven (*tail-recursive*) Funktionsanwendungen.

2.5.1 Baumrekursion

Aufgabe: Berechne die Fibonacci-Zahlen, die wie folgt definiert sind:

$$\mathrm{Fib}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & \text{sonst} \end{cases}$$

Programm:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))
```

Beobachtungen:

- Beim Aufruf dieser Funktion wird ein Baum von Aufrufen abgearbeitet. Man nennt diesen Prozess daher baumrekursiv.
- Der Rechenaufwand steigt exponentiell mit n an. Eine genauere Analyse zeigt ein Wachstum proportional zu $\mathrm{Fib}(n)$ selbst, was wiederum wie λ^n wächst, wobei $\lambda = \frac{1+\sqrt{5}}{2}$ der goldene ist.
- Wie die Darstellung mittels `trace` zeigt, geschehen viele Aufrufe in diesem Baum mehrfach.

Alternative: Auch diese Funktion kann man iterativ formulieren:

```
(define (fib-iter n a b)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (fib-iter (- n 1) b (+ a b)))))
(define fib (n)
   (fib-iter n 0 1))
```

Der zugehörige Rechenprozess ist linear iterativ und viel schneller als der baumrekursive.

2.5.2 Übungen

Aufgabe 7: (2 Punkte)
Sei `inc` eine Funktion, die ihr Argument um 1 erhöht und `dec` eine Funktion, die ihr Argument um 1 erniedrigt. Angenommen, dass `inc` und `dec` Operationen mit Aufwand O(1) sind: welche Art von Prozess erzeugen dann die beiden folgenden Funktionen?

```
(define (+ a b)
  (if (= a 0)
     b
     (inc (+ (dec a) b))))

(define (+ a b)
  (if (= a 0)
     b
     (+ (dec a) (inc b))))
```

Aufgabe 8: (2 Punkte)
Überprüfen Sie mit Hilfe des `time`-Befehls die Geschwindigkeit von rekursiver und iterativer Fakultätsberechnung. Ist die iterative Berechnung erheblich schneller? Warum (nicht)?

Aufgabe 9: (4 Punkte)
Die folgende Scheme-Prozedur berechnet die sogenannte „Ackermann-Funktion" $A(x, y)$:

```
(define (A x y)
  (cond ((= y 0) 0)
        ((= y 1) 2)
        ((= x 0) (* 2 y))
        (else (A (- x 1)
                 (A x (- y 1))))))
```

a) Welches sind die Werte von (A 1 10), (A 2 4), (A 3 3)?

b) Geben Sie äquivalente, gebräuchlichere Darstellungen von folgenden Funktionen an (ausprobieren reicht, beweisen ist besser):

i) $f(n) = A(0, n)$
ii) $g(n) = A(1, n)$
iii) $h(n) = A(2, n)$

Aufgabe 10: (4 Punkte)
Schreiben Sie eine Funktion die zu gegebenen natürlichen Zahlen n und $0 \leq k \leq n$ den zugehörigen Wert des „Binomialkoeffizienten" $B(n, k)$ berechnet. Es gilt:

a) $B(n, 0) = B(n, n) = 1$ für alle $n \geq 0$.

b) $B(n, k) = B(n-1, k-1) + B(n-1, k)$ für $n \geq 0, 1 \leq k \leq n-1$.

2.5.3 Wechselgeld

Frage: Wieviel Möglichkeiten gibt es, um einen bestimmten Betrag (z.B. 1 Euro=100 Cent) in kleinere Münzen (zu 50, 20, 10, 5, 2, 1 Cent) zu wechseln?

Antwort: Formuliere die Antwort als Rekursion: die Möglichkeiten n Cent in kleinere Münzen zu wechseln unterteilen sich in diejenigen, die eine bestimmte Münzsorte (z.B. das 50-Cent-Stück) mindestens einmal verwenden, und die, die es gar nicht verwenden. Die einmalige Verwendung reduziert den Betrag, und die Nichtverwendung reduziert die Zahl der verfügbaren Münzen, so dass die Unterprobleme immer einfacher werden!

Programm:

```
(define (muenzbetrag k)
  (cond ((= k 1) 1)
        ((= k 2) 2)
        ((= k 3) 5)
        ((= k 4) 10)
        ((= k 5) 20)
        ((= k 6) 50)))

(define (moeglichkeiten betrag k)
  (cond ((= betrag 0) 1)
        ((< betrag 0) 0)
        ((= k 0) 0)
        (else (+ (moeglichkeiten betrag (- k 1))
                 (moeglichkeiten (- betrag (muenzbetrag k)) k)))))
```

Bemerkung:

- Dieses Programm lässt sich nicht einfach in einen linear iterativen Prozess überführen.
- Eine genauere Untersuchung zeigt, dass der Aufwand von der Ordnung $O(n^{k-1})$ ist.
- Später werden wir eine Technik kennenlernen, mit der auch dieser Prozess auf einfache Weise beschleunigt werden kann (Tabellierung, *memoization*).
- Es gibt eine einfachere Schreibweise der Funktion `muenzbetrag` mit Hilfe der Sonderform `case`:

```
(define (muenzbetrag k)
  (case k
    (1 1) ... (6 50)))
```

2.5.4 Größenordnungen

Voraussetzung: n sei ein Parameter, der die Größe eines Problems misst, z.B. der Betrag einer Zahl, die Länge einer ganzen Zahl (Logarithmus des Betrags), die Größe der Mantisse bei Fließkommazahlen, die Größe einer Menge, ... Es ist abhängig vom jeweiligen Problem, was sinnvolle Parameter sind. $R(n)$ sei die Menge an Speicher bzw. Rechenzeit, die ein Prozess benötigt.

Definition: $R(n)$ hat die Ordnung $\Theta(f(n))$, wenn es von n unabhängige Konstanten $c_1, c_2 > 0$ gibt, mit

$$c_1 f(n) \leq R(n) \leq c_2 f(n)$$

Beispiele:

Programm	Speicheraufwand	Rechenaufwand
Fakultät, rekursiv	$\Theta(n)$	$\Theta(n)$
Fakultät, iterativ	$\Theta(1)$	$\Theta(n)$
Fibonacci, rekursiv	$\Theta(n)$	$\Theta(\lambda^n)$
Fibonacci, iterativ	$\Theta(1)$	$\Theta(n)$
Geldwechseln	$\Theta(n+k)$	$\Theta(n^{k-1})$

2.5.5 Schnelle Exponentiation

Die einfachste Exponentiationsfunktion basiert auf der Beziehung $b^n = b \cdot b^{n-1}$ und ließe sich so schreiben:

```
(define (expt b n)
  (if (= n 0)
      1
      (* b (expt b (- n 1)))))
```

Viel schneller aber ist folgende Rekursion

$$b^n = \begin{cases} 1 & n = 0 \\ (b^{\frac{n}{2}})^2 & n \text{ gerade} \\ b \cdot b^{n-1} & n \text{ ungerade} \end{cases}$$

Programm: (Schnelle Exponentiation)

```
(define (expt b n)
  (cond ((zero? n)
         1)
        ((even? n)
         (square (expt b (/ n 2))))
        (else
         (* b (expt b (- n 1))))))
```

Bemerkung:

- Die Konvention in Scheme ist, dass Prädikate wie `zero?`, `even?`, `odd?` immer mit dem Zeichen „?“ enden.
- Der Aufwand an Opertionen ist nur noch logarithmisch in n, wobei die Operationen selbst aber mit langen Zahlen operieren. (Der simple Algorithmus musste das aber auch.)

2.5.6 Euklids ggT-Algorithmus

Einer der wichtigsten Algorithmen der Computeralgebra ist der Euklidische Algorithmus zur Berechnung des größten gemeinsamen Teilers (ggT, engl. gcd) zweier Zahlen $a, b \in \mathbb{N}$. Er stammt etwa aus dem Jahre 300 vor Christus.

Programm: (Euklidischer Algorithmus)

```
(define (gcd a b)
  (if (= b 0)
      a
      (gcd b (remainder a b))))
```

Bemerkung: Der Euklidische Algorithmus braucht $O(\log \min(a, b))$ Schritte, um den $\mathrm{ggT}(a, b)$ zu berechnen. (Satz von Lamé: Am längsten brauchen die Fibonacci-Zahlen.)

2.5.7 Primzahlen

Frage: Ist eine Zahl $n \in \mathbb{N}$ eine Primzahl?

Idee: Teste alle $k \in \mathbb{N}$ mit $2 \leq k \leq \sqrt{n}$, ob sie Teiler sind.

Programm: (Einfacher Primalitätstest)

```
(define (find-divisor k n)
   (if (> (* k k) n)
       n
       (if (= (remainder n k) 0)
           k
           (find-divisor (+ k 1) n))))
(define (prime? n)
   (= (find-divisor 2 n) n))
```

Problem: Der Aufwand des Algorithmus ist für eine Primzahl p von der Ordnung $\Theta(\sqrt{p})$.

2.5.8 Der Fermat-Test

Satz: (Kleiner Satz von Fermat) Sei $n > 2$ Primzahl und $1 \leq a < n$. Dann gilt

$$a^{n-1} \equiv 1 \bmod n \quad [\Rightarrow a^n \equiv a \bmod n]$$

Beweis: $\mathbb{Z}_n^\times = \{a \in 1, \ldots, n-1 : \mathrm{ggT}(n, a) = 1\}$ ist eine Gruppe bezüglich der Multiplikation modulo n. Wenn n prim ist, so ist $|\mathbb{Z}_n^\times| = n - 1$, und es gilt in allen endlichen Gruppen $a^{|G|} = 1$.

Beobachtung: Die Beziehung $a^{n-1} \equiv 1 \bmod n$ ist ein erstaunlich guter Test auf Primalität! So versagt bereits der Test mit $a = 2$ unterhalb 10000 nur für die Zahlen 341, 561, 645,

1105, 1387, 1729, 1905, 2047, 2465, 2701, 2821, 3277, 4033, 4369, 4371, 4681, 5461, 6601, 7957, 8321, 8481, 8911. Nach einem weiteren Test mit $a = 3$ verbleiben nur noch 561, 1105, 1729, 2465, 2701, 2821, 6601, 8911. Dies sind alles sogenannte Carmichael-Zahlen.

Programm: (Fermat-Test)

```
(define (fermat-test? n k)
  (= k (expt-mod k n n)))
```

Satz: Im Normalfall (d.h., außer für die sehr seltenen Carmichael-Zahlen, bei denen man die Nichtprimalität auf andere Weise einfach feststellen kann) ist die Wahrscheinlichkeit, dass eine Nicht-Primzahl n den Fermat-Test für ein $a < n$ besteht, weniger als $\frac{1}{2}$.

Beweis: Die Zahlen a mit $a^{n-1} \equiv 1 \bmod n$ bilden eine Untergruppe U von $\mathbb{Z}_n^\times$. Im Falle der Carmichael-Zahlen ist $U = \mathbb{Z}_n^\times$, ansonsten aber eine echte Untergruppe, deren Elementzahl $|\mathbb{Z}_n^\times|$ teilen muss.

Folgerung: Wiederholtes Testen kann die Primalität einer großen Zahl mit sehr hoher Wahrscheinlichkeit feststellen.

Programm: (Wiederholter Fermat-Test) Der folgende Code führt einen solchen probabilistischen Primzahltest (ohne Berücksichtigung der Carmichael-Zahlen) aus:

```
(define (fermat-prime? n count)
  (or (zero? count)
      (and (fermat-test? n (random-integer 2 n))
           (fermat-prime? n (- count 1)))))
```

Bausteine:

- Schnelle Exponentiation modulo N
- Guter Zufallszahlengenerator

2.5.9 Schnelle Exponentiation modulo N

Programm: (Schnelle Exponentiation modulo n)
Das folgende Programm berechnet ein $x \in \{0, \ldots, n-1\}$ mit $b^k \equiv x \bmod n$:

```
(define (expt-mod b k n)
  (remainder
   (cond ((zero? k) 1)
         ((even? k) (square (expt-mod b (/ k 2) n)))
         (else (* b (expt-mod b (- k 1) n))))
   n))
```

2.5.10 Zufallszahlengenerator

Problem: Die Sprache Scheme verlangt keine Pseudo-Zufallszahlen. DrScheme/MzScheme besitzt zwar einen Zufallszahlengenerator (Funktion `random`), der erlaubt aber nur Argumente kleiner 2^{31}.

Abhilfe: Setze eine große Zufallszahl aus kleinen zusammen.

Programm: (Große Zufallszahlen)

```
(define (random-integer from to)
  (define block-size (expt 2 30))
  (define delta (- to from))
  (do ((r 0 (+ (* r block-size)
               (random block-size))))
      ((> r delta)
       (+ from (remainder r delta)))))
```

Bemerkung: `do` ist ein Schleifenkonstrukt in Scheme, das durch eine Syntaxtransformation in einen endrekursiven Aufruf übersetzt wird. Die Syntax ist:

```
(do ((var-1 init-1 update-1)
     ...
     (var-n init-n update-n))
   (end-condition result)
 ...)
```

Bemerkung: Die zusammengesetzten Zufallszahlen sind nicht vollkommen gleichverteilt, wenn der Bereich keine Zweierpotenz ist. Außerdem könnten, je nach Komplexität der aufgerufenen `random`-Funktion, einige Zahlen nicht getroffen werden. Wir ignorieren diese Schwierigkeiten hier. Für ernsthafte Anwendungen ist es sowieso empfehlenswert, sich die Erzeugung der Zufallszahlen genauer anzuschauen—selbst wenn ein eingebauter Generator für große Zahlen zur Verfügung steht.

2.5.11 Kryptographie

Bemerkung: Der Fermat-Test liefert *keine Zerlegung* der Primzahl. Bisher ist auch kein Algorithmus bekannt, der eine solche Zerlegung schnell berechnet. Dies ist die Basis von Verschlüsselungsverfahren, die ohne Austausch geheimer Codes auskommen!

Skizze: Das RSA-Verfahren arbeitet dabei im Prinzip folgendermaßen:

- Person A generiert zwei große Primzahlen p und q (z.B. jede ist 1024 Bits lang).
- Das Produkt $n = pq$ wird öffentlich gemacht.
- Mit Hilfe von n kann Person B eine Nachricht an A verschlüsseln.
- Das Entschlüsseln geht nur, wenn man die Zerlegung von n hat.

Bemerkung: Der komplexeste Bestandteil des RSA-Verfahrens ist das Finden von großen zufälligen Primzahlen.

Programm: (Zufällige Primzahlen)
Finde eine zufällige Pseudo-Primzahl im Bereich $[\frac{n}{2}], \ldots, n$. Der `safety`-Parameter gibt an, wie oft der Fermat-Test bestanden werden muss.

```
(define (find-prime n safety)
  (define k (random-integer (quotient n 2) n))
  (if (fermat-prime? k safety)
      k
      (find-prime n safety)))
```

2.5.12 Übungen

Aufgabe 11: (4 Punkte)
Sei `M` die in der Vorlesung vorgestellte Prozedur, welche die Anzahl der Möglichkeiten berechnet, den Betrag `n` in `k` Münzarten zu wechseln.

a) Überlegen Sie sich, welchen Rechenaufwand `M` in Abhängigkeit seiner Parameter haben sollte.

b) Überprüfen Sie ihre Vorhersage zur Abhängigkeit von `n`, indem Sie die Prozedur `M` selber implementieren und dann folgende Ausdrücke auswerten:
```
(time (M 200 2))
(time (M 300 2))
(time (M 400 2))
(time (M 500 2))
```
(Der Befehl `time` misst die Ausführungszeit seines Operanden.)

Aufgabe 12: (4 Punkte)
Angenommen unsere Prozedur zur schnellen Potenzberechnung sähe so aus:

```
(define (fast-expt b n)
  (cond ((= n 0) 1)
        ((even? n) (* (fast-expt b (quotient n 2))
                      (fast-expt b (quotient n 2))))
        (else (* b (fast-expt b (- n 1))))))
```

Wäre ihr Name noch berechtigt? Wie gross ist der Rechenaufwand in Abhängigkeit von n?

Aufgabe 13: (2 Punkte)
$C = \{561, 1105, 1729, 2465, 2821, 6601, 8911\}$ ist die Menge aller Carmichael-Zahlen, die kleiner als 10000 sind. Schreiben Sie eine Prozedur, die überprüft, dass der Fermat-Test für $n \in C$ und $1 < a < n$ versagt.

2.6 Prozeduren höherer Ordnung

Bisher: Funktionen waren Abstraktionen für Operationen auf Zahlen.

Jetzt: Funktionen können als Argumente oder Werte wieder Funktionen haben.

Anwendung: Überall in der Mathematik: $\sum_{k=0}^{n} f(k)$, $\int_0^1 g(x)\,dx$, $\frac{d}{dx} : C^1 \to C^0$, ...

2.6.1 Funktionen als Argumente

Beispiel: Wir wollen die Summation $\sum_{n=a}^{b} f(n)$ für $a, b \in \mathbb{N}$ und $f : \mathbb{N} \to \mathbb{N}$ abbilden.

Programm: (Summe)

```
(define (sum a b f initial)
 (if (> a b)
     initial
     (sum (+ a 1) b f (+ initial (f a)))))
```

Anwendungen:

- Approximation von $\frac{\pi^2}{6} = \sum_{k=1}^{\infty} \frac{1}{k^2}$:

```
(define (f x)
   (/ (* x x)))
(sqrt (* 6
         (sum 1 1000 f 0.0)))
```

- Berechnung von $\int_a^b f(x)\,dx$:

```
(define (integral a b f)
  (define n 1000)
  (define h (/ (- b a) n))
  (define (cell-contribution k)
    (* h
       (f (+ a
             (* (- k 0.5) h)))))
  (sum 1 n cell-contribution 0.0))
```

2.6.2 Anonyme Funktionen

Problem: Bei der Approximation von $\frac{\pi^2}{6}$ wurde die Funktion $f : x \mapsto \frac{1}{x^2}$ nur einmal gebraucht. Trotzdem musste sie benannt werden. Gleiches gilt für die interne Funktion `cell-contribution`.

Abhilfe: Die Erzeugung anonymer Funktionen ist möglich durch die Sonderform `lambda`. Zum Beispiel entspricht

```
(lambda (x) (/ 1 (square x)))
```

direkt der mathematischen Formulierung „$x \mapsto \frac{1}{x^2}$“!

Bemerkung:

- Das Schlüsselwort `lambda` rührt vom sogenannten λ-Kalkül von Alonzo Church her. Seine Notation war so etwas wie $\lambda x.x^2$ für $x \mapsto x^2$.
- Man beachte, dass die frisch erzeugte Funktion auf *alle* Variablen in der Umgebung zurückgreifen kann, in der sie definiert wurde. Zum Beispiel:

```
(define (integral a b f)
  (define n 1000)
  (define h (/ (- b a) n))
  (sum 1 n (lambda (k)
             (* h
                (f (+ a
                      (* (- k 0.5) h)))))
       0.0))
```

- Diese Funktionalität ist in vielen maschinennahen Sprachen nicht verfügbar, da sie erstens nicht einfach zu implementieren ist, und zweitens die Eleganz beim Zwang zur Typdeklaration teilweise verloren geht. C++ soll sie aber im C++0x-Standard auch erhalten. Außerdem gibt es schon benutzbare Ansätze in der Boost-Bibliothek.

2.6.3 Lokale Variablen

Beobachtung: Mit `lambda` kann man lokal neue Bezeichner definieren, ähnlich wie mit dem internen `define`:

```
((lambda (flaeche umfang)
   ;; verwende 'flaeche' und 'umfang'
   )
 (* pi r r)
 (* 2 pi r))
```

Nachteil: Die Werte der Variablen kommen erst am Ende, was meist nicht wünschenswert ist. (Dennoch wurde dieses Konstrukt in den Frühzeiten von Lisp tatsächlich so verwendet.)

Abhilfe: Durch eine Syntaxtransformation kann man

```
((lambda (a b ...)
   code ...)
 operand-1 operand-2 ...)
```

ersetzen durch

```
(let ((a operand-1)
      (b operand-2)
      ...)
     code ...)
```

Beobachtung: Die Variablen werden „parallel“ zugewiesen, d.h.

```
(let ((a 1)
      (b 2)
      (square-area (* a b)))
  square-area)
```

geht nicht (bzw. verwendet global definierte Werte von `a` und `b`).

Abhilfe: Es gibt eine sequentielle lokale Variablendeklaration mit `let*`. Damit funktioniert dann

```
(let* ((a 1)
       (b 2)
       (square-area (* a b)))
  square-area)
```

wie (hoffentlich) gewünscht.

Bemerkungen:

- `let*` ist natürlich wieder syntaktischer Zucker für eine Verschachtelung von `let`- oder `lambda`-Konstrukten.
- In gutem Code scheint `let*` relativ selten nötig zu sein. Manchmal verwendet man es zum Zerlegen und zur Dokumentation eines komplizierteren Ausdrucks.

2.6.4 Funktionen als Werte

Beispiel: (Potenzengenerator)

```
(define (x^ n)
  (lambda (x) (expt x n)))

(define square (x^ 2))
(square 5)

(define cube (x^ 3))
(cube 5)
```

Beispiel: (Numerische Ableitung)

```
(define (numerical-gradient f h)
  (lambda (x)
    (/ (- (f (+ x h))
          (f (- x h)))
       (* 2 h))))

((numerical-gradient square 1e-4) 3)
```

2.6.5 Newton-Verfahren

Wissen: Das Newton-Verfahren zum Lösen allgemeiner nichtlinearer Gleichungen $f(x) = 0$ ist einer der wichtigsten Algorithmen der Numerischen Analysis.

Algorithmus: (Newton-Verfahren) $f \in C^1(\mathbb{R})$, $f' \in C^0(\mathbb{R})$ und $x^{(0)} \in \mathbb{R}$ seien gegeben. Man wiederhole

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})}$$

bis $|f(x^{(k)})| < \mathrm{TOL}$.

Programm:

```
(define (newton f Df x_k tolerance)
  (define (improve-guess x)
    (- x (/ (f x) (Df x))))
  (define (good-enough? x)
    (< (abs (f x)) tolerance))
  (if (good-enough? x_k)
      x_k
      (newton f Df (improve-guess x_k) tolerance)))

(newton (lambda (x) (- (* x x) 2.0))
        (lambda (x) (* 2 x))
        1.0
        1.0e-10)
```

Bemerkung: Mit Hilfe der numerischen Ableitung können wir eine einfachere Schnittstelle erzeugen:

```
(define (find-zero f)
  (newton f (numerical-gradient f 1.0e-4)
          1.0 1.0e-10))

(find-zero (lambda (x) (- (* x x) 2.0)))
```

2.7 Syntaxtransformationen

Definition: Syntaxtransformationen kann man in Scheme mit `define-syntax` schreiben:

```
(define-syntax <name>
   (syntax-rules ()
     (pattern-1 translation-1)
     ...
     (pattern-n translation-n)))
```

Beispiel: Das `let`-Konstrukt könnte damit folgendermaßen definiert werden:

```
(define-syntax let
  (syntax-rules ()
    ((let ((var exp) ...)
          body ...)
     ((lambda (var ...)
        body ...)
      exp ...))))
```

Beispiel: Die Definition von `let*` braucht mehrere Muster und verwendet `let`:

```
(define-syntax let*
  (syntax-rules ()
    ((let* () body ...)
     (let () body ...))
    ((let* (definition next ...) body ...)
     (let (definition)
       (let* (next ...) body ...)))))
```

Bemerkung:

- Diese Art der Definition neuer Syntax durch Musterabgleich (pattern matching) ist ein mächtiges Werkzeug, das in der Lisp-Familie spezifisch für Scheme ist. In Common Lisp verwendet man üblicherweise einen funktional im wesentlichen äquivalenten aber bodenständigeren Zugang (`defmacro`).
- Man beachte, dass die Muster alle relativ einfach sind, wobei die einfache Basissyntax von Lisp/Scheme entscheidend ist. Man stelle sich zum Beispiel die Schwierigkeit vor, eine Funktions- oder Klasssendefinition in C++ (Typdeklarationen, public/private etc.) syntaktisch zu erfassen.

Bemerkung: Wenn immer möglich, werden wir in Zukunft interne `define`-Ausdrücke durch `let` oder `let*` ersetzen, weil diese Schreibweise auch in anderen Lisp-Dialekten funktioniert.

2.7.1 Interne Funktionsdefinitionen

Beobachtung: Wir können mit `let/let*` zwar interne Funktionen definieren, z.B.

```
(define (1+x^2 x)
  (let ((square (lambda (x) (* x x))))
    (+ 1 (square x))))
(1+x^2 5)
```

Allerdings können diese nicht rekursiv sein:

```
(define (factorial n)
  (let ((fact-iter (lambda (n f)
                     (if (= n 0)
                         f
                         (fact-iter (- n 1) (* n f))))))
    (fact-iter n 1)))

(factorial 10)  ; Fehler!
```

Abhilfe: Mit der Sonderform `letrec` kann man die iterative Fakultät wie folgt definieren:

```
(define (factorial n)
  (letrec ((fact-iter
            (lambda (k f)
              (if (= k 0)
                  f
                  (fact-iter (- k 1) (* k f))))))
    (fact-iter n 1)))
```

Bemerkungen:

- Auch das `letrec`-Konstrukt kann man mit Hilfe von `lambda` und Syntaxtransformationen definieren, wir heben uns das aber für später auf.
- Wenn man interne Funktionen mit Namen braucht (egal, ob rekursiv oder nicht), ist die Verwendung des internen `define` wohl die syntaktisch hübscheste Art.

Bemerkung: Bei gegebenem `letrec` läßt sich allerdings ein sehr flexibles Schleifenkonstrukt namens `named-let` einfach definieren:

```
(define-syntax named-let
  (syntax-rules ()
    ((named-let name ((var exp) ...)
                body ...)
     (letrec ((name (lambda (var ...) body ...)))
       (name exp ...)))))
```

```
(define (factorial n)
  (named-let iter ((k n) (f 1))
    (if (= k 0)
        f
        (iter (- k 1) (* k f)))))
```

Bemerkung: Da man auch anhand des ersten Operanden entscheiden kann, ob es ein einfaches `let` oder ein `named-let` ist, verzichtet Scheme auf das separate Schlüsselwort und verwendet in beiden Fällen einfach `let`.

2.8 Übungen

Aufgabe 14: (4 Punkte)
Bei der numerischen Berechnung von $\int_a^b f(x)\,dx$ für eine glatte Funktion f liefert die im folgenden beschriebene Simpson-Regel normalerweise genauere Ergebnisse als die Mittelpunktsregel. Sei $h = \frac{b-a}{n}$ für eine gerade natürliche Zahl n, und sei $f_k := f(a + kh)$. Dann ist

$$S_n = \frac{h}{3}(f_0 + 4f_1 + 2f_2 + 4f_3 + 2f_4 + \ldots + 2f_{n-2} + 4f_{n-1} + f_n)$$

die durch die Simpson-Regel gegebene Approximation. Implementieren Sie die Simpson-Regel als eine Funktion mit den Argumenten f, a, b, n, und berechnen Sie damit $\int_0^1 x^3\,dx$ und $\int_0^1 e^x\,dx$ (beispielsweise mit $n = 5, 10, 20$). In Scheme heisst die Exponentialfunktion übrigens `exp`.

Aufgabe 15: (2 Punkte)
Die in der Vorlesung angegebene Funktion `sum` erzeugt eine lineare Rekursion. Schreiben Sie die Funktion so um, dass die Summe iterativ berechnet wird (Tip: Hilfsfunktion mit lokalem `define`).

Aufgabe 16: (3 Punkte)
Schreiben Sie analog zu `sum` eine Funktion `product`, die Produkte der Art $f(a) \cdots f(b)$ berechnet, und definieren Sie damit die Fakultätsfunktion, sowie eine Näherungsfunktion für π nach der Formel

$$\frac{\pi}{4} = \frac{2 \cdot 4 \cdot 4 \cdot 6 \cdot 6 \cdot 8 \cdots}{3 \cdot 3 \cdot 5 \cdot 5 \cdot 7 \cdot 7 \cdots}$$

Aufgabe 17: (4 Punkte)
Die Funktionen `sum` und `product` lassen sich weiter abstrahieren: sie sind Spezialfälle einer Funktion `accumulate` der Form

```
(accumulate combiner initial-value f a inc b)
```

abstrahieren lassen. Hierbei ist `combiner` eine Funktion zweier Argumente und entspricht dem Operator $+$ bei `sum`, `inc` ist eine Funktion, welche ihr Argument erhöht. Schreiben Sie `accumulate`, und definieren Sie `sum` und `product` damit.

Aufgabe 18: (4 Punkte)
Noch flexibler wird `accumulate`, wenn man ein Prädikat `filter?` übergibt, welches anzeigt, ob der aktuelle Wert a verwendet werden soll oder nicht. Erweitern Sie Ihre Funktion `accumulate` entsprechend, und berechnen Sie damit die Summe über alle Primzahlen zwischen 0 und 1000.

Aufgabe 19: (1 Punkt)
Erklären Sie die Fehlermeldung, die Sie durch die folgende Sequenz erhalten:

```
(define (f g) (g 2))
(f f)
```

Aufgabe 20: (1 Punkt)
Definieren Sie eine Funktion `(cubic a b c)`, welche das kubische Polynom $p(x) = x^3 + ax^2 + bx + c$ als Funktion $x \mapsto p(x)$ zum Wert hat.

Aufgabe 21: (4 Punkte)
Schreiben Sie eine Funktion `compose`, die zwei einstellige Funktionen f, g als Argumente nimmt und als Wert die Komposition $f \circ g$ liefert. Begründen Sie dann den Wert der folgenden Sequenz:

```
(define (1+ x) (+ x 1))
(define (double f)
   (compose f f))
(((double (double double)) 1+) 5)
```

Schreiben Sie ferner eine Funktion `repeated` mit den Argumenten `f` und `n` welche als Ergebnis die `n`-fache Hintereinanderausführung von `f` liefert.

3 Abstraktionen mit Daten

Bisher: Es wurde mit Zahlen und Funktionen gearbeitet.

Jetzt: Konstruktion zusammengesetzter Datenobjekte.

3.1 Einführung

3.1.1 Beispiel: Rationale Zahlen

Ansatz: Rationale Zahlen werden als Paare ganzer Zahlen dargestellt.

Konstruktion eines Datenobjekts

a) Konstruktor: `(make-rational numerator denominator)` konstruiert eine rationale Zahl aus Zähler und Nenner.

b) Selektoren: `(numerator rat)` liefert den Zähler, `(denominator rat)` den Nenner einer rationalen Zahl.

Definition der arithmetischen Operationen Z.B. überträgt sich die Formel der Bruchaddition

$$\frac{n_1}{d_1} + \frac{n_2}{d_2} = \frac{n_1 d_2 + n_2 d_1}{d_1 d_2}$$

in

```
(define (add x y)
  (make-rational
   (+ (* (numerator x) (denominator y))
      (* (denominator y) (numerator x)))
   (* (denominator x) (denominator y))))
```

Ähnlich für andere Funktionen: `sub`, `mul`, `div`, `equal?`, `smaller?`, ...

Implementation von Konstruktor und Selektoren Es gibt das Paar (*pair*) als eingebauten Datentyp. Der Konstruktor heißt `cons`, die Selektoren heißen `car` und `cdr`. Damit können wir rationale Zahlen implementieren als:

```
(define make-rational cons)
(define numerator car)
(define denominator cdr)
```

Bemerkung: `cons` kommt von *construct*, die Namen `car` und `cdr` rühren von der ersten Implementation auf einer IBM 704 her (*contents-of-address-part-of-register*, *contents-of-decrement-part-of-register*). `cdr` wird wie „kudder“ ausgesprochen.

Verwendung:

```
(add (make-rational 1 2)
     (make-rational 1 3))
=> (5 . 6)
```

Bemerkung: Die Anzeige kann man wie folgt verbessern:

```
(define (display-rat rat)
  (display (numerator rat))
  (display "/")
  (display (denominator rat))
  (newline))
```

Problem:

- Kein Kürzen,
- Nenner kann negativ sein.

Verbesserung:

```
(define (make-rational numerator denominator)
  (if (negative? denominator)
      (make-rational (- numerator) (- denominator))
      (let ((g (gcd numerator denominator)))
        (cons (/ numerator g) (/ denominator g)))))
```

Man beachte, dass nur der Konstruktor geändert werden musste!

Beobachtung: Das Paket zum Arbeiten mit rationalen Zahlen ist in drei sauber voneinander getrennten Ebenen implementiert:

a) Arithmetische Operationen (`add`, `mul`, ...)

b) Konstruktion und Zugriff auf rationale Zahlen (`make-rational`, `numerator`, `denominator`)

c) Scheme-Implementation der Paare (`cons`, `car`, `cdr`)

Alle Ebenen sind durch Schnittstellen (*interfaces*) voneinander getrennt. Diese Schnittstellen sind durch die jeweiligen Funktionen definiert.

3.1.2 Was sind Daten

Definition: Ein zusammengesetzter abstrakter Datentyp (ADT) ist definiert durch einen Konstruktor, Selektoren, sowie Kompatibilitätsbedingungen zwischen diesen.

Beispiel: Die Kompatibilitätsbedingung für den ADT *rationale Zahl* ist

$$\frac{\texttt{(numerator x)}}{\texttt{(denominator x)}} = \frac{n}{d} \text{ wenn } x = \texttt{(make-rational n d)}\,.$$

Man beachte, dass die Metasprache der Mathematik verwendet wurde.

Beispiel: Für den ADT *Paar* gibt es zwei Kompatibilitätsbedingungen:

a) `(car (cons x y)) = x`

b) `(cdr (cons x y)) = y`

Bemerkung: Jede Implementation von `cons`/`car`/`cdr`, die dies erfüllt, ist zulässig. Zum Beispiel:

```
(define (cons x y)
  (lambda (message)
    (if message x y)))
(define (car z) (z #t))
(define (cdr z) (z #f))
```

Oder ausschließlich mit `lambda`:

```
(define (cons x y)
  (lambda (f)
    (f x y)))

(define (car z)
  (z (lambda (x y) x)))
(define (cdr z)
  (z (lambda (x y) y)))
```

3.2 Hierarchische Datenstrukturen

Beobachtung: Die Bestandteile eines Paars können beliebige Datenstrukturen sein, insbesondere wieder Paare. Diese Eigenschaft nennt man Abgeschlossenheit. (In Sprachen, in denen Datentypen deklariert werden müssen, ist das nicht selbstverständlich!)

Beispiel:

```
(cons (cons 1 2) (cons 3 4))
```

3.2.1 Listen

Ziel: Darstellung einer Liste mit Hilfe von Paaren.

Definition: Eine Liste ist entweder die leere Liste `()` oder aber von der Gestalt `(cons x liste)`, wobei `liste` wieder eine Liste ist.

Beispiel:

```
(cons 1 (cons 2 (cons 3 (cons 4 ()))))
```

Hierfür gibt es eine Abkürzung: `(list 1 2 3 4)`. Wie + ist auch `list` eine Funktion, die beliebig viele Argumente akzeptiert.

3.2.2 Einfache Listenoperationen

- `(null? liste)` - vergleicht `liste` mit `()`.
- `(list-ref liste n)` - liefert das n-te Element von `liste`, wobei man von 0 an zählt.
- `(reverse liste)` - dreht `liste` um. Diese Funktion könnte so geschrieben werden:

```
(define (my-reverse liste)
  (let loop ((liste liste)
             (result ()))
    (if (null? liste)
        result
        (loop (cdr liste)
              (cons (car liste) result)))))
```

- `(append liste1 liste2)` - hängt (eine Kopie von) `liste1` vor `liste2`.

3.2.3 Listenoperationen höherer Ordnung

Problem: Oft möchte man Operationen auf alle Listenelemente anwenden, wie z.B. das Multiplizieren von allen Elementen mit einem Faktor, was man wie folgt schreiben könnte:

```
(define (scale-list factor liste)
  (if (null? liste)
      ()
      (cons (* factor (car liste))
            (scale-list factor (cdr liste)))))
```

Ebenso könnte man alle Elemente quadrieren, etc. Es ist aber unpraktisch, für jede solche Operation eine neue Funktion schreiben zu müssen: `scale-list`, `square-list`, ...

Abhilfe: Wir führen eine höhere Abstraktionsstufe ein: die Abbildung von Listen. Diese ist durch die Funktion `map` gegeben.

Programm:

```
(define (map f liste)
  (if (null? liste)
      ()
      (cons (f (car liste))
            (map f (cdr liste)))))
```

Anwendung:

- Skalierung: `(map (lambda (x) (* 2 x)) liste)`
- Quadrierung: `(map square liste)`
- ...

Bemerkung:

- Die Abstraktion durch `map` versteckt die rekursive Abarbeitung und könnte bei Bedarf leicht durch eine iterative Version ersetzt werden. Diese könnte man folgendermaßen schreiben:

```
(define (map f liste)
  (let loop ((rest liste)
             (result ()))
    (if (null? rest)
        (reverse result)
        (loop (cdr rest)
              (cons (f (car rest)) result)))))
```

- Man beachte, dass die ursprüngliche Liste weder von der rekursiven noch von der iterativen Version verändert wird!

3.2.4 Funktionen mit variabler Parameterzahl

Frage: Wie schreibt man Funktionen wie +, −, `list`, die eine beliebige Zahl von Parametern akzeptieren.

Antwort: Wenn in einer Parameterliste der letzte Parameter durch einen Punkt separiert ist, so werden alle restlichen Argumente diesem Parameter als Liste übergeben.

Beispiel: Die `list`-Funktion könnte man wie folgt schreiben:

```
(define (list . args) args)
```

Syntax: Ein wichtiges Gegenstück zu diesem Sprachmerkmal ist die Funktion `apply`. Diese hat die Syntax

```
(apply function arg1 ... argn restargs)
```

Der letzte Parameter `restargs` ist eine Liste, deren Inhalt an die Funktion als Argumente übergeben wird.

Beispiel: Die eingebaute Funktion `map` akzeptiert auch beliebig viele Listen. Diese Version könnte man folgendermaßen schreiben:

```
(define (map f liste . lists)
  (if (null? liste)
      ()
      (cons (apply f (car liste) (map car lists))
            (apply map f
                   (cdr liste) (map cdr lists)))))
```

3.2.5 Baumstrukturen

Beobachtung: Weil Listenelemente wieder Listen sein können (Abgeschlossenheit), kann man mit Ihnen beliebige Baumstrukturen aufbauen. So könnte man etwa die Liste `((1 2) 3 4)` als einen Baum mit 3 Ästen interpretieren, dessen erster Ast einen Unterbaum mit 2 Blättern beinhaltet.

Wichtig: Für den Umgang mit Bäumen ist Rekursion das entscheidende Werkzeug.

Beispiel: Zähle die Blätter eines Baums:

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else (+ (count-leaves (car x))
                 (count-leaves (cdr x))))))
```

Das Prädikat `pair?` prüft dabei, ob ein Objekt ein Paar ist.

3.2.6 Listen als Schnittstellen

Beobachtung: `accumulate-tree` ist zwar eine hübsche Funktion, wenn man oft mit Bäumen operiert. Wenn das aber nicht der Fall sein sollte, gäbe es einen schnelleren Weg, um etwa `count-leaves` zu definieren:

```
(define (count-leaves tree)
  (length (flatten tree)))
```

Idee: Es wird eine Liste als Zwischenergebnis erzeugt, die an eine weitere Funktion weitergereicht wird.

Beispiel: Berechne die Summe der Quadrate aller ungeraden Zahlen in einem Baum. Dies könnte man schreiben als

```
(define (sum-odd-squares-of-tree tree)
  (tree-accumulate
   tree + 0 (lambda (x)
              (if (odd? x)
                  (square x)
                  0))))
```

Alternativ wäre folgende Form möglich:

```
(define (sum-odd-squares-of-tree tree)
  (apply +
         (map square
              (filter odd?
                      (flatten tree)))))
```

Bemerkung: Die Funktion `filter` arbeitet dabei auf Listen und ist wie folgt definiert:

```
(define (filter pred? liste)
  (cond ((null? liste) ())
        ((pred? (car liste))
         (cons (car liste) (filter pred? (cdr liste))))
        (else (filter pred? (cdr liste)))))
```

Bemerkung: Dies erinnert an die Verkettung von Filtern und transformierenden Bausteinen in der Signalverarbeitung.

Vorteil:

- Viele Listenoperationen können wiederverwendet werden (LISP=list processing).
- Die Einzelfunktionen bleiben einfach.

Zitat: Alan Perlis: "It is better to have 100 functions operating on one datastructure, than 10 functions operating on 10 datastructures.".

3.3 Der Lambda-Kalkül

3.3.1 Geschichte

- Die funktionale Denkweise ist ein wichtiger Aspekt der modernen Mathematik.
- Der Lambda-Kalkül wurde von Alonzo Church 1934 eingeführt. (*lambda calculus*). Er macht Funktionen zur Grundlage der Mathematik überhaupt.
- Er ist äquivalent zur Turing-Maschine [Turing, 1936].
- Der Lambda-Kalkül ist Grundlage der Computersprache Lisp (ca. 1958).
- Heutzutage beschäftigt sich die Forschung oft mit getypten Varianten des LK. Hier haben Funktionen eine Signatur, die angibt, welche Datentypen die Funktion akzeptiert. Insbesondere das automatische Ableiten von Signaturen und Datentypen von Variablen ist dabei wichtig.
- Sprachen, die auf dem getypten LK basieren, sind z.B. Haskell, ML, ...

3.3.2 Beschreibung

Der LK kennt nur zwei Sprachelemente:

- Funktionserzeugung mit λ, z.B. ist $\lambda x.x$ die Identität. In Scheme ist die Notation geklammert, also `(lambda (x) x)`.
- Funktionsanwendung, z.B. bedeutet fx die Anwendung von f auf x. In Scheme ist die Notation `(f x)`.

Bemerkung: Common Lisp hat verschiedene Namensbereiche für Funktionen und Variablen. Darum muss `(funcall f x)` geschrieben werden, wenn `f` ein Variablenname ist.

Beobachtung: Der Lambda-Kalkül ist offenbar eingebettet in die Sprache Scheme.

Ziel: Wir wollen den Lambda-Kalküls so weit in Richtung einer vollen Scheme-Implementation weiterentwickeln, dass einige Programme aus Kapitel 1 (Fakultät, Fibonacci, ...) damit geschrieben werden können.

3.3.3 Syntaxtransformationen

Problem: Programme im Lambda-Kalkül sind offenbar geklammerte Konstrukte der Form `((((lambda (x y) (lambda (z) ...))))))`, so wie Daten im Computerspeicher eine Folge von Nullen und Einsen sind. Dies ist sehr schlecht lesbar.

Abhilfe: Wir erlauben Syntaxtransformationen, mit denen wir Ausdrücke in andere Ausdrücke überführen. Die Expansion dieser Ausdrücke liefert dann eine Form, die nur `lambda` und Funktionsauswertungen enthält.

Definition: Syntaxtransformationen kann man in Scheme mit `define-syntax` schreiben:

```
(define-syntax <name>
   (syntax-rules ()
     (<pattern-1> <translation-1>)
     ...
     (<pattern-n> <translation-n>)))
```

Ein Muster ist dabei irgendein Ausdruck, der mit dem Schlüsselwort `<name>` eingeleitet wird. In diesem Ausdruck dürfen beliebige Bezeichner vorkommen, die dann in der Übersetzung wiederverwendet werden. Eine Besonderheit ist ein Muster der Form „`<expr> ...`“. Dies bedeutet, dass mehrere Ausdrücke der Form `<expr>` an dieser Stelle auftreten können.

Bemerkung: Lokal gültige Syntaxtransformationen kann man mit `letrec-syntax` einführen. Dies wollen wir im folgenden verwenden.

3.3.4 Neue Bezeichner

Wissen: Ein wichtiges Sprachkonstrukt war die Einführung neuer Bezeichner mit `let` und `let*`. Wir haben bereits gesehen, wie man dies mit Hilfe von `lambda` einfach erreichen kann.

Programm: (let/let*)

```
(letrec-syntax
 ((let (syntax-rules ()
         ((let ((var exp) ...)
                 body ...)
         ((lambda (var ...)
             body ...)
          exp ...))))
  (let* (syntax-rules ()
          ((let* () e1 ...)
           ((lambda () e1 ...)))
              ((let* ((a1 b1) c1 ...) e1 ...)
               ((lambda (a1)
                  (let* (c1 ...) e1 ...))
                b1)))))
   ;; ab jetzt darf let und let* verwendet werden!
  )
```

3.3.5 Wahr und falsch

Programm:

```
(let* ((identity (lambda (x) x))
       (constantly (lambda (x) (lambda (y) x)))
       (true constantly)
```

```
    (false (constantly identity))
    (_if_ (lambda (b x y) ((b x) y))))

  ;; nun haben wir auch true/false/_if_
  )
```

Bemerkung: `_if_` ist im Augenblick noch als Funktion definiert, daher nicht in allen Fällen verwendbar.

3.3.6 Die Kommunikation mit Scheme

Beobachtung: Wir können Ausdrücke in unserer erweiterten Sprache schreiben und auswerten. Leider ist die Ausgabe von Scheme unbefriedigend.

Abhilfe: Wir schreiben Scheme-Funktionen, welche die Rückgabewerte unseres Lambda-Kalkül in die entsprechenden Scheme-Objekte transformieren.

Programm: (Konversion logischer Werte)

```
(define (scheme-boolean lc-boolean)
   ((lc-boolean #t) #f))
```

Diese Routine kann man auf das Ergebnis des Lambda-Kalkül-Programms anwenden, um den Rückgabewert in einen Scheme-Boolean zu transformieren.

Bemerkung: Dies ist im Prinzip dasselbe wie das Ablesen eines Ergebnisses vom Band einer Turing-Maschine, oder das Auslesen eines Speicherbereichs z.B. zur Ausgabe eines Bildes.

3.3.7 Verzögerte Auswertung und `if`

Beobachtung: Wie schon bemerkt, kann man `if` in Sprachen mit Auswertung der Funktionsargumente nicht direkt als Funktion schreiben. Der Grund ist, dass dann Konstrukte wie

```
(let ((_if_ (lambda (a b c) (if a b c)))
      (x 0))
  (_if_ (= x 0) 1 (/ x)))
```

einen Fehler liefern.

Abhilfe: In Scheme kann man Auswertungen explizit verzögern. Dazu verwendet man den Operator `lambda` und die Funktionsauswertung wie folgt:

```
(let ((_if_ (lambda (a b c) (if a b c)))
      (x 0))
  ((_if_ (= x 0)
         (lambda () 1)
         (lambda () (/ x)))))
```

Bemerkung:

- Durch eine Syntax-Transformation kann man daher den üblichen `if`-Operator aus einer `_if_`-Funktion herstellen.
- Die explizite Verzögerung der Auswertung ist hin und wieder nützlich. Daher kann man sie etwas expliziter machen durch die Einführung der Syntax `delay` (implementiert durch Einwickeln in einen Lambda-Ausdruck) und der Funktion `force` (implementiert durch einfache Funktionsauswertung).

Programm: Wir erhalten daher:

```
(letrec-syntax
 ((if (syntax-rules ()
        ((if cond then else)
         ((_if_ cond
                (lambda () then)
                (lambda () else)))))))
    ;; ab hier haben wir ein richtiges if
  )
```

Bemerkung: Basierend auf `if` können folgende weitere logische Operatoren definiert werden:

```
(letrec-syntax
    ((or (syntax-rules ()
           ((or) false)
           ((or expr1 expr2 ...)
            (if expr1 true (or expr2 ...)))))
     (and (syntax-rules ()
            ((and) true)
            ((and expr1 expr2 ...)
             (if expr1 (and expr2 ...) false)))))

  ;; ab hier haben wir auch and/or
...)
```

3.3.8 Die Churchschen Numerale

Frage: Wie repräsentieren wir Zahlen in unserer Welt, die nur aus Funktionen besteht?

Antwort: Wir repräsentieren die Zahl $n \in \mathbb{N}$ als Funktional, welche angewandt auf eine Funktion f als Ergebnis die Funktion f^n zurückgibt. Die Eins wird durch die Identität dargestellt, die Null durch eine Funktion, die immer die Identität liefert. Diese Darstellung nennt man die Churchschen Numerale.

Programm:

```
(let* ((zero (constantly identity))
       (one identity)
       (succ (lambda (n)
               (lambda (f)
                  (lambda (x)
                     (f ((n f) x))))))
       (two   (succ one))      (three (succ two))
       (four  (succ three))    (five  (succ four))
       (six   (succ five))     (seven (succ six))
       (eight (succ seven))    (nine  (succ eight))
       (ten (succ nine))
       ;;
       (zero? (lambda (n) ((n (constantly false)) true))))

      ;; ab hier haben wir natürliche Zahlen
   )
```

Bemerkung: Es fehlt noch eine Funktion, die Churchsche Numerale in eine Scheme-Zahl umwandelt. Diese kann aber einfach geschrieben werden als

```
(define (scheme-natural-number n)
  ((n (lambda (x) (+ x 1))) 0))
```

3.3.9 Addition und Multiplikation

Programm:

```
(let* ((+ (lambda (n1 n2)
            (lambda (f)
              (lambda (x)
                ((n1 f) ((n2 f) x))))))
       (* (lambda (n1 n2)
            (lambda (f)
              (n1 (n2 f))))))
   ;; ab hier können wir addieren und multiplizieren
   (* nine (+ (* ten seven) three))
   )
```

3.3.10 Zusammengesetzte Daten

Programm: Wir wissen schon, dass man die Operationen `cons`, `car`, `cdr` über `lambda` ausdrücken kann:

```
(let* ((cons (lambda (a b)
               (lambda (boolean)
                 (_if_ boolean a b))))
       (car (lambda (pair) (pair true)))
       (cdr (lambda (pair) (pair false))))

 ;; ab hier können wir Daten zusammenbauen
 )
```

3.3.11 Subtraktion

Idee: Erzeuge zuerst einen Dekrementierungsoperator. Zur natürlichen Zahl n erhält man diesen durch n-fache Anwendung der Abbildung

$$\texttt{(a . b)} \mapsto \texttt{(b . succ(a))}$$

auf das Paar `(0 . 0)`. Im `car`-Teil des Ergebnisses ist dann der Vorgänger von n enthalten.

Programm:

```
(let* ((pred
        (let* ((next-pair
                (lambda (pair)
                  (cons (cdr pair)
                        (succ (cdr pair))))))
          (lambda (n)
            (car ((n next-pair) (cons zero zero))))))
       (- (lambda (n1 n2)
            ((n2 pred) n1))))

 ;; ab jetzt haben wir auch die Subtraktion
 )
```

3.3.12 Definition rekursiver Funktionen

Frage: Wie definiert man rekursive Funktionen? Mit unseren bisherigen Hilfsmitteln scheint das auf den ersten Blick unmöglich, da der Name der Funktion ja zur Zeit der Funktionsdefinition noch nicht definiert ist.

Idee: Man erweitere die Argumente der Funktion um ein Funktionsobjekt. Beim Aufruf der Funktion übergibt man dort die Funktion selbst, innerhalb der Funktionsdefinition kann das Argument verwendet werden.

Beispiel: Die rekursive Fakultät könnte mit dieser Idee wie folgt geschrieben werden:

```
(let* ((factorial-helper
        (lambda (f n)
          (if (zero? n)
              one
              (* n (f f (pred n))))))
       (factorial
        (lambda (n)
          (factorial-helper factorial-helper n))))
  ;; use of factorial
  (factorial five))
```

Bemerkung: Wieder kann eine Syntax-Transformation verwendet werden, um solche rekursiven Definitionen schöner zu machen.

Programm: (`recursive`)

```
(let-syntax
  ((recursive
    (syntax-rules ()
      ((recursive () body ...)
       (let () body ...))
      ((recursive ((name (arg1 ...) subbody ...)) body ...)
       (let* ((helper
               (lambda (helper arg1 ...)
                 (let ((name (lambda (arg1 ...)
                               (helper helper arg1 ...))))
                   subbody ...)))
              (name
               (lambda (arg1 ...)
                 (helper helper arg1 ...))))
         body ...)))))
 ;; ab hier können wir rekursive Funktionen definieren
 )
```

3.3.13 Die rekursive Fakultät

Programm: Die rekursive Fakultät kann nun wie folgt definiert werden

```
(recursive ((factorial (n)
              (if (zero? n)
                  one
                  (* n (factorial (- n one))))))
  ;; ein Aufruf
  (factorial ten))
```

3.3.14 Was fehlt?

Beobachtung: Einiges fehlt unserer Sprache noch. Dazu zählen insbesondere

- Introspektion, z.B. `number?`, `boolean?`, `pair?`, ...
- Der Aufbau des Zahlsystems: ganze Zahlen, rationale Zahlen, etc
- Weitere Datentypen (Vektoren, Strings, etc)

Bemerkung: Insbesondere der erste Punkt ist interessant. Seine Verwirklichung erfordert eine neue Schicht von Abstraktionen, die die bisher erhaltenen ablösen und auf getypten Daten definiert sind. Die Implementation dieser neuen Abstraktionen kann aber auf den Operatoren und Datenstrukturen aus der von uns implementierten Schicht aufbauen! Zum Beispiel könnte als zugrundeliegende Datenstruktur der neuen Schicht ein Paar der Form `(Typ . Datum)` verwendet werden.

3.4 Symbole

Bisher: Nur Zahlen und `#t/#f` waren elementare Datentypen.

Jetzt: Beliebige Symbole als elementare Objekte.

3.4.1 Quotierung

Ziel: Erzeugen einer *Liste* `(+ 1 2)` (das Symbol + zusammen mit den Zahlen 1 und 2) oder `(a b)` (zwei Symbole). Um dies zu erhalten braucht man eine neue Sonderform namens `quote`.

Syntax:

```
(quote <expr>)
```

Liefert `<expr>`, ohne es auszuwerten!
Für `(quote <expr>)` gibt es die Abkürzung `'<expr>`. Der Apostroph und der folgende Ausdruck werden bereits vom Reader in `(quote <expr>)` transformiert.

Beispiele:

```
'(Apfel Birne Banane)
'Hase
'(+ 2 3)
'(define (factorial n)
    (if (= n 0)
        1
        (* n (factorial (- n 1)))))
```

Syntax: Ein Vergleich von Symbolen ist mit `eq?` möglich:

```
(eq? 'b 'a)
(eq? x 'a)
```

Bemerkung: Wenn man Variablen vergleichen will, die nicht nur Symbole sondern auch Zahlen enthalten können, sollte man stattdessen den Operator `eqv?` verwenden. Wenn man zusätzlich Listen rekursiv auf die Gleichheit ihrer Elemente vergleichen möchte, muss man `equal?` verwenden. Dieser Operator könnte wie folgt definiert sein:

```
(define (equal? tree1 tree2)
  (if (pair? tree1)
      (if (pair? tree2)
          (and (equal? (car tree1) (car tree2))
               (equal? (cdr tree1) (cdr tree2)))
          #f)
      (eqv? tree1 tree2)))
```

3.4.2 Strings und Symbole

Syntax: Zeichenketten erhält man in Scheme durch Einschließen in Anführungszeichen. (Wenn ein Anführungszeichen selbst im String enthalten sein soll, muss man ihm einen Backslash voranstellen.)

Beispiel: `"Hello, world!"`

Bemerkung: Für Symbole gab es im wesentlichen nur den Operator eq?. Im Gegensatz dazu gibt es für Strings viele Funktionen: `string-length`, `string-ref`, `string=?`, `string<=`, ...

Bemerkung: Nur selten ist die Manipulation von Symbolnamen notwendig. Dafür gibt es die Funktionen `symbol->string` und `string->symbol`.

Bemerkung: MzScheme unterscheidet bei Symbolnamen normalerweise Groß- und Kleinschreibung. In manchen Lisp-Varianten (hier ist insbesondere Common Lisp zu erwähnen) werden Symbolnamen im Normalfall beim Einlesen in Großschreibung konvertiert.

3.4.3 Mengen

Absicht: Darstellung von Mengen mit folgenden Operationen:

- `(element? x set)`
- `(adjoin x set)`
- `(remove x set)`
- `(union set1 set2)`
- `(intersection set1 set2)`

Idee: Implementiere eine Menge als Liste, in der jedes Element nur einmal vorkommt.

3.4.4 Implementation 1: als ungeordnete Liste

Programm:

```
(define (element? x set)
  (cond ((null? set) #f)
        ((equal? (car set) x) #t)
        (else (element? x (cdr set)))))

(define (adjoin x set)
  (if (element? x set)
      set
      (cons x set)))
```

```
(define (intersection set1 set2)
  (filter (lambda (x) (element? x set2))
          set1))

(define (union set1 set2)
  (append (remove-if (lambda (x) (element? x set2)) set1)
          set2))
```

Bemerkung: Der Aufwand der einzelnen Operationen ist (jeweils im schlimmsten Fall): `element?`: O(n), `adjoin`: O(n), `intersection`: $O(n_1 n_2)$, `union`: $O(n_1 n_2)$.

Bemerkung: Es wurden die Nicht-Standard-Funktionen `filter` und `remove-if` verwendet. Diese und andere nützliche Funktionen findet man in `useful.scm`. Diese Bibliothek laden wir anfangs mit Hilfe des Befehls

```
(load "useful.scm")
```

3.4.5 Implementation 2: als geordnete Liste

Voraussetzung: Auf den Elementen gibt es eine totale Ordnung, d.h.

a) $a \leq b, b \leq c \Rightarrow a \leq c$ (Transitivität)

b) $a \leq b, b \leq a \Rightarrow a = b$ (Identivität)

c) $a \leq b \vee b \leq a$ (Totalität)

Programm:

```
(define (element? x set)
  (cond ((null? set) #f)
        ((< x (first set)) #f)
        ((= x (first set)) #t)
        (else (element? x (rest set)))))

(define (adjoin x set)
  (if (element? x set)
      set
      (let rec ((set set))
        (cond ((null? set) (list x))
              ((< x (first set)) (cons x set))
              (else (cons (first set) (rec (rest set))))))))

(define (intersection set1 set2)
  (if (or (null? set1) (null? set2))
```

```
      ()
      (let ((x1 (first set1))
            (x2 (first set2)))
        (cond ((= x1 x2)
               (cons x1 (intersection (rest set1) (rest set2))))
              ((< x1 x2) (intersection (rest set1) set2))
              ((< x2 x1) (intersection set1 (rest set2)))
              (else (error "no total ordering"))))))

(define (union set1 set2)
  (cond ((null? set1) set2)
        ((null? set2) set1)
        (else
         (let ((x1 (first set1))
               (x2 (first set2)))
           (cond ((= x1 x2)
                  (cons x1 (union (rest set1) (cdr set2))))
                 ((< x1 x2) (cons x1 (union (rest set1) set2)))
                 ((< x2 x1) (cons x2 (union set1 (rest set2))))
                 (else (error "no total ordering"))))))))
```

Bemerkung: Hier ist vor allem der Aufwand von `intersection` und `union` nur noch von der Ordnung $O(n_1 + n_2)$.

3.4.6 Implementation 3: Mengen als Binärbäume

Bemerkungen:

- Auch hier ist eine totale Ordnung notwendig.
- Der Aufwand von `element?` und `adjoin` ist $O(\log n)$, *falls der Baum balanciert gehalten wird*.
- `union` und `intersection` können immer noch in $O(n_1+n_2)$ Operationen durchgeführt werden, weil man zwischen Bäumen und geordneten Listen in $O(n)$ Operationen konvertieren kann.

3.5 Mehrfachdarstellung abstrakter Daten

3.5.1 Einführung

Problem: Wir können nicht beide Darstellungen gleichzeitig verwenden, es sei denn, wir benennen unsere Funktionen um: z.B. `unordered-list-element?`, `ordered-list-union`,

. . . Dies ist aber umständlich und wird spätestens dann Probleme bereiten, wenn mehrere Gruppen an verschiedenen Darstellungen arbeiten.

Abhilfe: Wir heften den Daten Identifikationsmarken an. Eine globale (generische Funktion entscheidet dann, was mit dem jeweiligen Argument zu tuns ist

Bemerkungen:

- Dieser Zugang über generische Funktionen ist eine Variante von datengesteuerter oder objektorientierter Programmierung.
- Eine Alternative besteht darin, einem Objekt (z.B. einem Binärbaum) eine Reihe von Operationen mitzugeben, wie er mit anderen z.B. die Vereinigung (`union`) bilden kann.
- Diese letztere Variante kann man über Botschaften, die an ein Objekt geschickt werden modellieren (message-passing-Modell). Sie wird z.B. in Smalltalk, C++ oder Java verwendet.
- Die Variante mit generischen Funktionen ist flexibler, weil die Operation in Abhängigkeit von mehreren Parametern variiert werden kann.

3.5.2 Bezeichnete (etikettierte) Daten

Idee: Wir verwenden ein Paar zur Repräsentation eines etikettierten Datums. Der car-Teil enthält Typinformation in der Form (`<object> . <type>`). Der cdr-Teil enthält das Datum.

Programm:

```
(let ((object-id '<object>))

  (define (attach tag obj)
    (cons (cons object-id tag) obj))

  (define (user-defined-type? obj)
    (and (pair? obj)
         (pair? (car obj))
         (eq? (caar obj) object-id)))
  )

(define (type obj)
  (cond ((number? obj) '<number>)
        ((symbol? obj) '<symbol>)
        ((pair? obj)
         (if (user-defined-type? obj)
             (cdar obj)
```

```
            '<pair>))
        (else '<unknown-type>)))

(define (contents obj)
  (if (user-defined-type? obj)
      (cdr obj)
      obj))
```

Programm: Für ungeordnete und geordnete Listen könnte man jetzt folgende Konstruktoren definieren:

```
(define (make-unordered-list lst)
  (attach '<unordered-list> lst))

(define (make-ordered-list lst)
  (if (or (null? lst) (single? lst) (apply <= lst))
      (attach '<ordered-list> lst)
      (error "list is not ordered")))
```

Programm: Auch Operationen wie `adjoin` müssten wieder etikettierte Mengen zurückliefern:

```
(define (unordered-list-adjoin x lst)
  (lambda (x lst)
    (make-unordered-list
     (adjoin x (contents lst)))))
```

Probleme:

- Man braucht immer noch unterschiedliche Namen für jeden Mengentyp.
- Der Zugriff auf `contents` ist (vielleicht) unschön.

3.5.3 Typabhängige Zuteilung

Verwendung: Verwendet werden könnte die Etikettierung nun in folgender Weise:

```
(define (adjoin x set)
  ((cond ((eq (type set) '<unordered-list>)
          unordered-list-adjoin)
         ((eq (type set) '<ordered-list>)
          ordered-list-adjoin)
         ((eq (type set) '<binary-tree>)
          binary-tree-adjoin))
   x set))
```

Bezeichnung: Diese Art der Methodenauswahl nennt man type dispatch (typabhängige Auswahl).

Probleme: Das Hinzufügen von neuen Mengendarstellungen ist nicht möglich.

3.5.4 Datengesteuerte Programmierung und Additivität

Annahme: Wir haben Tabellierungsoperationen im System

- `(put-method! operator types method)`
- `(get-method operator types)`

mit denen wir die jeweiligen Methoden definieren und wieder abrufen können.

Bemerkung: Wir laden diese Operationen mit der Datei `generic-functions.scm`, ohne genauer darauf einzugehen. Sie passen nicht so ganz in unsere bisherige funktionale Welt (genau wie z.B. der eingebaute Operator `define`), weil sie offenbar irgendeine versteckte Tabelle verändern müssen.

Anwendung: Mit dieser Funktionalität kann man dann z.B. folgendes definieren:

```
(put-method! 'element? '(<number> <unordered-list>)
             unordered-list-element?)
(put-method! 'element? '(<number> <ordered-list>)
             ordered-list-element?)

(put-method! 'adjoin '(<number> <unordered-list>)
             (lambda (x lst)
               (make-unordered-list
                (adjoin x lst))))
 ...
```

Programm: Den Typ-Dispatch kann man dann folgendermaßen durchführen:

```
(define (apply-generic operator . args)
  (let ((proc (get-method operator (map type args))))
    (if proc
        (apply proc (map contents args))
        (error "no method for: " operator args))))

(define (element? x set)
   (apply-generic 'element? x set))
(define (adjoin x set)
   (apply-generic 'adjoin x set))
(define (intersection set1 set2)
   (apply-generic 'intersection set1 set2))
(define (union set1 set2)
   (apply-generic 'union set1 set2))
```

Bemerkung: Man beachte, dass man jetzt bei der Methodendefinition kein `contents` mehr auf die Argumente anwenden muss. Am besten definiert man nun Methoden für eine bestimmte Mengendarstellung als interne Funktionen und fügt diese dann der Funktionalität der globalen generischen Funktion hinzu.

Beispiel:

```
(define (install-unordered-list-generic-functions)

  ;; initial procedures for unordered lists (unchanged!)

  (define (element? x set)
    (cond ((null? set) #f)
          ((equal? (car set) x) #t)
          (else (element? x (cdr set)))))

  (define (adjoin x set)
    (if (element? x set)
        set
        (cons x set)))

  ;; [etc]

  ;; interface to the external system

  (put-method! 'element? '(<number> <unordered-list>) element?)
  (put-method! 'adjoin '(<number> <unordered-list>)
               (lambda (x lst)
                 (make-unordered-list
                  (adjoin x lst))))

  ;; [etc]
  )
```

3.6 Strukturierte Daten: Klassen/Strukturen

3.6.1 Einführung

Frage: Wie repräsentieren wir komplexe Objekte mit mehreren Eigenschaften?

Beispiel:

- Rechteck: Breite, Höhe (Position, Drehwinkel)
- Kreis: Radius (Mittelpunkt)

Varianten:

- Listen, z.B. `'(4.0 3.1)` könnte Breite und Höhe eines Rechtecks bezeichnen. Problem: Code muss wissen, was es bedeutet und an welcher Position welches Datum steht.
- Assoziationslisten: `'((width . 4.0) (height . 3.1))` und Zugriff z.B. über

```
(define (width object)
  (cdr (assq object 'width)))
```

  Hier ist die Position der Daten innerhalb der Liste flexibel. Nachteile sind aber:
  - Mehr Speicherplatz
  - Der Code muss immer noch die Bedeutung einer solchen Liste kennen.
- Etikettierung der Daten: man versieht die Liste `(4.0 3.1)` mit einem Etikett, dass die Daten beschreibt.

Bemerkungen:

- Am geschicktesten ist es, wenn die Etikettierung selber ein Datenobjekt ist, welches unter anderem die Slotnamen `(width height)` enthält. Dieses Datenobjekt nennen wir Klasse oder auch Struktur.
- Der Unterschied zur Etikettierung des letzten Abschnitts ist, dass wir eine ganze Gruppe von Daten zu einer Einheit zusammenfassen und die einzelnen Bestandteile wieder selektieren können.

3.6.2 Implementation

Idee: Beschreibe eine Klasse als Liste `(class-name slot-names)`.

Programm: (Klassenerzeugung)

```
;;; meta class
(define <class>
  (list '<class> '(my-name my-slots)))

(define (class-name class) (second class))
(define (class-slots class) (third class))

;;; class generation

(define (make-class name slots)
  (list <class> name slots))

(define-syntax define-class
  (syntax-rules ()
    ((define-class name slots)
     (define name
       (make-class 'name 'slots)))))

(define (class? obj)
  (and (pair? obj) (eq? (car obj) <class>)))

;;; some built-in types

(define-class <boolean> ())
(define-class <number> ())
(define-class <pair> ())

(define (class-of obj)
  (cond ((number? obj) <number>)
        ((boolean? obj) <boolean>)
        ((pair? obj)
         (if (class? (first obj))
             (first obj)
             <pair>))
        (else (error "unknown object"))))


;;; access to slot values

(define (slot-value obj name)
  (let ((pos (position name (class-slots (class-of obj)))))
    (if pos
        (list-ref (rest obj) pos)
        (error "unknown slot"))))
```

```
(define ^ slot-value)


;;; example of use
(define <rectangle>
  (make-class '<rectangle> '(width height)))

(define-class <rectangle> (width height))
```

Bemerkung: `(list-ref list n)` selektiert das `n`-te Element von `list`. `position` findet die Position eines Elements in einer Liste.

3.6.3 Objekterzeugung

Programm: Mit der Funktion `make` können wir nun Objekte einer Klasse erzeugen:

```
(define (make class slot-initializations)
  (cons class
        (map (lambda (name)
               (let ((slot-def (assq name slot-initializations)))
                 (if slot-def
                     (cdr slot-def)
                     'slot-unbound)))
             (class-slots class))))
```

3.6.4 Slotzugriff

Programm: Über `slot-value` (mit Abkürzung `^`) können wir den Wert von Datenfeldern (slots) auslesen:

```
(define (slot-value obj name)
  (let ((pos (position name (class-slots (class-of obj)))))
    (if pos
        (list-ref (rest obj) pos)
        (error "unknown slot"))))

(define ^ slot-value)
```

3.6.5 Vererbung

Beobachtung: Oftmals sind Klassen Sonderfälle von allgemeineren Klassen, welche sich auszeichnen durch

- zusätzliche Slots,
- andere Eigenschaften (bei der Verwendung von generischen Funktionen),
- wegfallende oder spezialisierte Slots (z.B. Quadrat als Spezialfall eines Rechtecks, reelle Zahlen als Spezialfall von komplexen Zahlen).

In solchen Fällen ist es nützlich, wenn man Klassen- und Methodendefinitionen nicht völlig neu aufbauen muss, sondern wenigstens Teile davon übernehmen kann.

Bemerkung: Im folgenden stellen wir einen Mechanismus vor, der wenigstens die ersten beiden Punkte erreicht. Den dritten Punkt könnte man z.B. durch automatische Transformationen wie Quadrat → Rechteck beheben, oder indem man die Funktionalität des Rechtecks in eine abstrakte Basisklasse steckt.

Anwendung: Man könnte eine Klasse `<rectangle>` (Breite, Höhe) mit einer Klasse `<graphics>` (Farbe, Transparenz) kombinieren, so dass am Ende eine Klasse `<graphics-rectangle>` entsteht.

Bemerkung: Diese Art der Vererbung heißt Mehrfachvererbung, im Gegensatz zur Einfachvererbung, die nur von einer Klasse ableitet. Mehrfachvererbung wird z.B. von CLOS und C++ bereitgestellt. Java dagegen stellt aus Gründen der Einfachheit nur Einfachvererbung zur Verfügung.

Implementation: Wir führen einen zusätzlichen Slot `super-classes` in die Klassendefinition ein. Dieser enthält dann eine Liste der Elternklassen. Alle Slots der Klasse ergeben sich durch geeignete Vereinigung der Elternslots mit den eigenen.

Programm: Wir müssen die obigen Teile hierzu leicht modifizieren und erweitern:

```
;;; meta objects
(define <class>
  (list #t '<class>
        ()
        '(my-name super-classes direct-slots)))

(set-car! <class> <class>)  ; we close the set of classes

;; the type <top> fits for everything
(define <top> (list <class> '<top> () ()))

;;; classes are implemented as lists
(define (class-name class) (second class))
(define (class-supers class) (third class))
(define (class-direct-slots class) (fourth class))

;;; class generation
```

```
(define (ordered-union set1 set2)
  "(ordered-union '(1 3 2) '(1 2 3)) => (1 3 2)"
  (cond ((null? set1) set2)
        ((null? set2) set1)
        ((eqv? (first set1) (first set2))
         (cons (first set1)
               (ordered-union (rest set1) (rest set2))))
        ((not (member (first set1) set2))
         (cons (first set1)
               (ordered-union (rest set1) set2)))
        ((not (member (first set2) set1))
         (cons (first set2)
               (ordered-union set1 (rest set2))))
        (else ; decide for set1
         (cons (first set1)
               (ordered-union (rest set1)
                              (remove (first set1) set2))))))

(define (class-slots class)
  "Compute all class slots."
  (ordered-union (class-direct-slots class)
                 (reduce ordered-union (class-supers class)
                         () class-direct-slots)))

(define (class-precedence-list class)
  "Compute the class precedence list."
  (ordered-union (list class)
                 (reduce ordered-union (class-supers class)
                         (list <top>) class-precedence-list)))

(define (make-class name supers direct-slots)
  (list <class> name supers direct-slots))

(define-syntax define-class
  (syntax-rules ()
    ((define-class name (super ...) direct-slots)
     (define name
       (make-class 'name (list super ...) 'direct-slots)))))

(define (class? obj)
  (and (pair? obj)
       (eq? (car obj) <class>)))

(define (standard-object? obj)
```

```
  (and (pair? obj)
       (class? (car obj))))

;;; standard classes

(define-class <boolean> () ())
(define-class <number> () ())
(define-class <pair> () ())

(define (class-of obj)
  (cond ((number? obj) <number>)
        ((boolean? obj) <boolean>)
        ((pair? obj)
         (if (class? (first obj))
             (first obj)
             <pair>))
        (else (error "unknown object"))))

(define (subtype? class1 class2)
  (if (member class2 (class-precedence-list class1))
      #t #f))

;;; object generation

(define (make class slot-initializations)
  (cons class
        (map (lambda (name)
               (let ((slot-def (assq name slot-initializations)))
                 (if slot-def
                     (cdr slot-def)
                     'slot-unbound)))
             (class-slots class))))

;;; access to slot values

(define (slot-value obj name)
  (let ((pos (position name (class-slots (class-of obj)))))
    (if pos
        (list-ref (rest obj) pos)
        (error "unknown slot"))))

(define ^ slot-value)  ; abbreviation
```

Bemerkungen:

- Wir bezeichnen Klassen mit <...>. Das ist eigentlich ganz nett und in Scheme wohl auch üblich. In Common Lisp ist es nicht üblich.
- Die Verwendung der Zuweisung `set-car!` um `<class>` wieder zu einem Objekt zu machen, könnte man vermeiden, wenn man in einigen Routinen (nämlich `class?`, `class-of`, `standard-object?`) den Fall `<class>` speziell abfängt.

3.6.6 Inspektion

Programm: Wir wollen eine strukturierte Ausgabe von Klassen:

```
(define (describe object)
  (display "This is an object of class ")
  (display (class-name (class-of object)))
  (display ".  ")
  (cond ((standard-object? object)
         (display "Its slots are:")
         (for-each (lambda (slot)
                     (newline)
                     (display slot)
                     (display " -> ")
                     (display (^ object slot)))
                   (class-slots (class-of object))))
        (else
         (display "Its value is: ")
         (display object)))
  (newline))
```

3.6.7 Generische Funktionen

Schwierigkeit: Die Methodendefinition für generische Funktionen spezifiziert nun wieder bestimmte Datentypen der Argumente. Es passen auch alle Datentypen der Argumente, deren Klassen sich davon ableiten. Bei Mehrfachvererbung muss man aber folgende Situation entscheiden: Klasse A hat als Elternklassen B und C, und Methoden sind sowohl für B als auch für C spezifiziert (oder für Eltern von diesen). Welche hat Vorrang?

Abhilfe: Die Funktion `ordered-union` die zwei geordnete Listen geordnet vereinigt. Diese wird verwendet um eine Klassenpräzedenzliste (`class-precedence-list`) aufzubauen. Die Position in dieser Liste entscheidet, welche Methode bevorzugt wird.

Programm:

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; generic functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
;;; global table
(define *generic-function-methods* ())

(define-class <method> ()
  (operator types procedure))

(define (insert-method! operator types procedure)
  (set! *generic-function-methods*
        (cons (make <method>
                `((operator . ,operator)
                  (types . ,types)
                  (procedure . ,procedure)))
              (remove-if (lambda (method)
                           (and (eq? (^ method 'operator) operator)
                                (equal? (^ method 'types) types)))
                         *generic-function-methods*))))

(define (get-applicable-methods operator types)
  "Returns the applicable methods sorted with most-specific first."
  (sort (curry every subtype?)
        (filter (lambda (method)
                  (and (eq? (^ method 'operator) operator)
                       (every subtype? types (^ method 'types))))
                *generic-function-methods*)
        (rcurry ^ 'types)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; generic function application
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(define (apply-generic operator . args)
  (let ((methods (get-applicable-methods operator (map class-of args))))
    (if (null? methods)
        (error "No applicable methods: "
               operator (map class-name (map class-of args)))
        (apply (^ (first methods) 'procedure) args))))

;;; and an improved syntax

(define-syntax define-generic
  (syntax-rules ()
    ((define-generic (name arg ...))
```

```
      (define (name arg ...)
        (apply-generic 'name arg ...)))))

(define-syntax define-method
  (syntax-rules ()
    ((define-method (name (arg type) ...)
       body ...)
     (insert-method! 'name (list type ...)
                     (lambda (arg ...) body ...)))))
```

3.6.8 Anwendung

Programm: Wir definieren die geometrischen Objekte Rechteck und Kreis. Dann definieren wir eine generische Funktion, welche die Fläche berechnet und zugehörige Methoden. Als nächstes fügen wir Graphikinformation zu den Rechtecken hinzu, und modifizieren die Methode für die abgeleitete Klasse.

```
(define-class <rectangle> () (width height))
(define-class <circle> () (radius))

(define-generic (area object))

(define-method (area (rect <rectangle>))
  (* (^ rect 'width) (^ rect 'height)))

(define-method (area (circ <circle>))
  (* pi (expt (^ circ 'radius) 2)))

(define rect
  (make <rectangle>
    '((width . 5) (height . 10))))
(area rect)

(define circ (make <circle> '((radius . 4.0))))
(area circ)


(define-class <graphics> () (color transparency))
(define-class <graphics-rectangle> (<rectangle> <graphics>)
  ())

(define grect
  (make <graphics-rectangle>
    '((width . 10)
```

```
      (height . 5)
      (color . red)
      (transparency . no))))

(define-method (area (grect <graphics-rectangle>))
  "Has a border of size 1."
  (* (+ (^ grect 'width) 2)
     (+ (^ grect 'height) 2)))

(area grect)
(area rect)

(describe grect)
```

3.6.9 Beziehung zu CLOS/TinyCLOS

Bemerkungen:

- In Common Lisp wurden gegen Ende der 80er Jahre mehrere derartige OO-Systeme ausprobiert, teilweise auf dem Message-Passing Zugang basierend, teilweise auf dem Zugang über generische Funktionen. CLOS (Common Lisp Object System) ist eine Synthese dieser Bibliotheken, wobei sich der allgemeinere Rahmen der generischen Funktionen durchgesetzt hat.
- Genau wie unser kleines OO-System war es auch damals möglich, CLOS als Bibliothek in Lisp zu implementieren.
- CLOS ist Bestandteil des ANSI Common Lisp Standard (1994). Obwohl man Lisp eher mit funktionalem Programmieren verbindet, war Common Lisp dadurch auch die erste standardadisierte OO-Sprache.
- Die Basisfunktionalität von CLOS ist in etwa dieselbe wie in unserem System. Wichtige Erweiterungen sind aber: Methodenkombination, automatische Definition von Schreib-, Lese-Methoden für Slots, Klassenänderung.
- Für CLOS gibt es eine Meta-Ebene (Klassen und generische Funktionen sind dort Metaobjekte, die in ihrem Verhalten in gewissen Grenzen variabel sind). Die Schnittstelle dieser Metaebene ist teilweise im CL-Standard definiert und noch tiefer gehend in einem einflussreichen Buch ("The Art of the Metaobject Protocol") beschrieben. Damit wird objektorientiertes Programmieren selbst programmierbar.
- CLOS wurde intensiv optimiert. Als Ergebnis stellt Objektorientiertheit trotz der großen Flexibilität selten den Flaschenhals in realistischen Anwendungen dar. Die ohne wesentliche Flexibilitätseinschränkungen erreichbare Effizienz ist dabei etwa so: Zugriffe auf Slots und Aufrufe von generische Funktionen entsprechen in etwa einem normalen Funktionsaufruf.

- Selbst diese Effizienzgrenze kann man durchbrechen, allerdings unter Opfern von Flexibilität. Wenn man Klassendefinitionen „einfriert“, ist es z.B. möglich Slot-Zugriffe perfekt durch Inlining zu eliminieren. Dieser Zugang wurde in der Programmiersprache Dylan versucht. Diese Sprache wurde bei Apple entwickelt, kam aber durch den von SUN finanzierten Java-Hype nie recht in Fahrt.

4 Veränderbare Objekte und Zustände

4.1 Zuweisung und lokaler Zustand

4.1.1 Einführung

Beobachtung: Bisher waren unsere Programme fast ausschließlich im funktionalen Stil geschrieben, d.h. das Ergebnis einer Funktionsauswertung hing nur von den Argumenten und etwaigen Werten für freie Variablen ab.

Ausnahmen:

- Tabellen in `object.scm`.
- Verwendung des globalen `define`.

Bemerkung: Obwohl wir funktional viel erreicht haben, sind sich verändernde Datenobjekte allgegenwärtig in der realen Welt. Wir wollen daher jetzt die erweiterten Ausdrucksmöglichkeiten durch das Zulassen von veränderlichen Daten studieren.

4.1.2 Beispiel

Programm: (Modellierung eines Kontos)

```
(define balance 100)

(define (withdraw amount)
  (set! balance (- balance amount)))

(define (deposit amount)
  (set! balance (+ balance amount)))
```

Nachteile:

- Nur ein Konto
- Unkontrollierter Zugriff

Programm: (Bessere Implementation)

```
(define (konto balance)
  (define (withdraw amount)
    (set! balance (- balance amount)))
  (define (deposit amount)
    (set! balance (+ balance amount)))
  (lambda (message)
    (case message
      ((withdraw) withdraw)
      ((deposit) deposit)
      ((show) balance))))

(define mein-konto (konto 100))

((mein-konto 'withdraw) 20)

(mein-konto 'show)
```

Bemerkung: Die „Funktion“ `mein-konto` verhält sich anders als alle bisher bekannten Funktionen, weil sie nicht nur von ihren Argumenten, sondern auch von der versteckten Kontobilanz `balance`. Aufrufe mit gleichen Argumenten können daher andere Ergebnisse liefern, z.B.

```
(mein-konto 'show)
((mein-konto 'withdraw) 20)
(mein-konto 'show)
```

4.1.3 Monte-Carlo-Berechnung von π

Idee: Wir approximieren π, indem wir n zufällige Vektoren in $[0,1]^2$ erzeugen und zählen, wie viele davon innerhalb des Kreissegments zu liegen kommen. π ist dann das Vierfache dieses Anteils.

Programm:

```
(define (random-vector dim)
  (if (zero? dim)
      ()
      (cons (random) (random-vector (1- dim)))))

(define (norm vec)
  (sqrt (reduce + vec 0.0 square)))

(define (approximate-pi trials)
  (let loop ((i 0)
             (positive 0))
```

```
(if (= i trials)
    (* (/ positive trials) 4.0)
    (loop (1+ i)
          (if (< (norm (random-vector 2)) 1)
              (1+ positive)
              positive)))))

(approximate-pi 100000)
```

Bemerkung: Ähnlich dem Konto sind die Werte von `random` nicht durch die Argumente bestimmt (es hat ja nicht mal welche).

Frage: Was sind Vorteile und Nachteile der zufälligen Punktwahl gegenüber einem regelmäßigen Gitter auf dem man eine ähnliche Integration durchführen könnte?

Frage: Was hat `random` mit Zuweisung zu tun?

Antwort: Die Generierung von Zufallszahlen arbeitet mit einem internen Zustand, der beim Aufruf verändert wird

4.1.4 Problematik der Zuweisung

Probleme:

- Das Substitutionsmodell (Variable ist ein Name für einen Wert) reicht nicht mehr aus. (Was ist der Wert von `balance` innerhalb `mein-konto`?)
- Gleichheit ist unklar: identisch erzeugte Konten

  ```
  (define mein-konto (konto 100))
  (define dein-konto (konto 100))
  ```

  können sich später unterschiedlich verhalten.
- Es kommt oft auf die Reihenfolge von Operationen an.

  ```
  (define (imperative-factorial n)
    (let ((k 1)
          (f 1))
      (let loop ()
        (if (> k n)
            f
            (begin
              ;; Reihenfolge?
              (set! f (* k f))
              (set! k (+ k 1))
              (loop))))))
  ```

Andererseits: Die Zuweisung erweitert die Ausdrucksfähigkeit einer Sprache erheblich. Die Veränderbarkeit der Umgebung ist praktisch immer nützlich, und auch in anderen Fällen kann die Veränderung von Daten die naheliegendste Wahl sein, um einen Zweck zu erreichen. In solchen Fällen erfordert das Insistieren auf einem funktionalen Stil Klimmzüge wie z.B. Monaden.

4.2 Das Umgebungsmodell

- Im Umgebungsmodell ist eine Variable ein Name für einen *Ort*, an dem ein Wert abgelegt werden kann.
- Variablen werden in sogenannten Umgebungen (environments) verwaltet.
- Eine solche Umgebung besteht dabei aus hierarchisch angeordneten Rahmen (frames), die Tabellen mit der Zuordnung Name – Ort speichern.
- Der Wert einer Variablen *bezüglich einer Umgebung* ist der Wert, den sie in dem ersten Rahmen der Umgebung hat, in der sie gebunden ist (bound variable).
- Eine in keinem Rahmen gebundene Variable heißt freie Variable.
- `define` erzeugt eine neue Bindung.
- `set!` ändert bestehende Bindungen.
- Funktionsdefinitionen geschehen immer bezüglich einer bestimmten Umgebung. In genau dieser Umgebung wird die Funktion später auch ausgewertet.
- Lokale Definitionen sind in der globalen Umgebung nicht sichtbar.

4.3 Veränderbare Daten

4.3.1 Veränderung von Paaren/Listen

Syntax: `set-car!` (bzw. `set-cdr!`) setzt den `car` (bzw. `cdr`) eines Paars auf einen neuen Wert.

Beispiel:

```
(define x (cons 1 2))
(set-car! x 3)
(car x)    ; => 3
```

Bemerkungen: Auch Listenstrukturen mit Zyklen kann man auf diese Weise erstellen.

Anwendung: (Destruktive Version von `append`)

```
(define (my-append! x y)
  (set-cdr! (last-pair x) y))

(define (last-pair x)
  (if (null? (cdr x))
      x
      (last-pair (cdr x))))

(define x (list 1 2))
(define y (list 3 4))
(my-append! x y)
```

Bemerkung: Funktionen, die ihre Argumente verändern können, bezeichnet Scheme mit einem Ausrufezeichen. In Common Lisp ist so eine unterschiedliche Bezeichnung nicht ganz konsequent durchgehalten. Viele derartige Operatoren beginnen aber mit „n“ für *non-consing*. Das Pendant zu `append!` heißt etwa `nconc`. (Die unangenehmste Namensgebung ist `sort`, da dieser Befehl sein Argument (leider) ändern darf.)

Aufgabe: Schreiben Sie `reverse!`, welches eine Liste auf der Stelle umkehrt.

Aufgabe: Schreiben Sie eine Funktion, welche prüft, ob eine Liste einen Zyklus enthält. Geht das auch mit einem zusätzlichen Speicheraufwand von $O(1)$?

4.3.2 Veränderung und Zuweisung

Beobachtung: Ebenso wie man zusammengesetzte Daten funktional zusammensetzen konnte, kann man auch `set-car!` und `set-cdr!` mit Hilfe von `set!` erreichen:

```
(define (kons x y)
  (lambda (message)
    (case message
      ((kar) x)
      ((kdr) y)
      ((set-kar!) (lambda (value) (set! x value)))
      ((set-kdr!) (lambda (value) (set! y value))))))

(define (kar z) (z 'kar))
(define (kdr z) (z 'kdr))
(define (set-kar! x value)
  ((x 'set-kar!) value))
(define (set-kdr! x value)
  ((x 'set-kdr!) value))

(define x (kons 1 2))
(set-kar! x 3)
(kar x)
```

Andererseits ist `set!` von der Implementation her die Änderung einer Tabelle (die man z.B. mit `set-car!` und `set-cdr!` verändern könnte).

4.3.3 Anwendung: Warteschlangen

Programm:

```
;;; Konstruktor
(define (make-queue) (cons () ()))

;;; Selektoren
(define front-ptr car)
(define rear-ptr cdr)

;;; ... und Mutatoren
(define (set-front-ptr! queue item)
  (set-car! queue item))
(define (set-rear-ptr! queue item)
  (set-cdr! queue item))

;;; Andere Methoden

(define (empty? queue)
  (null? (front-ptr queue)))

(define (front queue)
  (if (empty? queue)
      (error "queue empty")
      (car (front-ptr queue))))

(define (insert! queue item)
  (let ((new-pair (list item)))
    (cond ((empty? queue)
           (set-front-ptr! queue new-pair)
           (set-rear-ptr! queue new-pair))
          (else
           (set-cdr! (rear-ptr queue) new-pair)
           (set-rear-ptr! queue new-pair)))))

(define (delete! queue)
  (if (empty? queue)
      (error "queue empty")
      (set-front-ptr! queue (cdr (front-ptr queue)))))
```

4.3.4 Objektorientiert und Zuweisung

Syntax: Wir führen einen schreibenden Slot-Zugriff ein:

```
(define (set-slot-value! obj name value)
  (let ((pos (position name (class-slots (class-of obj)))))
    (if pos
        (set-car! (list-tail (rest obj) pos) value)
        (error "unknown slot"))))
```

Anwendung: (Queues objektorientiert)

```
(load "object.scm")

(define-generic (empty? x))
(define-generic (front x))
(define-generic (insert! x value))
(define-generic (delete! x))

(define-class <queue> () (front-ptr rear-ptr))

(define (make-queue)
  (make <queue>
    '( (front-ptr . ()) (rear-ptr . ()))))

(define-method (empty? (queue <queue>))
  (null? (^ queue 'front-ptr)))

(define-method (front (queue <queue>))
  (if (empty? queue)
      (error "queue empty")
      (car (^ queue front-ptr))))

(define-method (insert! (queue <queue>) (item <top>))
  (let ((new-pair (list item)))
    (cond ((empty? queue)
           (set-slot-value! queue 'front-ptr new-pair)
           (set-slot-value! queue 'rear-ptr new-pair))
          (else
           (set-cdr! (rear-ptr queue) new-pair)
           (set-slot-value! queue 'rear-ptr new-pair)))))

(define-method (delete! (queue <queue>))
  (if (empty? queue)
      (error "queue empty")
      (set-front-ptr! queue (cdr (front-ptr queue)))))
```

4.3.5 Verallgemeinerte Zuweisung

Beobachtung: In Common Lisp gibt es ein recht praktisches Konstrukt namens `setf` (set form). Hier kann man schreiben:

```
(setf (car pair) 1)  ; anstatt (set-car! pair 1)
(setf (cdr pair) 1)
(setf (list-ref list 3) 1)
(setf (slot-value rect 'width) 10)
...
```

Vorteile:

- Nicht so viele neue Symbole.
- Eventuell Wiederverwendung von Code wie `(cdr pair)`.

Implementation: Wir geben eine sehr rudimentäre Implementation, die leider (im Gegensatz zu dem echten Common Lisp `setf`) nicht einfach erweiterbar ist:

```
(define-syntax setf!
  (syntax-rules (car cdr slot-value ^)
    ((setf! (car x) y)
     (let ((z y))
       (set-car! x z)
       z))
    ((setf! (cdr x) y)
     (let ((z y))
       (set-cdr! x z)
       z))
    ((setf! (slot-value x slot) y)
     (let ((z y))
       (set-slot-value! x slot z)
       z))
    ((setf! (^ x slot) y)
     (let ((z y))
       (set-slot-value! x slot z)
       z))))
```

4.4 Tabellen

Definition: Tabellen sind Abbildungen einer Schlüsselmenge auf eine Wertemenge. Eine Tabelle heißt veränderbar, wenn man

- den Definitionsbereich erweitern kann,

- den Definitionsbereich verkleinern kann oder
- den Wert für einen gegebenen Schlüssel verändern kann.

Schnittstelle: Unsere Tabellen stellen folgende Funktionalität bereit:

- `(insert! key value)` – fügt die Zuordnung `key` → `value` in die Tabelle ein
- `(remove! key)` — entfernt eine Zuordnung `key` → `value` (wenn vorhanden)
- `(lookup key)` — gibt `value` zurück, wenn eine Zuordnung `key` → `value` existiert, ansonsten einen bestimmten `not-found`-Wert, der bei Tabellenerzeugung mitgeliefert wird.
- `(dic-for-each func)` — wendet die Funktion `func` auf alle Einträge der Tabelle an
- Auch der Gleichheitstest wird mitgeliefert

4.4.1 Assoziationslisten

Implementation: Hier ist eine Beispielimplementation einer Tabelle, die Assoziationslisten verwendet. Wir haben hier die OO-Strategie über Message-Passing gewählt (hauptsächlich, um einen Zirkel zu vermeiden, da wir ja für die generischen Funktionen schon Tabellen gebraucht haben).

```
(define (make-association-list-dictionary assoc not-found)
  (define my-table ())
  (define (lookup key)
    (cond ((assoc key my-table) => cdr)
          (else not-found)))
  (define (insert! key value)
    (let ((record (assoc key my-table)))
      (if record
          (set-cdr! record value)
          (set! my-table (cons (cons key value) my-table)))))
  (define (remove! key)
    (set! my-table (remove (assoc key my-table) my-table)))
  (define (dic-for-each proc)
    (for-each (lambda (item) (proc (car item) (cdr item)))
              my-table))
  (publish lookup insert! remove! dic-for-each))
```

Bemerkung: Den Message-Passing-Dispatch haben wir durch `publish` etwas abgekürzt:

```
(define-syntax publish
  (syntax-rules ()
    ((publish sym1 ...)
     (lambda (message)
       (case message
         ((sym1) sym1)
         ...)))))
```

Bemerkung: Man kann etwas Klammern sparen beim Gebrauch der Schnittstelle, wenn man definiert:

```
(define (lookup table key)
   ((table 'lookup) key))
(define (insert! table key value)
   ((table 'insert!) key value))
(define (remove! table key)
   ((table 'remove!) key))
(define (dic-for-each table proc)
   ((table 'dic-for-each) proc))
```

Dies ist ähnlich zu der Funktionalität von `define-generic` beim OO über generische Funktionen. Allerdings wird der Dispatch nur mit dem ersten Argument durchgeführt.

4.4.2 Vektoren

Syntax: Vektoren werden erzeugt durch

```
(vector element-1 ...)
(make-vector nr-of-elements initial-value)
```

Der Zugriff geschieht mit

```
(vector-ref vector i)
(vector-set! vector i value)
```

Bemerkung: Wir werden Vektoren wenig brauchen. Für viele Anwendungen sind sie allerdings sehr wichtig, z.B. bei der Implementation von schnellen Matrix-Vektor-Operationen. Um diese wirklich schnell zu machen, muss die Implementation allerdings uniforme Felder (alle Einträge haben denselben Typ) unterstützen, was viele Scheme-Implementationen nicht tun.

Bemerkung: Vom Standpunkt des Theoretikers stellt ein Vektor keine wesentliche Funktionalitätssteigerung gegenüber der Liste oder (noch besser) einem balanzierten Binärbaum dar. Der Zugriff auf einen als balanzierten Binärbaum implementierten Vektor hat den Aufwand $O(log_2 n)$, was sogar der theoretisch optimale Wert ist (Warum?).

Bemerkung: Wir können natürlich auch wieder unsere allgemeine Schnittstelle darübersetzen:

```
(define (make-vector-dictionary N not-found)
  (define store (make-vector N not-found))
  (define (lookup i)
    (if (<= 0 i (1- N))
        (vector-ref store i)
        not-found))
  (define (insert! i value)
    (if (<= 0 i (1- N))
        (vector-set! store i value)
        (error "out of range")))
  (define (remove! i)
    (when (<= 0 i (1- N))
          (vector-set! store i not-found)))
  (define (dic-for-each proc)
    (do ((i 0 (1+ i)))
        ((= i N))
      (unless (eq? (vector-ref store i) not-found)
              (proc i (vector-ref store i)))))
  (publish lookup insert! remove! dic-for-each))
```

4.4.3 Hash-Tabellen

Beobachtung: In der Praxis oft noch bessere Zugriffswerte als mit balanzierten Binärbäumen ist mit sogenannten Hash-Tabellen möglich.

Idee: Man nimmt an, dass es eine Hash-Funktion gibt, die die vorkommenden Schlüssel recht gleichmäßig auf eine Indexmenge $1, \ldots, N$ verteilt, wobei N in etwa der Größe der Tabelle entspricht. Dann speichert man ein `key`/`value`-Paar in einem Vektor der Länge N, der in jeder Komponente die Tabelle für diesen Hash-Code verwaltet (z.B. als Assoziationsliste).

Syntax: (MzScheme)

```
(make-hash-table)
(make-hash-table 'equal)
(hash-table-put! table key value)
(hash-table-get table key)
(hash-table-get table key not-found)
```

Bemerkung: Wieder kann man eine Message-Passing-Schnittstelle für Hash-Tabellen schreiben:

```
(define (make-hash-table-dictionary keys not-found)
  (define table (apply make-hash-table keys))
  (define (lookup key)
    (hash-table-get table key not-found))
  (define (insert! key value)
```

```
    (hash-table-put! table key value))
  (define (remove! key)
    (hash-table-remove! table key))
  (define (dic-for-each proc)
    (hash-table-for-each table proc))
  (publish lookup insert! remove! dic-for-each))
```

Bemerkung:

- Die Effizienz von Hash-Tabellen hängt entscheidend von der Güte der Hash-Funktion ab.
- Für viele gebräuchliche Hash-Funktionen ist nur der durchschnittliche Zugriff schnell, eine obere Schranke für den Zugriff ist nicht garantiert.

4.4.4 Memoisation/Memoization

Idee: Für Funktionen, die viel Rechenzeit benötigen, aber nur für relativ wenige Werte aufgerufen werden, kann es nützlich sein diese Werte zu tabellieren. Hiermit sind oft erhebliche Effizienzsteigerungen möglich.

Bemerkung: Es gibt viele verwandte Konzepte, so z.B. Caching bei Speicherhierarchien, Datenbank- oder Webzugriffen.

Implementation: Folgende Funktion wickelt eine Tabellierung um ihr Argument (eine Funktion mit langer Laufzeit) herum:

```
(define (memoize f)
  (let* ((not-found (list ()))
         (dic (make-hash-table-dictionary '(equal) not-found)))
    (lambda args
      (let ((value (lookup dic args)))
        (when (eq? value not-found)
              (set! value (apply f args))
              (insert! dic args value))
        value))))
```

Bemerkung: `when` und `unless` sind zwei aus Common Lisp übernommene Formen, die folgendermaßen implementiert wurden:

```
(define-syntax when
  (syntax-rules ()
    ((when condition body ...)
     (if condition (begin body ...)))))

(define-syntax unless
```

```
  (syntax-rules ()
    ((unless condition body ...)
     (if (not condition) (begin body ...)))))
```

Bemerkung: In vielen Fällen reicht nun folgendes Vorgehen aus:

```
(define memoized-f (memoize f))
;; ab hier verwende memoized-f
(memoized-f ...)
```

Dies versagt allerdings bei Funktionen, die aufgrund von Rekursion lange Laufzeit haben:

```
(define (fib n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (else (+ (fib (- n 1))
                 (fib (- n 2))))))

(time (fib 30))
(define memoized-fib (memoize fib))
(time (memoized-fib 30))
```

Hier ist es nötig, `fib` selbst zu modifizieren:

```
(define fib (memoize fib))
(time (fib 30))
(time (fib 10000))
```

Bemerkung: Interessant ist die minimale Invasivität, die uns Scheme hier erlaubt. Der Code der zu memoisierenden Funktion wird nicht verändert, man kann das Memoisieren später bei Bedarf aktivieren. In vielen anderen Sprachen muss man die Funktion dagegen schon von Anfang mit einer Memoisier-Struktur versehen.

4.5 Datenflussprogrammierung

4.5.1 Einführung

Idee: Das Ändern von Daten erzwingt die Änderung abhängiger Werte.

Verwendung:

- Tabellenkalkulation
- GUI-Programmierung
- Web-Anwendungen
- anderes

4.5.2 Beispiel: Schaltkreissimulation

Situation: Ein logischer Schaltkreis besteht aus

- Und-Gattern
- Oder-Gattern
- Invertierern
- Verbindungsleitungen

Jedes der Bauteile hat bestimmte Verzögerungszeiten, bis es den Ausgang auf veränderte Eingangssignale einstellt.

Aufgabe: Simulation von Schaltkreisen.

4.5.3 Halbaddierer

Definition: Ein Halbaddierer hat zwei Eingänge A und B, sowie zwei Ausgänge S (Summe) und C (Carry=Übertrag). S ergibt sich als exklusiv-oder von A und B, d.h.

$$S = A \dot{\vee} B = (A \vee B) \wedge \neg(A \wedge B)\,,$$

C erhält man als $C = A \wedge B$.

4.5.4 Volladdierer

Definition: Der Volladdierer ergibt sich durch Zusammensetzen zweier Halbaddierer. Er hat drei Eingänge A, B, und C_{in} und zwei Ausgänge S und C_{out}. Ein HA führt die Operation $S_1 = A + B$ mit Übertrag C_1 durch, ein anderer addiert dann $S = S_1 + C_{\text{in}}$ mit Übertrag C_2. C_{out} ergibt sich dann als $C_{\text{out}} = C_1 \vee C_2$.

4.5.5 Addierwerk (Ripple-Carry-Adder)

Definition: Ein Addierwerk entsteht durch Aneinaderreihen von Volladdierern, wobei C_{out} des ersten gleich mit C_{in} des zweiten ist, usw.

4.5.6 Implementation

Programm:

```
;;; -*- mode: scheme; fill-column: 64; -*-

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;; Digital circuit simulator
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(load "useful.scm")
(load "queues.scm")

;;; Concepts:

;;; The basic building blocks are wires.  Changes to the wire
;;; signal induce other actions to be carried out according to
;;; which gates are connected to the wire.

;;; wires
(define (make-wire)
  "Creates a wire.  A wire is in a certain state and carries out
a list of actions when its state changes."
  (let ((signal-value 0)
        (actions ()))
    (define (get-signal) signal-value)
    (define (set-signal! new-value)
      (unless (= signal-value new-value)
              (set! signal-value new-value)
              (for-each funcall actions)))
    (define (add-action! proc)
      (push proc actions)
      (proc)  ; !!
      )
    (publish get-signal set-signal! add-action!)))

(mp-generic (get-signal wire))
(mp-generic (set-signal! wire new-value))
(mp-generic (add-action! wire action))

;;;; Wire connectors

(define (logical-not s)
  (if (= s 0) 1 0))

(define (logical-and i1 i2)
  (if (= i1 0) 0 i2))

(define (logical-or i1 i2)
```

```
  (if (= i1 0) i2 i1))

(define (inverter input output)
  (define (invert-input)
    (let ((new-value (logical-not (get-signal input))))
      (after-delay inverter-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! input invert-input))

(define (and-gate i1 i2 output)
  (define (and-action-procedure)
    (let ((new-value
           (logical-and (get-signal i1) (get-signal i2))))
      (after-delay and-gate-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! i1 and-action-procedure)
  (add-action! i2 and-action-procedure))

(define (or-gate i1 i2 output)
  (define (or-action-procedure)
    (let ((new-value
           (logical-or (get-signal i1) (get-signal i2))))
      (after-delay or-gate-delay
                   (lambda ()
                     (set-signal! output new-value)))))
  (add-action! i1 or-action-procedure)
  (add-action! i2 or-action-procedure))

;;;; composites

(define (half-adder a b s c)
  (let ((d (make-wire))
        (e (make-wire)))
    (or-gate a b d)
    (and-gate a b c)
    (inverter c e)
    (and-gate d e s)))

(define (full-adder a b c-in sum c-out)
  (let ((s (make-wire))
        (c1 (make-wire))
        (c2 (make-wire)))
    (half-adder b c-in s c1)
    (half-adder a s sum c2)
```

```
    (or-gate c1 c2 c-out)))

;;;; The agenda

(define (make-time-segment time queue)
  (cons time queue))
(define (segment-time s) (car s))
(define (segment-queue s) (cdr s))

(define (make-agenda)
  (let ((current-time 0)
        (segments ()))
    (define (empty?) (null? segments))
    (publish (accessor current-time) (accessor segments) empty?)))

(mp-generic (current-time agenda))
(mp-generic (segments agenda))
(mp-generic (empty? agenda))

(define (add-to-agenda! time action agenda)
  (define (belongs-before? segments)
    (or (null? segments)
        (< time (segment-time (car segments)))))
  (define (make-new-time-segment time action)
    (let ((q (make-queue)))
      (insert-queue! q action)
      (make-time-segment time q)))
  (define (add-to-segments! segments)
    (if (= (segment-time (car segments)) time)
        (insert-queue! (segment-queue (car segments))
                       action)
        (let ((rest (cdr segments)))
          (if (belongs-before? rest)
              (set-cdr!
               segments
               (cons (make-new-time-segment time action)
                     (cdr segments)))
              (add-to-segments! rest)))))
  (let ((segments (segments agenda)))
    (if (belongs-before? segments)
        ((agenda '(set! segments))
         (cons (make-new-time-segment time action)
               segments))
        (add-to-segments! segments))))

(define (remove-first-agenda-item! agenda)
```

```
  (let ((q (segment-queue (first (segments agenda)))))
    (delete-queue! q)
    (if (empty-queue? q)
        ((agenda '(set! segments))
         (rest (segments agenda))))))

(define (first-agenda-item agenda)
  (if (empty? agenda)
      (error "Agenda is empty - FIRST-AGENDA-ITEM")
      (let ((first-seg (first (segments agenda))))
        ((agenda '(set! current-time)) (segment-time first-seg))
        (front (segment-queue first-seg)))))

;;; the global agenda

(define *the-agenda* (make-agenda))

(define (after-delay delay action)
  (add-to-agenda! (+ delay (current-time *the-agenda*))
                  action
                  *the-agenda*))

(define (probe name wire)
  (add-action! wire
               (lambda ()
                 (newline)
                 (display "Time = ")
                 (display (current-time *the-agenda*))
                 (display "  ID = ")
                 (display name)
                 (display "  New-value = ")
                 (display (get-signal wire))))
  (newline))


(define (propagate)
  (if (empty? *the-agenda*)
      (newline)
      (let ((first-item (first-agenda-item *the-agenda*)))
        (remove-first-agenda-item! *the-agenda*)
        (first-item)
        (propagate))))

;;;; Simple test: half-adder

(define inverter-delay 2)
```

```
(define and-gate-delay 3)
(define or-gate-delay 5)

(define input-1 (make-wire))
(define input-2 (make-wire))
(define sum (make-wire))
(define carry (make-wire))

;;(probe 'sum sum)

;;(probe 'carry carry)

;; (half-adder input-1 input-2 sum carry)

;; (set-signal! input-1 1)

;; (propagate)

;; (set-signal! input-2 1)

;; (propagate)

;;;; Ripple-carry-adder

(define (int n)
  (let ((vec (make-vector n)))
    (dotimes (i n vec)
       (vector-set! vec i (make-wire)))))

(define (ripple-carry-adder a b s)
  (unless (apply = (map vector-length (list a b s)))
          (error "inputs/output should have the same length"))
  (let* ((n (vector-length s))
         (carry (int (1+ n))))
    (dotimes (i n)
             (full-adder
              (vector-ref a i)
              (vector-ref b i)
              (vector-ref carry i)
              (vector-ref s i)
              (vector-ref carry (1+ i))))))

(define (show-bits int)
  (do ((i (1- (vector-length int)) (1- i)))
      ((negative? i) (newline))
    (display (get-signal (vector-ref int i)))))
```

```
(define (get-int int)
  (do ((i 0 (1+ i))
       (result 0 (+ result
                    (* (expt 2 i)
                       (get-signal (vector-ref int i))))))
      ((= i (vector-length int)) result)))

(define (probe-int name int)
  (dotimes (i (vector-length int))
           (probe (list name i) (vector-ref int i))))


(define (set-int! int k)
  (let loop ((i 0)
             (k k))
    (when (< i (vector-length int))
          (set-signal! (vector-ref int i) (remainder k 2))
          (loop (1+ i) (quotient k 2)))))


(define a (int 8))
(define b (int 8))
(define sum (int 8))

(probe-int 'sum sum)

(ripple-carry-adder a b sum)

;; (show-bits sum)
;; (show-bits a)
;; (show-bits b)
;; (set-int! a 77)
;; (set-int! b 155)
;; (get-int sum)
;; (propagate)
;; (get-int sum)
```

Bemerkung: Nicht wesentlich: Den `publish`-Befehl zur Unterstützung der Message-Passing-OO-Programmierung habe ich zum Export von Accessoren auf Datenfelder erweitert:

```
(publish (reader slot))
```

liefert einen Lesezugriff mittels `((object 'slot))`, ebenso erhält man mit

```
(publish (writer slot))
```

einen schreibenden Zugriff als `((object '(set! slot)) value)`.

4.5.7 Bemerkungen

- Etwas einfacher hätte man die Agenda als einfache Liste mit Elementen der Form `(time . action)` implementieren können, bei der man beim Einfügen auf die korrekte Ordnung achtet, insbesondere auch darauf, dass ein neues Paar nach allen Paaren mit der gleichen Zeit eingefügt wird. Der Nachteil wäre, dass dies bei großen Schaltkreisen ein $O(n^2)$-Verhalten aufweisen würde.
- Verwandt zur Datenflussprogrammierung ist die deklarative Programmierung. Der Hauptunterschied ist, dass die Propagation von Änderungen nicht nur in einer Richtung geschieht.
- Die Datenflussprogrammierung wird im Augenblick in den meisten Programmiersprachen nur durch Bibliotheken unterstützt. Insbesondere die Integration von funktionalen und datenflussgetriebenen (oder auch deklarativen) Programmteilen ist (bisher) nicht so natürlich wie etwa die Integration von objektorientierter und funktionaler Programmierung.

4.6 Gleichzeitige Prozesse

4.6.1 Einführung

Beobachtung: In der realen Welt geschehen viele Prozesse gleichzeitig. Es ist naheliegend, dass eine Nachbildung im dieses Verhaltens im Rechenmodell Vorteile bei der Behandlung praktisch relevanter Probleme bieten sollte.

Beispiele: (im Softwarebereich)

- Alle modernen Betriebssysteme und Benutzeroberflächen
- Verteilte Anwendungen, z.B. Web-Server, Telefonnetz, Internet, ...
- Datenbanken

Vorteile: Die Möglichkeit paralleler Prozesse erlaubt insbesondere:

- Modularere Programme
- Schnellere Abarbeitung, wenn mehrere Prozessoren verwendet werden.

Beispiel: Ausführung einer längeren Rechnung im Hintergrund:

```
(load "wechselgeld.scm")
(thread (lambda ()
          (display (moeglichkeiten 300 6))
          (newline)))
```

Syntax: In MzScheme gibt es den Befehl `thread`, der eine argumentlose Prozedur (einen sogenannten `thunk`="already thought about") als parallelen Prozess ausführt. Man kann dies etwas erweitern durch

```
;;; mehrfaches thread wie in SICP
(define (parallel-execute . thunks)
  (for-each thread thunks))
```

Damit kann man dann mehrere Prozeduren parallel ausführen.

Aber: In den interessantesten Fällen sind die einzelnen Prozesse nicht fein säuberlich getrennt, sondern stehen miteinander in Wechselwirkung durch

- Gemeinsam genutzte Daten
- Gemeinsam genutzte andere Ressourcen
- Austausch von Botschaften

Dies führt zu einem erheblichen Komplexitätszuwachs.

4.6.2 Probleme

Beispiel: Verschiedene Prozesse greifen auf dasselbe Konto zu, welches wir als globale Variable `mein-konto` modellieren. Anfangs ist die Bilanz 100 Euro, dann hebt ein Prozess 10 Euro ab, und ein anderer Prozess hebt 25 Euro ab.

```
(define mein-konto 100)
(begin
  (set! mein-konto 100)
  (parallel-execute
   (lambda ()
     (set! mein-konto (- mein-konto 10)))
   (lambda ()
     (set! mein-konto (- mein-konto 25))))
  (sleep 0.1)
  mein-konto)
```

Das scheint alles zu funktionieren. Für komplizierte Anwendungen realistischer ist allerdings folgende Variante.

```
(define (mein-konto-with-delays delay1 delay2)
  (sleep delay1)
  (let ((value mein-konto))
    (sleep delay2)
    value))

(begin
```

```
  (set! mein-konto 100)
  (parallel-execute
   (lambda ()
     (set! mein-konto
           (- (mein-konto-with-delays 0.0 0.0) 10)))
   (lambda ()
     (set! mein-konto
           (- (mein-konto-with-delays 0.1 0.1) 25))))
  (sleep 1.0)
  mein-konto)
```

Wir beobachten, dass hier, je nach den Verzögerungswerten, die Werte 90, 75, 65 am Ende auftreten können.

4.6.3 Serialisierer

Definition: Ein Serialisierer sorgt dafür, dass nur jeweils ein Prozess gleichzeitig auf eine Ressource zugreifen darf.

Syntax: Ein Serialisierer ist eine Funktion, die `thunk`-Prozeduren so abändert, dass sie hintereinander gestartet werden. Die Anwendung soll wie folgt sein:

```
(let ((s (make-serializer)))
  (set! mein-konto 100)
  (parallel-execute
   (s (lambda ()
        (set! mein-konto
              (- (mein-konto-with-delays 0.0 0.0) 10))))
   (s (lambda ()
        (set! mein-konto
              (- (mein-konto-with-delays 0.1 0.1) 25)))))
  (sleep 1.0)
  mein-konto)
```

Implementation: Wir verwenden die Semaphore, die MzScheme bereitstellt. Damit könnte man `make-serializer` wie folgt implementieren:

```
(define (make-serializer)
  (let ((sema (make-semaphore 1)))
    (lambda (proc)
      (lambda args
        (semaphore-wait sema)
        (let ((value (apply proc args)))
          (semaphore-post sema)
          value))))))
```

Bemerkung: Tatsächlich sind die Semaphoren ein flexiblerer (aber nicht ganz so einfach zu bedienender) Mechanismus zur Steuerung paralleler Prozesse. Zum Beispiel kann man eine Ressource durch (`make-semaphore n`) für n Prozesse zulassen, oder mit `semaphore-try-wait?` prüfen, ob eine Ressource verfügbar ist.

Bemerkung: Bei einer guten Implementation verschwendet ein Prozess beim Warten auf ein Semaphor oder eine Ressource keine Rechenzeit.

Programm: Natürlich kann man das ganze auch im objektorientierten Kontext absichern:

```
(define (konto balance)
  (define (withdraw amount)
    (set! balance (- balance amount)))
  (define (deposit amount)
    (set! balance (+ balance amount)))
  (let ((protect (make-serializer)))
    (publish (alias withdraw (protect withdraw))
             (alias deposit (protect deposit))
             (alias show (protect (lambda () balance))))))

(let ((mein-konto (konto 100)))
  (parallel-execute
   (lambda ()
     ((mein-konto 'withdraw) 10))
   (lambda ()
     ((mein-konto 'withdraw) 30)))
  (sleep 0.1)
  ((mein-konto 'show)))
```

Aber: Dennoch bleiben Probleme:

- Was geschieht, wenn man den Inhalt zweier Kontos austauschen will? Hier muss der Gesamtprozess serialisiert werden, wozu man die Serialisierer von beiden Konten benötigt.
- Wenn die Serialisierer von beiden Konten zugänglich sind: was geschieht, wenn Person A und Person B die Konten 1 und 2 austauschen wollen? Hier kann folgendes geschehen: A reserviert sich zuerst den Serialisierer 1, und kann dann Serialisierer 2 nicht reservieren, weil er bereits von B reserviert wurde. Nun wartet A auf S2 und B auf S1. Dieses Problem nennt man `Deadlock`, und es ist (vor allem in komplizierteren Situationen) nicht einfach zu vermeiden.
- Für verteilte Systeme ohne schnellen Zugriff auf eine gemeinsame Ressource ist die Koordination nochmals schwieriger (z.B. Geldautomaten). Hier ist eigentlich nicht einmal klar, was der aktuelle Kontostand (z.B. Kontostand) ist, was etwas an das Gleichzeitigkeitsproblem der Relativitätstheorie erinnert, was ja auch durch die endliche Lichtgeschwindigkeit hervorgerufen wird.

Bemerkung: Es gibt Sprachen, die besonders auf das Programmieren mit parallelen Prozessen ausgerichtet sind. Ein Beispiel ist Erlang, was im Rahmen der Telekommunikationsbranche von der Firma Ericsson entwickelt wurde (es ist aber mittlerweile für die Allgemeinheit freigegeben). Die Stärke von Erlang ist insbesondere, dass es sehr viele und sehr leichtgewichtige Prozesse handhaben kann, die dennoch gegeneinander gut abgeschirmt sind und über eine Art Mailboxen miteinander kommunizieren.

4.7 Datenströme und verzögerte Auswertung

Idee: Statt eines veränderbaren Objekts $x = x(t)$ oder $x : k \mapsto x_k$ betrachtet man die ganze Funktion $x : t \mapsto x(t)$ bzw. Folge $(x_k)_{k\in\mathbb{N}}$.

4.7.1 Verzögerte Listen

Erinnerung: Wir hatten Listen als Schnittstellen propagiert, z.B.

```
(sum-primes a b) =
(reduce + (filter prime? (range a b)))
```

Problem: Nicht immer effizient genug, z.B. wenn die Aufgabe lautet, die zweite Primzahl im Bereich $10^5, \ldots, 10^6$ zu finden, ist es nicht sinnvoll, alle Zahlen in diesem Bereich zu sammeln und auf Primalität zu testen.

Abhilfe: Wir konstruieren ein spezielles Paar, bei dem der `cdr`-Teil verzögert ausgewertet wird.

Implementation:

```
;;; Stream-Konstruktor
(define-syntax s-cons ; muss Sonderform sein!
  (syntax-rules ()
    ((s-cons x y)
     (cons x (delay y)))))

;;; Selektoren
(define s-car car)
(define (s-cdr s) (force (cdr s)))

;;; Null-Test
(define s-null? null?)
```

Bemerkung: `delay` und `force` sind in R5RS (Revised^5 Report on Scheme) bereits enthalten. Man könnte sie aber auch sehr einfach implementieren:

```
(define-syntax delay
  (syntax-rules ()
     ((delay expr)
      (lambda () expr))))

(define (force thunk)
  (thunk))
```

Bemerkung: Die implementierten Operatoren `delay` und `force` haben allerdings einen Vorteil, den man erst nach genauerer Untersuchung feststellt. Sie memoisieren:

```
(define (fib n)
  (if (< n 2)
      n
      (+ (fib (- n 1))
         (fib (- n 2)))))

(define x (delay (fib 33)))
(force x)  ; langsam
(force x)  ; schnell
```

Das ist sinnvoll, weil in Anwendungen Werte oft mehrfach verlangt werden. Natürlich könnten wir das auch sehr einfach in unserer Implementation einbauen:

```
(define-syntax delay
  (syntax-rules ()
     ((delay expr)
      (memoize (lambda () expr)))))
```

Implementation: Mit den Operationen `s-cons`, `s-car`, `s-cdr` und `s-null?` kann man jetzt den Listenoperationen entsprechende Funktionen definieren: `s-ref`, `s-map`, `s-for-each`, `s-range`, ...

Bemerkungen:

- Mit Hilfe generischer Funktionen hätte man anstelle von `s-car`, `s-cdr`, `s-map`, ... auch die üblichen Namen verwenden können. `s-cons` ist allerdings beizubehalten.
- Es gibt Lisp/Scheme-Varianten, bei denen sich Listen generell wie Datenströme verhalten, oder die sogar alle Auswertungen verzögern. Eine solche Variante wird zum Beispiel im Kapitel 4 des SICP-Buchs vorgestellt.

4.7.2 Unendliche Datenströme

Beispiele:

- Natürliche Zahlen:

```
(define (naturals>= n)
  (s-cons n (naturals>= (+ n 1))))

(define naturals (naturals>= 0))

(display naturals)
(s-ref naturals 10000)
```

- Fibonacci-Zahlen

```
(define fibs
  (letrec ((fibgen  ; falls man fibgen nicht oeffentlich haben will
            (lambda (a b)
              (s-cons a (fibgen b (+ a b))))))
    (fibgen 0 1)))

(s-ref fibs 100)
```

- Primzahlen mittels des Siebs von Eratosthenes

```
(define (divides? k n) (zero? (remainder n k)))

(define (sieve s)
  (s-cons (s-first s)
          (sieve (s-remove-if (curry divides? (s-first s))
                              (s-rest s)))))

(define primes (sieve (s-cdr (s-cdr naturals))))
```

Programm: (Ausgabe von Datenströmen) Folgende Routine gibt höchstens n Elemente eines Datenstroms aus:

```
(define (s-display s n)
  (do ((s s (s-cdr s))
       (k n (- k 1)))
      ((or (s-null? s) (zero? k)))
    (display (s-car s)) (newline)))
```

4.7.3 Implizite Definition von Datenströmen

Idee: Wir definieren Datenströme nicht über generierende Funktionen sondern drücken sie rekursiv aus.

Beispiele:

```
(define ones (s-cons 1 ones))
(define naturals
  (s-cons 0 (s-add ones naturals)))
(define fibs
  (s-cons 0 (s-cons 1 (s-add fibs (s-rest fibs)))))
(define powers-of-two
  (s-cons 1 (s-scale 2 powers-of-two)))
(define (partial-sums s)
  (define p-sums
    (s-cons (s-first s)
            (s-add (s-rest s) p-sums)))
  p-sums)
(s-ref (partial-sums naturals) 1000)
```

wobei wir verwendet haben

```
(define (s-add s1 s2)
   (s-map + s1 s2))
(define (s-scale factor stream)
  (s-map (curry * factor) stream))
```

4.7.4 Iterationen als Datenströme

Idee: Wir stellen eine reelle Zahl als approximierende Folge dar.

Beispiel: Es gilt

$$\pi = 4(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \pm \ldots)$$

Dies könnte man so erreichen:

```
(define alternating
  (s-cons 1 (s-scale -1 alternating)))
(define odds
  (s-filter odd? naturals))
(define (s-mul s1 s2)
  (s-map * s1 s2))
(define pi-summands
  (s-mul (s-map / odds) alternating))
(define pi-sequence
  (s-scale 4 (partial-sums pi-summands)))
(s-display pi-sequence 50)
```

Bemerkung: Die Folge konvergiert sehr langsam gegen π. Es gibt aber eine Technik zur Konvergenzbeschleunigung gewisser Folgen, die sogenannte Euler-Transformation:

```
;;; This is also known as Aitken's method (see Stoer/Bulirsch).
(define (euler-transform s)
  (let ((s0 (s-ref s 0))
        (s1 (s-ref s 1))
        (s2 (s-ref s 2)))
    (s-cons (- s2 (/ (square (- s2 s1))
                     (+ s0 (* -2 s1) s2)))
            (euler-transform (s-cdr s)))))

(s-display (euler-transform pi-sequence) 50)

;;; and an even faster acceleration by transforming over and over again...
(define (super-accelerate transform s)
  (define (make-tableau s)
    (s-cons s (make-tableau (transform s))))
  (s-map s-car (make-tableau s)))

(s-display (super-accelerate euler-transform pi-sequence) 10)
```

4.7.5 Ströme von Paaren

Problem: Gegeben seien zwei Datenströme S und T. Wir wollen einen Datenstrom konstruieren, der alle möglichen Paare aufzählt.

Idee: Das erste Element des Datenstroms ist das Paar (S_0, T_0). Der Rest entsteht durch Mischen der Datenströme $(S_0, T_i)_{i\geq 1}$, $(S_i, T_0)_{i\geq 1}$ und $(S_i, T_j)_{i,j\geq 1}$.

Implementation:

```
(define (s-merge s t)
  (if (s-null? s)
      t
      (s-cons (s-car s)
              (s-merge t (s-cdr s)))))

(define (pairs s t)
  (s-cons (cons (s-car s) (s-car t))
          (s-merge
           (s-merge (s-map (lambda (x) (cons x (s-car t)))
                           (s-rest s))
                    (s-map (lambda (x) (cons (s-car s) x))
                           (s-rest t)))
           (pairs (s-cdr s) (s-cdr t)))))

(define NxN (pairs naturals naturals))
```

4.7.6 Bemerkungen

- Datenströme stellen eine interessante Möglichkeit zur funktionalen Modellierung von Zustandsänderungen dar. Beispiel: Modelliere die Interaktion eines Kunden mit einem Bankkonto als Abbildung des Datenstroms der Aufträge in einen Datenstrom der Kontobilanz.

- Leider treten aber auch hier Schwierigkeiten auf. Zum Beispiel muss man beim Zugriff von zwei Personen auf ein gemeinsames Konto zwei Datenströme von Aufträgen mischen, wobei wieder die reale Zeit ins Spiel kommen muss.

Index