2 Funktionsorientierte Programmierung

2.1 Charakteristik

2.1.1 Funktionale und funktionsorientierte Sprachen

Funktionale und *funktionsorientierte* Programmiersprachen gehören (zusammen mit den logikbasierten) zur Gruppe der *deskriptiver* Sprachen. Rein funktionale Sprachen besitzen keinerlei Wertzuweisung, womit beispielsweise bestimmte Zustände (von Daten und Berechnungen) repräsentiert werden könnten. Dies macht Programmierung frei von *Nebenwirkungen* (*Seiteneffekten*) und ermöglicht die Anwendung mathematischer Methoden.

Von besonderer Bedeutung ist die *Abstraktion*: Variablen-, Daten- und prozedurale Abstraktion, s. Kap. 1. Das hohe deskriptive Niveau der funktionalen Programmierung wird nicht zuletzt durch leistungsfähige Programmierkonzepte, wie Rekursion für Daten und Programme, Funktionen höherer Ordnung, und alternative *Evaluationstechniken*[1] bestimmt. Diese Themen werden in den folgenden Abschnitten vertieft.

Die wichtigsten theoretischen Grundlagen für die funktionale Programmierung wurden 1941 von Alonzo Church (1903-1995) in Gestalt des sog. λ-*Kalküls* gelegt. Damals bestand die Motivation nicht darin, einen neuen Sprachtyp zu fundieren, sondern theoretisch zu klären, welche Berechnungsaufgaben von Computern gelöst werden können und welche nicht. Dieses Ziel erreichte man mit dem λ-Kalkül, wobei es hierfür gleichwertige Alternativen gibt. Darüber hinaus wurde damit die Grundlage für Lisp[2] – neben Fortran eine der ältesten Programmiersprachen überhaupt – geschaffen.

[1] Der Begriff „Evaluation“ wird hier sehr oft verwendet und bedeutet soviel wie Auswertung oder Bewertung.

[2] 1964 hat Landin das *closure*-Konzept (closure = an expression, frozen together with its environment, for later evaluation if and when needed. Im Zusammenhang mit der Diskussion um den Namensaufruf in Algol 60 wurde dafür 1960 der Begriff „thunk“ („thunk about“ als eine abstruse Vergangenheitsform für „think about“) geprägt; nicht etwa „a dull hollow sound – the basketball made a thunk as it hit the rim“.) eingeführt und eine Interpretation des λ-Kalküls auf einer abstrakten Maschine vorgestellt. 1976 folgten Beiträge von Henderson und Morris und unabhängig davon von Friedman und Wise zur *verzögerten* Evaluation (call by need), einer alternativen Evaluationstechnik für Ausdrücke, die 1992 mit Haskell pure Gestalt annahm.

© Springer Fachmedien Wiesbaden 2016
C. Wagenknecht, *Programmierparadigmen*,
DOI 10.1007/978-3-658-14134-9_2

Typische Vertreter funktionaler bzw. funktionsorientierter Programmiersprachen sind Lisp, Scheme, ML, Haskell, Gofer, Miranda, FP und Backus.

Racket ist eine *funktionsorientierte* Sprache. Neben streng funktionalen Eigenschaften stellt sie einige Zusätze bereit, die es uns gestatten, auch andere Paradigmen mit Racket zu studieren. Eine Einführung in die Arbeit mit Racket wurde in Kap. 1 gegeben.

2.1.2 Seiteneffektfrei, zustandslos und zeitunabhängig

Eine Haupteigenschaft des Paradigmas der funktionsorientierten Programmierung ist die *Seiteneffektfreiheit* (engl. side effect). Dies bedeutet, dass wir kein Zustandsmodell, sondern ein *Datenflussmodell* betrachten. Anschaulich darf man dabei an unsere Trichterbilder aus Kap. 1 denken: Dabei werden gewisse Daten oben in die dafür vorgesehenen Trichter hinein gegeben, im Inneren entsprechend weitergeleitet und schließlich ausgegeben. Idealerweise geschieht dies alles ohne Wertveränderungen von (zustandsverwaltenden) Variablen.

Dies hat zur Folge, dass wiederholt ausgeführten Evaluationen ein und desselben symbolischen Ausdrucks stets den gleichen Wert beschreiben und z. B. mit Racket auch liefern. Während der Evaluation wird also weder die eigene Umgebung, in der diese Evaluation stattfindet, modifiziert, noch die einer anderen Funktion (Prozedur). Das Ergebnis hängt ausschließlich von den Argumenten (aktuellen Parametern) ab, die beim Prozeduraufruf angegeben wurden. Dies entspricht dem mathematischen Begriff der Funktion als Abbildung.

Obwohl absolute Seiteneffektfreiheit in rein funktionalen Sprachen durchaus anzutreffen ist, wird dies in funktionsorientierten Programmiersprachen nicht mit ausnahmsloser Strenge umgesetzt. Nehmen wir beispielsweise einen Generator für Zufallszahlen, so wie er mit `random` in Racket zur Verfügung steht.

```
> (random 10)
6
> (random 10)
2
```

Bei jedem Aufruf von `(random 10)` wird eine natürliche Zahl aus $[0, 9]$ zurückgegeben. Man sagt, die Wahl dieses Rückgabewertes trägt zufälligen Charakter. In Wirklichkeit handelt es sich aber gar nicht um echten Zufall, sondern um den Einsatz eines Verfahrens, das auf die jeweils zuletzt erzeugte „Zufallszahl" z_i angewandt wird, um die jeweils folgende z_{i+1} zu generieren. Für den dann folgenden Schritt muss z_{i+1} in der entsprechenden Umgebung gespeichert werden, ebenso wie vorher z_i. Es findet also eine Nebenwirkung, ein Seiteneffekt, statt.

Seiteneffekte, wie beispielsweise bei `random`, entstehen also dadurch, dass man Rechenobjekte konstruiert, die über einen Satz lokaler Variablen verfügen. Die Werte dieser Variablen charakterisieren den jeweils aktuellen *Zustand* dieser Objekte. Dieses Vorgehen

wird in der *objektorientierten Programmierung* zu einem tragenden Prinzip ausgebaut. Es folgt dem *Modell der Zustandsmaschine*, wie es in der *imperativen Programmierung* fest verankert ist.

Demgegenüber sind Funktionen *zustandslos*. Sie verwenden keine Zustandsvariablen und benötigen folglich auch keine Wertzuweisung – ein Markenzeichen imperativer Sprachen. Um es ganz deutlich zu sagen: Eine funktionsorientierte Programmiersprache kommt ganz ohne Wertzuweisung aus. Dies mag insbesondere mit dem Blick auf rekursive Prozeduren einige Zweifel hervorrufen. Es ist aber so und kann auch nachgewiesen werden.

In Racket gibt es Sprachelemente für Wertzuweisungen (Initialisierung und Mutation): `define` und `set!` Die damit verbundene Bequemlichkeit beim Umgang mit (lokalen und globalen) Variablen muss durch besondere Sorgfalt bei deren Verwaltung bezahlt werden.

Aus der Zustandslosigkeit folgt die Zeitunabhängigkeit: In einem System, das sich mit dem *Datenflussmodell*[3] beschreiben lässt, spielt es keine Rolle, zu welchem Zeitpunkt die entsprechende Funktionsanwendung, d. h. der zugehörige Berechnungsprozess, stattfindet. Natürlich müssen die dafür erforderlichen Argumente vorliegen. Diese Eigenschaft macht funktionsorientierte Sprachen attraktiv für Systeme mit zeitgleichen Prozessen. Schaut man etwas genauer hin, ist eine vollständige Zeitunabhängigkeit nicht in allen Anwendungen zu erreichen. Insofern haben auch funktionsorientierte Sprachen mit ihren für die mathematische Analyse von Programmen vorzüglichen Eigenschaften ihre Grenzen.

2.1.3 Erweiterbarkeit

Eine Funktionsdefinition erfordert einen konstruktiven Prozess, wie man das von einem Baukasten her kennt. Dabei werden aufbauend auf *Basisbausteinen* (eingebaute Funktionen) und bereits *vorgefertigten Bauelementen* (nutzerdefinierte Funktionen) vorhandene *Konstruktionswerkzeuge* eingesetzt. Funktionsorientierte Sprachen sind also *erweiterbare* Sprachen: Vorhandenen Funktionen können neue hinzugefügt werden. Bei der Benutzung neuer Funktionen gibt es keinen Unterschied zu den eingebauten.

Die Details dieses „Baukastensystems“ werden im λ-Kalkül geregelt, s. Abschn. 2.5. Wie dies in Racket geschieht, haben wir in Kap. 1 bereits kennen gelernt. Grundsätzlich ist es deshalb möglich, Sprachen für spezielle Anwendungsbereiche, sog. *domain specific languages* (DSL), mit Racket zu entwickeln.

2.1.4 Funktions- und Prozedurbegriff

Um die in Racket implementierten Funktionen von den mathematischen begrifflich abzugrenzen, sprechen wir von Racket-*Prozeduren*. Prozeduren sind also Implementationen

[3] Komposition von Trichterbildern: Daten strömen hindurch.

zugehöriger Funktionen. Es ist klar, dass eine Funktion durch mehrere (sogar abzählbar unendlich viele) bedeutungsgleiche Prozeduren implementiert werden kann: Man braucht ja nur so etwas wie `(void 'Hallo)` oder einen Kommentar an einer (unkritischen) Stelle der Prozedurdefinition hinzu zu fügen, um eine weitere Prozedur für die betrachtete Funktion zu erhalten.

Ist jede (mathematisch definierte) Funktion implementierbar? Die wohl etwas überraschende Antwort lautet: Nein. Dies gilt sogar unabhängig von der verwendeten Implementationssprache. Wir geben ein Beispiel für eine nicht implementierbare Funktion h an: h sei eine einstellige Funktion über der Menge der nullstelligen[4] Racket-Prozeduren und wie folgt definiert:

$$h(\texttt{proc}) = \begin{cases} w, & \text{wenn } (\texttt{proc}) \text{ stoppt.} \\ f, & \text{sonst.} \end{cases}$$

`(proc)` ist der Aufruf der nullstelligen Prozedur `proc`. `xxxx` und `eternity` sind nullstellige Racket-Prozeduren. Testen Sie sie.

```
(define xxxx
  (lambda ()
    'hallo))

> (xxxx)
hallo
```

```
(define eternity
  (lambda ()
    (eternity)))

> (eternity)
...  stoppt nicht (Stop-Knopf drücken)
```

`eternity` implementiert die nirgendwo definierte Funktion, die keine Argumente nimmt. (Dies gelingt auch mit höherstelligen Funktionen, die für keines der möglichen Argumente definiert sind. Solche Funktionen benötigen wir hier jedoch nicht und beschränken uns auf `eternity`.)

Wir fragen nun nach einer Prozedur `haelt?` zur Implementation von h und erwarten, dass `(haelt? xxxx)` den Wert `#t` und `(haelt? eternity)` den Wert `#f` liefern.

Im Folgenden wird gezeigt, dass es die (Racket-)Prozedur `haelt?` zur Implementation von h nicht gibt. Dabei verwenden wir die Methode des *indirekten Beweises*. Wir nehmen an, es gäbe eine solche Prozedur `haelt?` für h.

Dann gibt es auch die Prozedur `crazy`, denn `crazy` wird wie folgt ausschließlich aus eingebauten Racket-Sprachelementen, `eternity` und `haelt?` konstruiert.

```
(define crazy
  (lambda ()
    (if (haelt? crazy)
        (eternity)
        'done)))
```

[4] s. Abschn. 2.1.5

Damit ist die Bedeutung von `crazy` klar: Falls die nullstellige Prozedur `crazy` anhält, d.h. `(haelt? crazy)` den Wert `#t` liefert, stoppt `crazy` nicht, denn es wird `(eternity)` ausgeführt. Dies ist jedoch unmöglich: Eine Prozedur kann nicht gleichzeitig anhalten und ewig weiterlaufen.

Versuchen wir es mit der alternativen Annahme, wonach `crazy` nicht stoppt. In diesem Falle hält `crazy` an und gibt `done` aus. Auch dies ist ein Widerspruch, woraus wir beweistechnisch auf die Richtigkeit der negierten Annahme schließen.

Somit haben wir gezeigt, dass es die Prozedur `haelt?` zur Berechnung von h nicht gibt (und niemals geben kann).[5] Mit anderen Worten: Es gibt mindestens eine mathematisch korrekt definierte Funktion, für deren Berechnung es keine (Racket-)Prozedur gibt. Deshalb sprechen wir in der Racket-Welt von Prozeduren anstatt von Funktionen.

In der Berechenbarkeitstheorie spricht man von der Unlösbarkeit (Unentscheidbarkeit) des *Halteproblems*.

2.1.5 Verallgemeinerungen des Funktionsbegriffes

Leistungsfähige funktionsorientierte Programmiersprachen, wie etwa Racket, verwenden einen *verallgemeinerten Funktionsbegriff*, der sich in folgenden Prozedurarten niederschlägt:

- n-stellige Prozeduren, mit $n > 0$. Beispiele dafür wurden in Kap. 1 betrachtet.
- *nullstellige* Prozeduren, wie z. B. `xxxx` und `eternity` aus Abschn. 2.1.4.
- *variabelstellige* Prozeduren, wie z. B. `+`: `(+ 1 2)`, `(+ 1 2 3 4)`.
- *Prozeduren höherer Ordnung*, die Prozeduren als Argumente nehmen und/oder Prozeduren als Werte von Prozeduren zurückgeben, s. Abschn. 2.2.2.
- Prozeduren, die in einem *mehrwertigen Kontext* (`multiple-value-context`) mehr als einen Rückgabewert liefern. Hinweis: Funktionen mit mehr als einem Funktionswert gehen über den Funktionsbegriff aus der Mathematik hinaus. Dort werden sie mit vektorwertigen Funktionen nachgebildet.

Variabelstellige Prozeduren gibt es nicht nur in der Racket-eingebauten Form. Sie können vom Nutzer ebenso definiert werden wie andere Prozeduren. Das folgende Beispiel zeigt eine mindestens zweistellige Prozedur `vproc`. Dies wird durch die Anzahl der formalen Parameter vor dem Punkt bestimmt.

```
(define vproc
  (lambda (x y . z)
    (printf "x: ~a, y: ~a, z: ~a~n" x y z)))
```

[5] Dieser Nachsatz betont, dass es bisher nicht etwa an dem genialen Geist mangelte, eine Prozedur für h anzugeben, sondern, dass es absolut unmöglich ist, eine solche zu finden.

Beim Aufruf von `vproc` müssen also mindestens zwei Argumente übergeben werden.

```
> (vproc 1 2 3 4 5)
x: 1, y: 2, z: (3 4 5)
```

Die ersten beiden Argumente, 1 und 2, werden an `x` bzw. `y` vermittelt. Die (grundsätzlich beliebig vielen) restlichen Argumente werden in einer Liste zusammengefasst. Diese Liste wird beim Aufruf an `z` gebunden.

Natürlich ist es auch möglich, eine beliebigstellige Prozedur zu definieren:

```
(define zproc
  (lambda x
    (printf "x: ~a~n" x)))
```

Die aktuellen Parameter des Aufrufausdrucks (falls vorhanden) werden als Liste an `x` gebunden.

```
> (zproc 1)
x: (1)
> (zproc)
x: ()
```

Um die Stelligkeit (arity) einer Prozedur festzustellen, stehen entsprechende Sprachelemente bereit:

```
> (procedure-arity vproc)
(arity-at-least 2)
> (procedure-arity zproc)
(arity-at-least 0)
> (procedure-arity sqrt)
1
```

Zur Illustration der Arbeit mit mehrwertigen Prozeduren, betrachten wir die folgenden (nicht gerade sinnvolle) Prozedur `eins2drei`. Sie nimmt ein Argument und gibt drei zurück. Zur Rückgabe verwendet sie das Sprachelement `values`.

```
(define eins2drei
  (lambda (x)
    (values x (* 2 x) 'hallo)))

> (eins2drei 7)
7
14
'hallo
```

Die vom *Generator* `eins2drei` erzeugten drei Resultatwerte werden hier einfach untereinander geschriebenen, im Allgmeinen jedoch von einer anderen Prozedur, die wir

Empfänger nennen, aufgenommen und weiterverarbeitet. Der Generator verpackt den erforderlichen Prozeduraufruf in eine nullstellige Prozedur.

```
> (call-with-values
     (lambda () (eins2drei 7))
     (lambda (x y z) (list x y z)))
'(7 14 hallo)
```

Racket stellt eine Prozedur `split-at` bereit, die eine gegebene Liste an einer wählbaren Position in zwei Teillisten zerlegt und diese als zwei Resultate zurückgibt.

```
> (split-at '(0 1 2 3 4 5 6) 3)
'(0 1 2)
'(3 4 5 6)

> (let ((ls '(1 9 22 3 63 44 83 23 4 94 27 62 73 12 13)))
    (split-at ls (quotient (length ls) 2)))
'(1 9 22 3 63 44 83)
'(23 4 94 27 62 73 12 13)
```

Die im Folgenden angegebene Prozedur `mergsort` repräsentiert ein sehr bekanntes Sortierverfahren, das wir im Zweiwertkontext implementieren.

```
(define mergesort
  (lambda (ls)
    (let ((middle (quotient (length ls) 2)))
      (if (= middle 0)
          ls
          (call-with-values
           (lambda () (split-at ls middle))
           (lambda (lls rls)
             (merge (mergesort lls) (mergesort rls))))))))
```

Die Idee von Mergesort besteht darin, die zu sortierende Liste etwa hälftig zu zerteilen, die Teillisten anschließend rekursiv zu sortieren und die nunmehr sportierten Teillisten zusammenzuführen (Prozedur `merge`).

```
(define merge          ; für zwei SORTIERTE Listen
  (lambda (ls1 ls2)
    (cond
      ((null? ls1) ls2)
      ((null? ls2) ls1)
      ((< (car ls1)(car ls2))
       (cons (car ls1) (merge (cdr ls1) ls2)))
      (else
       (cons (car ls2) (merge ls1 (cdr ls2)))))))
```

```
> (mergesort (shuffle (range 1000)))
'(0 1 2 ... 999)
```

Aufgabe 2.1:
Machen Sie sich in der Literatur mit dem Sortierverfahren Quicksort vertraut, um die folgende Prozedur zu verstehen. Sie verwendet eine zweiwertige Funktion.
`call-with-values` nimmt einen Generator und einen Empfänger. `partition` zerlegt die Liste (ohne erstes Element) in zwei Teillisten gemäß Prädikat und gibt diese als (die beiden Werte) der Generatorfunktion aus. Der Empfänger nimmt sie als aktuelle Parameter `lls` und `rls`. Wenn das nicht kognitive Effizienz ist!

```
(define quicksort
  (lambda (ls)
    (if (empty? ls)
        ls
        (let ((pivot (car ls)))
          (call-with-values
           (lambda () (partition (lambda (n)(< n pivot)) (cdr ls)))
           (lambda (lls rls)
             (append
              (quicksort lls)
              (list pivot)
              (quicksort rls)))))))))

; > (quicksort (shuffle (range 100)))
; (0 1 2 3 ... 98 99)
```

2.1.6 Arbeitsstil bei funktionsorientierter Programmierung

Der beim funktionsorientierten Programmieren vorherrschende Arbeitsstil ist das *Beschreiben des Wertes*, der sich als Resultat der Evaluation des beschreibenden Ausdrucks ergibt. In Kap. 1 haben wir dies immer wieder betont. Dabei wird die Abstraktion von Berechnungsprozessen insbesondere durch *rekursive Beschreibungsformen* maßgeblich befördert.

Kurz und knapp kann man vereinfachend sagen: „Beschreibung *rekursiver* Daten mit *rekursiven* Prozeduren.“

Dies wollen wir an folgendem Extrembeispiel nochmals verdeutlichen.

Im Wettbewerb um das ineffizienteste Sortierverfahren behandeln wir nun einen Anwärter auf den ersten Platz. Das ist kein Problem, denn hier geht es nicht um Effizienz, sondern um eine Demonstration dessen, was mittels deskriptiver Programmierung möglich ist. Die Rekursion spielt auch in diesem Beispiel eine ganz besondere Rolle: Wir entwickeln

eine Prozedur mit fünf rekursiven Aufrufen.

Zu sortieren ist eine beliebig lange Liste paarweise verschiedener, natürlicher Zahlen.

```
(define usls '(1 9 22 3 63 44 83 23 4 94 27 62 73 12 13))
```

Die zugehörige aufsteigend sortierte Liste liefert

```
> (sort< usls)
'(1 3 4 9 12 13 22 23 27 44 62 63 73 83 94)
```

Zur Entwicklung von `sort<` beschreiben wir die Resultatliste.

1. Elementarfälle:

   a) Ist die zu sortierende Liste leer, so ist die leere Liste das Ergebnis.

   b) Ist die zu sortierende Liste einelementig, so ist diese Liste das Ergebnis.

   Um die rekursiven Fälle auszudrücken, nehmen wir uns zwei sehr einfache Beispiele vor. `(sort< '(2 4 8 3 9)) = '(2 3 4 8 9) = (cons 2 '(3 4 8 9)) = (cons 2 (sort< '(4 8 3 9)))`. Die angegebene Rechnung gilt jedoch nur, wenn das erste Element der zu sortierenden Liste, also 2, kleiner ist als das erste Element der sortierten Restliste (ohne die 2, also die 3). Da $2 < 3$ ist das der Fall. Diese Überlegung wird in 2a verwendet.
   Zur Vorbereitung von 2b benötigen wir ein anderes Beispiel: `(sort< '(4 2 8 3 9))`. Die für 2a angegebene Bedingung ist hier nicht erfüllt, denn $4 < 2$ ist falsch. Das Resultat ergibt sich aus `(cons 2 (sort< (cons 4 (sort< '(8 3 9)))))`. Die `2` im Beispiel ist das erste Element der sortierten Restliste (ohne `4`). Die `4` ist das erste Element der unsortierten Liste.

2. Rekursive Fälle:

   a) Wenn das erste Element `x` der zu sortierenden Liste `ls` kleiner ist als das erste Element der sortierten Restliste ohne das erste Element `(rest ls)`, so ergibt sich das Resultat aus der sortierten Restliste mit vorangestelltem `x`.

   b) Anderenfalls ergibt sich die sortierte Gesamtliste aus `(cons y (sort< (cons x (rest (sort< (rest ls))))))`, wobei `y = (first (sort< (rest ls)))` und `x = (first ls)`. Dies lässt sich nur schwer versprachlichen.

Die folgende Prozedur setzt die erarbeiteten Fälle um.

```
(define sort<
  (lambda (ls)
    (cond
      [(null? ls) '()]
      [(null? (rest ls)) ls]
      [(< (first ls)(first (sort< (rest ls))))
       (cons (first ls) (sort< (rest ls)))]
      [else
```

```
        (cons
         (first (sort< (rest ls)))
         (sort< (cons (first ls) (rest (sort< (rest ls))))))]))))

> (sort< usls)
'(1 3 4 9 12 13 22 23 27 44 62 63 73 83 94)
```

Es ist schon beeindruckend, welche deskriptive Kraft mit Rekursion erzielt werden kann.

2.2 Werte erster Klasse

2.2.1 Begriff

Nach Christopher Strachey (1916-1975) nennt man einen Racket-Wert einen *Wert*[6] *erster Klasse*, wenn er

- mit einer Variablen benannt,
- als Eingabewert (Argument) an eine Prozedur übergeben,
- von einer Prozeduranwendung als Resultat zurückgegeben und
- mit Datenstrukturen gespeichert (explizit gemacht)

werden kann.

Für Zahlen, Zeichen, Wahrheitswerte und Zeichenketten ist das offensichtlich der Fall. Dies gilt für alle höheren Programmiersprachen.

Wie steht es aber mit Prozeduren? Sind sie auch Werte erster Klasse?

Für Racket lautet die Antwort: Ja. In vielen anderen Sprachen versucht man mit (abstraktionsgefährdenden) Hilfskonstruktionen, etwa mit Zeigern (pointer) und Referenzen auf dynamische Datenstrukturen, formal Vergleichbares zu erreichen.

2.2.2 Prozeduren höherer Ordnung

Wir betrachten zunächst die Möglichkeit, *Prozeduren als Argumente von Prozeduren* zu benutzen. Solche Prozeduren nennen wir *Prozeduren höherer Ordnung* (higher order functions/procedures).

Ein Musterbeispiel ist die eingebaute Prozedur `map`. Sie erwartet zwei Argumente, nämlich eine einstellige Prozedur und eine Liste. Ihr Rückgabewert ist eine Liste, die sich aus der

[6] Mit gleicher Bedeutung spricht man auch von „Objekt“ erster Klasse. Wir bevorzugen es hier statt dessen von Wert (first class value) zu sprechen, um Verwechslungen mit dem Objektbegriff in der objektorientierten Programmierung zu vermeiden.

Anwendung der Prozedur auf jedes Element der Eingabeliste (unter Beibehaltung der Reihenfolge) ergibt.

```
> (map sqrt '(1 2 3 4 5))
(1 1.4142135623730951 1.7320508075688772 2 2.23606797749979)
```

Die einstellige Prozedur braucht nicht benannt zu sein.

```
> (map (lambda (a) (* a a)) '(1 2 3 4))
(1 4 9 16)
```

Für ein komplexeres Beispiel beziehen wir uns auf die Prozedur `sort<` in Abschn. 2.1.6. Nachdem wir die Sortierung auf die Anwendung der Kleinerrelation eingeschränkt hatten, ist es naheliegend, dies nun aufzuheben. Zur *Generalisierung* auf eine beim Aufruf wählbare Binärrelation sind lediglich zwei Dinge notwendig:

1. Ersetzen des Relationszeichens `<` durch einen Platzhalter `rel`
2. Verwendung des eingeführten Platzhalters als formalen Parameter der verallgemeinerten Prozedur `mysort`. Zu beachten ist die Verwendung je eines zusätzlichen aktuellen Parameters in allen Aufrufen von `mysort`.

Das Ergebnis lautet:

```
(define mysort
  (lambda (ls rel)
    (cond
      [(null? ls) '()]
      [(null? (rest ls)) ls]
      [(rel (first ls)(first (mysort (rest ls) rel)))
       (cons (first ls) (mysort (rest ls) rel))]
      [else
       (cons
        (first (mysort (rest ls) rel))
        (mysort (cons (first ls) (rest (mysort (rest ls) rel))) rel))])))

;(mysort usls <)
; '(1 3 4 9 12 13 22 23 27 44 62 63 73 83 94)
;(mysort usls >)
; '(94 83 73 63 62 44 27 23 22 13 12 9 4 3 1)
```

Aufgabe 2.2:
Verwenden Sie die in Racket eingebaute Prozedur `sort` zur Sortierung einer Liste mit einer wählbaren Ordnungsrelation. Schreiben Sie eine zweistellige Prozedur `lex?` für die Zeichenketten a und b, die für `(lex? a b)` auf Zeichenebene(!) den Wert `#t` ermittelt, wenn `a` in der lexikografischen Ordnung vor `b` steht, und ansonsten `#f`. Hinweis: `lex?` soll das Gleiche leisten wie `string<?`. Verwenden Sie die folgenden beiden Aufrufe:

```
(sort '("haha" "jajd" "ksksss" "jaddj" "holla") lex?)
(sort '("haha" "jajd" "ksksss" "jaddj" "holla") string<?).
```

2.2.3 Prozedur als Rückgabewert

Ein typisches Beispiel einer Prozedur, deren Wert wiederum eine Prozedur ist, liefert die Analysis: die *Ableitung einer Funktion*. So ist beispielsweise die Ableitung der Funktion $f : x \mapsto x^2$ die Funktion $f' : x \mapsto 2x$.

Die Mathematik klärt, wie die Funktion f' als Ableitung von f berechnet wird:

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}.$$

In einer zugehörigen Racket-Prozedur `ableitung` ersetzen wir den eigentlichen Grenzübergang ($h \to 0$) durch eine sehr kleine (aber eben nicht differenziell kleine) Zahl h, z. B. $h = 0.000001$. Von daher können wir nur eine Näherung für $f'(x_0)$ erwarten.

```
(define ableitung
  (lambda (f)
    (lambda (x)
      (let ([h 0.000001])
        (/ (- (f (+ x h))(f x)) h)))))
```

`ableitung` ist eine Prozedur, die sowohl eine Prozedur als Eingabe erwartet, als auch eine Prozedur zurückgibt. In der Tat ergibt die Ableitung der mit dem angegebenen λ-Ausdruck beschriebenen Prozedur wieder eine Prozedur:

```
> (ableitung (lambda (x)(* x x)))
#<procedure>
```

Die Anwendung dieser Prozedur auf $x_0 = 4$ liefert einen Wert in der Nähe von $f'(x_0) = 2x_0 = 8$.

```
> ((ableitung (lambda (x)(* x x))) 4)
8.000000999430768
```

Aufgabe 2.3:
Schreiben Sie eine Racket-Prozedur `compose`, die zwei Prozeduren für die einstelligen Funktionen $f, g : \mathcal{N} \mapsto \mathcal{N}$ nimmt und eine Prozedur für $h : \mathcal{N} \mapsto \mathcal{N}$ mit $h(n) = f(g(n))$ zurückgibt. Für h schreibt man auch $h = f \circ g$ und sagt „f nach g“.

2.2.4 „Daten als Programme“ und „Programme als Daten“

In Abschn. 1.3.7 haben wir bereits festgestellt, dass sich Racket-Listen zur Repräsentation von Daten als auch von Programmen (Prozedurdefinitionen) syntaktisch nicht

unterscheiden. Listen werden vom Racket-Interpreter grundsätzlich evaluiert, wenn dies nicht mit `quote` bzw. `quasiquote` verhindert wird. Diese Quote-Operationen machen aus Prozeduranwendungen und λ-Ausdrücken Daten, die als solche weiterverarbeitet werden können.

```
> ((lambda (n)(* n n)) 3)
9
> '((lambda (n)(* n n)) 3)
((lambda (n)(* n n)) 3)
```

Umgekehrt ist es ebenso möglich, Daten „zum Leben zu erwecken“. Auf diese Weise können aus passenden Daten Prozeduren oder Prozeduranwendungen entstehen. Dies ist in bestimmten Anwendungsbereichen, wie etwa bei adaptiven Algorithmen in der künstlichen Intelligenz, sehr willkommen. Es gibt auch einfache Anwendungen, wie etwa Online-Funktionsplotter: In einem Eingabefeld wird eine Beschreibung der Funktion, deren Graph gedruckt werden soll, eingegeben. Zur Berechnung darzustellender Punkte muss die Funktion angewandt werden.

Racket bietet hierfür zwei überaus leistungsfähige Sprachelemente, nämlich `eval` und `apply`. Das erste Beispiel zeigt, wie das Sprachelement `eval` wirkt. Es handelt sich praktisch um den Aufruf des Racket-Interpreters in der globalen Umgebung. Das zweite Beispiel lässt erkennen, wie ein als Liste mittels Listen-Konstruktor erzeugter Racket-Ausdruck evaluiert wird.

```
> (eval '((lambda (n)(* n n)) 3))
9
> (eval (cons '+ '(1 2 3)))
6
```

Für `(eval (cons '+ '(1 2 3)))` kann man auch

```
> (apply + '(1 2 3))
6
```

schreiben. `apply` erwartet eine Prozedur, die auf die Argumentliste anwendbar ist. In dem betrachteten Beispiel gilt

```
(apply + '(1 2 3))
     = (+ 1 2 3)
     = (eval '(+ 1 2 3))
     = (eval (cons '+ '(1 2 3))).
```

```
> (apply (lambda (a b) (+ a b)) '(1 2))
3
```

entspricht

```
> ((lambda (a b) (+ a b)) 1 2)
3
```

Das folgende kleine Beispiel illustriert, wie man mit sehr wenig Code die Anzahl der Vokale in einer Zeichenkette bestimmt.

```
(define vokal
  (lambda (z)
    (if (member z '(#\a #\e #\i #\o #\u))
        1
        0)))

(define vokalanzahl
  (lambda (zk)
    (apply + (map vokal (string->list zk)))))

> (vokalanzahl "kaeeij")
4
> (vokalanzahl "kztrwjzz")
0
```

Zunächst liefert `(map vokal (string->list "kaeeij"))` die Liste `'(0 1 1 1 1 0)`. Je nachdem, ob es sich bei dem betreffenden Buchstaben um einen Vokal handelt oder nicht, steht `1` oder `0` in der Liste. Anschließend wird via `apply` der Ausdruck `(+ 0 1 1 1 1 0)` ausgewertet.

In funktionsorientierten Sprachen kann man also die Evaluation von Ausdrücken ebenso unterdrücken, wie man sie andererseits erzwingen kann.

Zwei anspruchsvollere Beispiele sollen zeigen, wie kompakt die aus der Mathematik bekannte *Potenzmenge* $\wp$ (endlicher Mengen) und das *kartesische Produkt* zweier endlicher Mengen in Racket beschrieben werden können. Die mathematischen Definitionen lauten:

$$\begin{aligned} \wp(X) &= \{A \mid A \subseteq X\} \\ A \times B &= \{(a,b) \mid a \in A \ \underline{\text{und}}\ b \in B\} \end{aligned}$$

Die Elemente des kartesischen Produkts $A \times B$ sind Paare, die wir adäquat mit Racket-Paaren repräsentieren. Jedes Element der ersten Menge wird mit jedem Element der zweiten „gepaart". Wir nehmen zunächst an, dass $A = \{1, 2, 3\}$ und $B = \{x\}$.

```
> (map (lambda (a) (cons a 'x)) '(1 2 3))
'((1 . x) (2 . x) (3 . x))
```

Wenn B mehr als ein Element besitzt, z. B. $B = \{h, i, j, k\}$, müssen wir für `'x` im obigen Ausdruck einen Parameter, z. B. `b`, platzieren: `(lambda (b) (map (lambda (a) (cons a b)) '(1 2 3)))`. Ein Aufruf mit `'x` führt zum bekannten Ergebnis:

```
> ((lambda (b) (map (lambda (a) (cons a b)) '(1 2 3))) 'x)
'((1 . x) (2 . x) (3 . x))
```

Um für alle Elemente der Menge B die entsprechenden Verbindungen herzustellen, verwenden wir wieder ein `map`:

```
> (map
   (lambda (b)
     (map (lambda (a) (cons a b)) '(1 2 3)))
   '(h i j k))

'(((1 . h) (2 . h) (3 . h))
  ((1 . i) (2 . i) (3 . i))
  ((1 . j) (2 . j) (3 . j))
  ((1 . k) (2 . k) (3 . k)))
```

Wie wir sehen, entsteht eine Liste mit vier Elementen, die ihrerseits Listen sind. Dies entspricht nicht unserem Wunsch, denn wir erwarten genau eine Ergebnisliste aus Paaren. Die erforderliche Korrektur geschieht sehr elegant durch Anwendung `apply` von `append` auf diese Liste.

```
> (apply
    append
     (map
       (lambda (b)
         (map (lambda (a) (cons a b)) '(1 2 3)))
   '(h i j k)))

((1 . h) (2 . h) (3 . h) (1 . i) (2 . i) (3 . i)
 (1 . j) (2 . j) (3 . j) (1 . k) (2 . k) (3 . k))
```

Aufgabe 2.4:
Geben Sie eine vollständige Prozedurdefinition `kreuz` für das kartesische Produkt zweier endlicher Mengen an.

Die Idee, die der folgenden Prozedurdefinition `potenzmenge` zur Berechnung der *Potenzmenge* (engl.: powerset) einer vorgegebenen endlichen Menge zugrunde liegt, gewinnt man aus dem Aufbauschema für Potenzmengen. Wir versuchen aus dem folgenden Beispiel eine allgemeingültige Aussage abzuleiten.

$$\begin{aligned}
\wp(\{\}) &= \{\{\}\} \\
\wp(\{c\}) &= \{\{\}, \{c\}\} \\
\wp(\{b, c\}) &= \{\underline{\{\}}, \underline{\{b\}}, \underline{\{c\}}, \underline{\{b, c\}}\} \\
\wp(\{\mathbf{a}, b, c\}) &= \{\underline{\{\}}, \{\mathbf{a}\}, \underline{\{b\}}, \{\mathbf{a}, b\}, \underline{\{c\}}, \{\mathbf{a}, c\}, \underline{\{b, c\}}, \{\mathbf{a}, b, c\}\}
\end{aligned}$$

Besonders aussagekräftig sind die letzten beiden Zeilen. $\wp(\{a, b, c\})$ erhält man wie folgt: Übernimm' für jedes Element (Menge) aus $\wp(\{b, c\})$ dieses Element selbst und die Menge, die sich durch Einfügen des neuen Elements a in diese Menge ergibt. D. h. für $\{\}$ werden $\{\}$ und $\{a\}$ in die Potenzmenge übernommen, für $\{b\}$ sind es $\{b\}$ und $\{a, b\}$ usw.

```
> (map (lambda (x) (list x (cons 'a x)))
       '(()(b)(c)(b c)))
'((() (a)) ((b) (a b)) ((c) (a c)) ((b c) (a b c)))
```

Wir sehen, dass die gewünschten Elemente zwar allesamt in der Liste vorhanden sind, jedoch entspricht deren Struktur nicht unseren Vorstellungen. Durch Anwendung (`apply`) von `append` wird die Liste „geglättet“.

```
> (apply
    append
    (map (lambda (x) (list x (cons 'a x)))
    '(()(b)(c)(b c))))
'(() (a) (b) (a b) (c) (a c) (b c) (a b c))
```

Diesen Racket-Ausdruck können wir nun in einer leicht verallgemeinerten Form direkt in `powerset` verwenden.

```
(define powerset
  (lambda (set)
    (if (null? set)
        '(())
        (apply
         append
         (map
          (lambda (x) (list x (cons (first set) x)))
          (powerset (rest set)))))))

> (powerset '(a b c))
'(() (a) (b) (a b) (c) (a c) (b c) (a b c))
```

Wir betrachten nun ein alternatives Aufbauschema für Potenzmengen: In der jeweils betrachteten Zeile wird die vorherige übernommen. Hinzu kommt eine Menge, die sich aus der übernommenen ergibt, wenn man in jedes Element das neue Element der Ausgangsmenge aufnimmt. Für $\wp(\{a,b,c\})$ bedeutet das: Übernimm' zuerst $\wp(\{b,c\})$, d. h. $\{\{\},\{b\},\{c\},\{b,c\}\}$, und füge die Mengen hinzu, die sich aus $\wp(\{b,c\})$ durch das Einfügen von a in jedem Element ergeben, d.h. $\{\{a\},\{a,b\},\{a,c\},\{a,b,c\}\}$. Das Ergebnis ergibt sich aus der Vereinigung der beiden genannten Mengen.

$$\begin{aligned}
\wp(\{\}) &= \{\{\}\} \\
\wp(\{c\}) &= \{\{\},\{c\}\} \\
\wp(\{b,c\}) &= \{\underline{\{\}},\underline{\{b\}},\underline{\{c\}},\underline{\{b,c\}}\} \\
\wp(\{\mathbf{a},b,c\}) &= \{\underline{\{\}},\underline{\{b\}},\underline{\{c\}},\underline{\{b,c\}}\{\mathbf{a}\},\{\mathbf{a},b\},\{\mathbf{a},c\},\{\mathbf{a},b,c\}\} \\
&= \wp(\{b,c\}) \cup \{\{a\},\{a,b\},\{a,c\},\{a,b,c\}\}
\end{aligned}$$

Dies lässt sich nun unmittelbar in eine Racket-Prozedurdefinition übertragen.

```
(define potenzmenge
  (lambda (ls)
    (if (null? ls)
        '(())
        (let* ([pm-alt (potenzmenge (rest ls))]
               [zuwachs (map (lambda (x) (cons (first ls) x)) pm-alt)])
          (append pm-alt zuwachs)))))

> (potenzmenge '(a b c))
'(() (c) (b) (b c) (a) (a c) (a b) (a b c))
```

Die Ergebnisse der beiden Prozeduraufrufe stimmen überein.

Aufgabe 2.5:
Verwenden Sie die im Folgenden gegebene Prozedur `schnapszahl?` um aus einer Liste natürlicher Zahlen von 0 bis n alle durch `schnapszahl?` definierten Schapszahlen herauszusuchen und als Liste zurückzugeben. In der Definition kommt das neue Sprachelement `andmap` vor. Die angegebene Definition illustriert, wie ein Problem aus der „Zahlenwelt" in der „Zeichenkettenwelt" gelöst wird. Dies ist ein durchaus gängiges Vorgehen und unterstreicht die große Bedeutung von Entscheidungen im Rahmen der Datenmodellierung in der Programmierung.

```
(define schnapszahl?
  (lambda (n)
    (let ([n-charlist (string->list (number->string n))])
      (if (< (length n-charlist) 2)
          #f
          (andmap
           (lambda (c) (char=? (first n-charlist) c))
           (rest n-charlist))))))

> (filter schnapszahl? (range 10000))
'(11 22 33 44 55 66 77 88 99 111 222 333 444 555 666 777 888
  999 1111 2222 3333 4444 5555 6666 7777 8888 9999)
```

Aufgabe 2.6:
Der Aufruf `(length (filter prim? (range 1000)))` berechnet die Anzahl von Primzahlen bis 1000. Entwickeln Sie eine geeignete Prozedur `prim?`.

Aufgabe 2.7:
Schreiben Sie eine Racket-Prozedur `wtab`, die den definierenden Term eines Polynoms p in x mit ganzzahligen Koeffizienten sowie zwei ganze Zahlen a und b ($a < b$) nimmt und eine Wertetafel $x \mid p(x)$ für $x := a(1)b$ (lies: x läuft von a in 1-er Schritten bis einschl. b) ausdruckt. Beispiel: $p = 5x^2 - 3x + 4$, $a = -2$, $b = 11$.

Der zugehörige Aufruf lautet `(wtab '(+ (- (* 5 x x)(* 3 x)) 4) -2 11)`.

Das Hin- und Herschalten zwischen Daten und Programmen ist in vielen anderen Programmiersprachen unmöglich. Dort sind bestimmte Schlüsselwörter fest vorgegeben, deren Verwendung keineswegs in beliebigem Kontext möglich ist. Dies hat auch Gründe in der compilierenden[7] vs. interpretativen Verarbeitung der Programme.

2.2.5 Funktionsobjekte mit Java

Prozeduren höherer Ordnung kann man grundsätzlich auch mit Java realisieren. Um das folgende Beispiel verstehen zu können, sind Grundkenntnisse in der Programmierung mit Java erforderlich. Alternativ kann man diesen Abschnitt auch überspringen.

Wir betrachten die folgende Summe:

$$\sum_{i=a}^{b} f(i) = f(a) + f(a+1) + \ldots + f(b).$$

Die Racket-Prozedur `summe` erwartet beim Aufruf die Eingaben für f, a und b mit $a \leq b$.

```
(define summe
  (lambda (f a b)
    (if (> a b)
        0
        (+ (f a) (summe f (+ a 1) b)))))
```

`(summe (lambda (n) (* n n)) 1 4)` liefert das Ergebnis $30 = \sum_{i=1}^{4} i^2$.

Mit Java wird zunächst ein Interface `SummenFunktion` definiert.

```
public interface SummenFunktion {
  public double f(int i);
}
```

Die gewünschte Funktionsklasse `Quadrat` implementiert dieses Interface.

```
public class Quadrat implements SummenFunktion {
  public double f(int i) {
    double d = (double) i;
      return d * d;
  }
}
```

[7] Compiler übersetzen einen Programmtext zuerst vollständig in ein äquivalentes Programm einer Zielsprache, oft in Maschinencode. Das so erzeugte Compilat wird anschließend interpretiert. Bei der Compilation kann eine Reihe von Fehlertypen abgefangen werden, die bei Interpretation des Quellcodes als Laufzeitfehler auftreten würden. Mit der Übersetzung von Programmiersprachen befassen wir uns hier nicht weiter.

Die Klasse `Summation` besteht nur aus der Methode, die die Summe gemäß obiger Formel berechnet. Semantisch entspricht sie der Racket-Prozedur.

```
public class Summation {
  double summe(SummenFunktion sf, int a, int b) {
    double sum = 0.0;
    for (int i = a; i <= b; i++) {
      sum += sf.f(i);
    }
    return sum;
  }
}
```

Im Java-Hauptprogramm wird mit einer Instanz der Klasse `Summation` gearbeitet. Das ist nichts Besonderes. Der „Trick“ besteht darin, ein Funktionsobjekt `sumfkt` als Instanz der Klasse `Quadrat` zu erzeugen und dafür den Interface-Typ `SummenFunktion` anzugeben. Damit kann der Methodenaufruf `summe(sumfkt, 1, 4)` für `s` stattfinden.

```
Summation s = new Summation();
SummenFunktion sumfkt = new Quadrat();  // !!!
double sum = s.summe(sumfkt, 1, 4);
System.out.println(sum);
```

Das Ergebnis ist ebenfalls 30.

Auch ohne Java-Programmierkenntnisse kann man erkennen, dass die Verwendung von Prozeduren höherer Ordnung in Java prinzipiell und systematisch möglich ist. Dafür sind jedoch Maßnahmen erforderlich, die sich von der klaren Linie funktionsorientierter Programmiersprachen deutlich unterscheiden.

Ab Java 8 ist es möglich, Aspekte funktionaler Sprache explizit auszudrücken.

2.3 Transformation echt rekursiver Prozeduren in endständige

In den vorangehenden Abschnitten haben wir folgende Erkenntnisse gewonnen:

1. Rekursion beschreibt (gesuchte) Werte präzise und abstrahiert dabei von den einzelnen Bearbeitungsschritten.
2. Rekursiv definierte Datentypen, wie z. B. Listen, implizieren deren rekursive Verarbeitung in natürlicher Weise.
3. In der Struktur rekursiver Prozeduren spiegelt sich die Datenstruktur wider und führt zu endständiger bzw. echter (linearer oder baumartiger) Rekursion.

Gerade im dritten Punkt findet man die Quelle kognitiver Effizienz: Bei einer gewissen Vertrautheit mit Rekursion, kann man entsprechende Prozeduren relativ leicht ableiten.

Mit der Entwicklung einer entsprechenden Prozedur sind wir eigentlich am Ziel, wenn da nicht das Performance-Problem wäre: Rekursiven Prozeduren wird oft mangelnde Ausführungsgeschwindigkeit vorgeworfen. In der Tat muss der Interpreter für echte Rekursion Stapelarbeit organisieren, um die oben beschriebene Speicherung lokaler Variablenwerte zu bewerkstelligen. Das kostet Zeit.

Es stellt sich daher die Frage nach der Übertragbarkeit echt rekursiver Prozeduren in endständig rekursive. Falls das im Allgemeinen möglich ist, könnte der Vorteil eines kognitiv effizienten Entwurfs mit dem der effizienten Verarbeitung kombiniert werden. Wir fragen also nach der Existenz eines allgemeingültigen Transformationsverfahrens für echt rekursive in endständig rekursive Prozeduren.

Wie wir noch in diesem Kapitel detaillierter ausführen werden, beruhen Programme in funktionalen Sprachen auf dem Funktionsbegriff der Mathematik. Folglich kann man mit ihnen rechnen, so wie mit anderen mathematischen Objekten. Das ist schon mal ein hoffnungsvoller Start bei der Suche eines solchen Transformationsverfahrens bei gleichbleibender Semantik.

Akkumulatortechnik

Diese Technik haben wir oben für `tailfak` bereits eingesetzt. Die Idee besteht darin, einen Wert statt durch echte Rekursion in einem Extra-Parameter zu generieren.

Wir sehen uns dazu noch ein Beispiel an: Die Prozedur `sum` beschreibt die Summe der Elemente (Zahlen) einer (nicht verschachtelten) Liste. `sum` ist echt rekursiv, da der rekursive Aufruf als Summand auftritt. Man spricht auch von *nachklappernden Operationen*, wenn die Anwendung eines Operators auf die Lieferung eines Operanden warten muss.

```
(define sum
  (lambda (ls)
    (if (null? ls)
        0
        (+ (first ls) (sum (rest ls))))))

> (sum '(1 2 3 4))
10
```

Die Transformation dieser Prozedur in eine endständig rekursive lässt das allgemeine Prinzip schon recht gut erkennen.

```
(define sumtail
  (lambda (ls result)
    (if (null? ls)
```

```
        result
        (sumtail (rest ls) (+ result (first ls))))))
```

Die hier vorgenommene Transformation kann leicht „nachgerechnet“ werden. Hierzu gehen wir von der oben definierten Funktion[8] *sum*

$$\begin{aligned} sum([]) &= 0 \\ sum([x|L]) &= x + sum(L) \end{aligned}$$

aus und definieren unter deren Verwendung eine Funktion *sumtail*:

$$sumtail(L, result) := result + sum(L).$$

Offensichtlich ist *sum* ein Spezialfall von *sumtail*:

$$sum(L) = 0 + sum(L) = sumtail(L, 0).$$

Hierbei wird ausgenutzt, dass 0 das neutrale Element der Addition natürlicher Zahlen ist. Nun können wir *sumtail* ohne *sum* angeben.

$$\begin{aligned} sumtail([], result) &= result + sum([]) \\ &= result + 0 \\ &= result \end{aligned}$$

Für den allgemeinen (rekursiven) Fall gilt unter Einsatz der Assoziativität der Addition natürlicher Zahlen:

$$\begin{aligned} sumtail([x|L], result) &= result + sum([x|L]) \\ &= result + (x + sum(L)) \\ &= (result + x) + sum(L) \\ &= sumtail(L, result + x) \end{aligned}$$

Damit erhalten wir die folgende Definition von *sumtail*.

$$\begin{aligned} sumtail([], result) &= result \\ sumtail([x|L], result) &= sumtail(L, result + x) \end{aligned}$$

Die Implementierung von *sumtail* in obiger Prozedur `sumtail` ist offensichtlich.

Zum bequemen Aufruf ist es sinnvoll, die Prozedur `sumtail` in `sum1` einzubetten.

```
(define sum1
  (lambda (ls)
```

[8] Die Schreibweise $[x|L]$ bedeutet, dass der betrachtete Wert eine Liste ist, deren erstes Element x heißt und die Restliste mit L bezeichnet wird. Ein solcher auf Musterabgleich beruhender Head-Tail-Separator wird vor allem in der logikbasierten Programmierung verwendet.

```
    (define sumtail
      (lambda (ls result)
        (if (null? ls)
            result
            (sumtail (rest ls) (+ result (first ls))))))
    (sumtail ls 0)))

> (sum1 '(1 2 3 4))
10
```

Aufgabe 2.8:
Protokollieren Sie die Abarbeitungen von `(sum '(1 2 3 4))` und `(sum-tail '(1 2 3 4) 0)`.

```
>(sum '(1 2 3 4))
> (sum '(2 3 4))
> >(sum '(3 4))
> > (sum '(4))
> > >(sum '())
< < <0
< < 4
< <7
< 9
<10
10
```

```
>(sum-tail '(1 2 3 4) 0)
>(sum-tail '(2 3 4) 1)
>(sum-tail '(3 4) 3)
>(sum-tail '(4) 6)
>(sum-tail '() 10)
<10
10
```

Es ist wünschenswert, dass die vorgestellte Transformation vom Compiler und nicht vom Programmierer durchgeführt wird. Ist sie allgemeingültig?

Continuation passing style (CPS)

Bei komplexeren Prozeduren kann es vorkommen, dass die Akkumulatortechnik und andere Methoden, die wir hier nicht betrachten, erfolglos bleiben. In diesen Fällen hilft eine Technik, die als *Continuation passing style* (CPS) bekannt geworden ist.

Wie der Name erahnen lässt, sind *Continuations* Prozeduren, die die Fortsetzung einer Berechnung charakterisieren. Bis auf die Anzeige/Rückgabe eines Wertes auf Kommunikationsebene besitzt jeder der zu berechnenden Teilausdrücke eine solche Fortsetzung. Beispielsweise besteht in `(* 3 (+ 4 6))` die Fortsetzung der Verarbeitung des zweiten Faktors `(+ 4 6)` = `10` in der Berechnung des Produktes `(* 3 10)` = `30`. Der Ausdruck `(lambda (v) (* 3 v))` evaluiert genau zu dieser Fortsetzung. Ihr übergeben wir nun den Wert des Ausdrucks `(+ 4 6)`: `((lambda (v) (* 3 v))(+ 4 6))` = `30`.

Auch in Prozedurdefinitionen können wir Continuations als aktuelle Parameter vorsehen, so wie das für Prozeduren in Racket allgemeingültig ist. Das eigentliche Ergebnis einer

Berechnung kann dann an eine Continuation übergeben, statt direkt ausgegeben werden.

Der binären Additionsprozedur

```
(define op+
  (lambda (a b)
    (+ a b)))

> (op+ 2 3)
5
```

wird deshalb ein dritter Parameter verordnet. Beim Aufruf nimmt die nun `k+` genannte Prozedur neben den beiden Summanden `a` und `b` eine Continuation `cont`, an die das Ergebnis `(+ a b)` übergeben wird. Im Beispiel sorgt die aktuelle Continuation dafür, dass die Summe ausgegeben wird.

```
(define k+
  (lambda (a b cont)
    (cont (+ a b))))

> (k+ 2 3 (lambda (res) res))
5
```

Nachdem wir uns an diesem einfachen Beispiel mit CPS ein wenig vertraut gemacht haben, können wir diese Technik nun auf eine echt rekursive Prozedur (`fak`) anwenden.

```
(define fak
  (lambda (n)
    (if (= n 1)
        1
        (* n (fak (- n 1))))))
```

Die CPS-Version lautet:

```
(define fak-cpsed
  (lambda (n cont)
    (if (= n 1)
        (cont 1)
        (fak-cpsed (- n 1) (lambda (res) (cont (* n res)))))))
```

Wieder macht ein Tracing deutlich, dass keine Stapelarbeit stattfindet: Es gibt keine Ein-/Ausrückungen.

```
(trace fak-cpsed)
> (fak-cpsed 5 (lambda (res) res))
>(fak-cpsed 5 #<procedure:...sktop\cpsing.rkt:97:13>)
>(fak-cpsed 4 #<procedure:...sktop\cpsing.rkt:94:27>)
>(fak-cpsed 3 #<procedure:...sktop\cpsing.rkt:94:27>)
>(fak-cpsed 2 #<procedure:...sktop\cpsing.rkt:94:27>)
```

```
>(fak-cpsed 1 #<procedure:...sktop\cpsing.rkt:94:27>)
<120
120
```

Diese Technik ist für diverse Anwendungskontexte (Abbruch rekursiver Abläufe; Exception handling usw.) sehr nützlich.

Aufgabe 2.9:
Wenden Sie die CPS-Methode auf die oben definierte Prozedur `sum` an.

Aufgabe 2.10:
Wenden Sie die CPS-Methode auf eine Prozedur `prod` an, die analog zu `sum` das Produkt der Zahlen einer Liste berechnet. Dabei soll die Tatsache ausgenutzt werden, dass das Produkt einer Zahl mit 0 das Ergebnis 0 liefert.

Da die CPS-Version bei etwas komplexeren rekursiven Prozeduren nicht mehr so offensichtlich hingeschrieben werden kann, lohnt sich ein allgemeines Verfahren zur Kenntnis zu nehmen. Dieses kann mit viel Theorie begründet werden. Wir begnügen uns jedoch mit folgendem Vorgehensmodell:

1. Füge jeder Prozedurdefinition einen zusätzlichen Parameter `k` hinzu.
2. Statt ein Resultat in einer Prozedur zurückzugeben, wird es an `k` geleitet.
3. Hole einen verschachtelten Aufruf einer (nicht primitiven) Prozedur aus ihrem Unterausdruck heraus, indem Du den Prozeduraufruf durch eine Variable X ersetzt. Wickle den Ausdruck ein mittels `(lambda (X) ...)` und liefere die resultierende Continuation als weiteres Argument der Prozedur.
   Beispiel: `(add1 (f a))` $\Rightarrow$ `(f a (lambda (X) (add1 X)))`.

Wir illustrieren die Anwendung dieser Handlungsvorschrift auf die Prozedur `mymap`. Semantisch soll sie mit `map` übereinstimmen.

```
(define mymap
  (lambda (f ls)
    (if (null? ls)
        '()
        (cons (f (car ls)) (map f (cdr ls))))))
```

Das Ergebnis ist

```
(define map-cpsed
  (lambda (f ls k)
    (if (null? ls)
        (k '())
        (f (car ls)
           (lambda (v)
```

```
              (map-cpsed
                f
                (cdr ls)
                (lambda (v2) (k (cons v v2)))))))))

> (map-cpsed (lambda (n cont)(cont (* n n))) '(1 2 3 4) (lambda (x) x))
'(1 4 9 16)
```

Natürlich lohnt sich auch hier ein Trace.

Aufgabe 2.11:
Wenden Sie das CPS-ing auf die folgenden beiden echt rekursiven Prozeduren an:
1. Ackermann-Peter-Funktion
2. Horner-Schema zur Berechnung von Polynomen

Wir fassen zusammen:

Man kann zeigen, dass jede echt rekursive Prozedur in eine semantisch äquivalente endständig rekursive Prozedur überführt werden kann.

Gelingt dies durch eine Transformation, d. h. eine mathematische Rechnung mit dem Programm als Argument, so erhalten wir eine Prozedur, die durch einen geeigneten Compiler in Schleifen bzw. Gotos umgesetzt werden kann.

Die CPS-Technik ist eher ein Programmiertrick. Obwohl sich auch im Ergebnis des CPS-ings die für echte Rekursion typische Stapelarbeit erübrigt, werden die Continuations bei jedem Aufruf im entsprechenden Parameter verkettet. Technisch hat dafür der Compiler mindestens den gleichen Aufwand zu betreiben, wie beim Stapelprinzip.

Die stets mögliche Übertragung echt rekursiver Prozeduren in endständig rekursive führt also nicht zwangsläufig zu effizienten Programmen.

2.4 Evaluation von Ausdrücken in funktionalen Sprachen

2.4.1 Evaluation: gierig (eager, greedy) vs. verzögert (lazy)

Wir wiederholen: Racket-Ausdrücke haben eine sehr einfache syntaktische Form:

$$(Operator\ Operand_1\ Operand_2\ \ldots\ Operand_n)$$

Operator steht für einen Racket-Ausdruck, der zu einer Prozedur evaluiert, die beim Aufruf n Argumente erwartet. Das in Racket eingebaute Evaluationsverfahren arbeitet folgendermaßen:

1. Evaluiere den Operator.

2. Evaluiere die Operanden, d. h. die Argumente oder Eingaben der Prozedur.
3. Ersetze die (formalen) Parameter im Prozedurkörper an jeder vorkommenden Stelle durch die Argumente.
4. Evaluiere den so entstandenen Ausdruck.

Diese Form der Evaluation nennt man *gierig* (*eager evaluation*). Sie ist begierig, die Werte der Operanden kennenzulernen, selbst wenn diese in Schritt 4 gar nicht benötigt werden.

Ein einfaches Beispiel macht sofort klar, dass gierige Evaluation auch problematisch sein kann: Die folgenden Ausdrücke evaluieren zu 5. Die im zweiten Ausdruck verwendete Prozedur `fib` ist wie auf S. 30 rekursiv definiert.

```
> ((lambda (n) 5) 12)
5
> ((lambda (n) 5) (fib 100))
```

Wir werden nicht genügend Geduld aufbringen, um das Racket-Ergebnis für den zweiten Ausdruck abzuwarten. Die Berechnung von `(fak 100)` dauert viel zu lange. Dumm nur, dass dieses Ergebnis für Schritt 3 angewandt auf `(lambda (n) 5)` nicht benötigt wird. In `5` kommt der Parameter `n` als Platzhalter für `(fak 100)` nicht vor, so dass jede Prozeduranwendung den Wert `5` zurückgibt.

Eine naheliegende Idee zur Verbesserung der gierigen Strategie besteht darin, die Evaluation von Ausdrücken solange aufzuschieben, bis sie wirklich benötigt werden. Man nennt dies *verzögerte Evaluation* (*lazy evaluation*):

1. Evaluiere den Operator.
2. Ersetze die (formalen) Parameter im Prozedurkörper an jeder vorkommenden Stelle durch die *unevaluierten* Operanden.
3. Evaluiere den so entstandenden Ausdruck.

Wählt man in DrRacket durch Vermerk von `#lang lazy` in der ersten Zeile die Sprache Lazy Racket aus, so evaluiert der obige Ausdruck `((lambda (n) 5) (fib 100))` sofort zu `5`, selbst dann, wenn die Definition von `fib` nicht gegeben ist.

Die Parametervermittlung bei der verzögerten Evaluation nennt man *call by need* im Gegensatz zu *call by value* bei der gierigen Evaluation. Beide Techniken wurden bereits in Algol 60 – einer prozeduralen, imperativen, keineswegs funktionalen Programmiersprache – verwendet. Es gibt funktionale Sprachen, die ausschließlich verzögerte Evaluation verwenden.

Der Vorteil, unnötige Operandenevaluationen zu vermeiden, wird jedoch durch folgende Nachteile erkauft.

Namenskonflikt: Stimmt der Name einer Variablen in einem Operanden mit dem Namen einer lokalen Variablen im betrachteten Ausdruck über, kann es zu einem

falschen Evaluationsergebnis kommen. Beispiel:

```
(define x 4)
(define lazy-problem
  (lambda (arg)
    (let ([x 1])
      (< x arg))))
```

`(lazy-problem x) = (let ([x 1])(< x x)) = (< 1 1) = #f` ist falsch.

Unterscheidet man das globale `x` mit dem Wert `4` vom lokalen `x` mit dem Wert `1` innerhalb des `let`-Ausdrucks, wie das bei gieriger Standardevaluation in Racket der Fall ist, so liefert `(< 1 4)` den korrekten Wert `#t`:

```
> (lazy-problem x)
#t
```

Mehrfachevaluation: Gleiche Operanden werden so oft evaluiert, wie sie im Körper der Prozedur vorkommen. Zeitaufwendige Berechnungen addieren sich zeitlich auf. Beispiel:

```
> ((lambda (n) (* n n n)) (fib 37))
59722225363795389930809
```

Bei gieriger Evaluation beansprucht die Verarbeitung dieses Ausdrucks kaum mehr Zeit als für `(fib 37)` selbst.

Um diese beiden Nachteile zu beheben, wird Schritt 2 der verzögerten Evaluationsstrategie modifiziert: Anstelle die unevaluierten Operanden zu übergeben, werden die betreffenden Operanden ein wenig „vorverarbeitet“, genauer gesagt, deren Evaluation wird zunächst „eingeschläfert“. Jeder Parameter wird bei Bedarf zwangsevaluiert. Die funktionale Sprache Gofer verwendet diese verbesserte Form der verzögerten Evaluation.

In Racket, d. h. bei gieriger Evaluation, kann dies mit Hilfe der Sprachelemente `delay` (aufschieben) und `force` (erzwingen) nachgebildet werden.

```
(define fib37 (delay (fib 37)))
> fib37
#<promise:fib37>
```

Offenbar wird mit `delay` eine `promise`, d. h. ein Versprechen, diesen Ausdruck bei Bedarf zu evaluieren, erzeugt. Mit `force` soll dieses Versprechen eingelöst werden:

```
> (force fib37)
39088169
```

Hier fällt dann auch die erforderliche Rechenzeit an, da bei der Anwendung von `delay` der Ausdruck `(fib 37)` eingeschläfert und eben nicht berechnet wurde.

Interessant ist nun, dass `force` offenbar einen Seiteneffekt ausgelöst hat: Fragen wir im Anschluss an die Anwendung von `force` erneut nach dem Wert der Variablen `fib37`, so

hat sich etwas verändert:

```
> fib37
#<promise!39088169>
```

Der zwangsberechnete Wert von `(fib 37)`, nämlich 39088169, wurde in der `promise` gespeichert. Es ist nun zu erwarten, dass eine erneute Evaluation dieses Ausdrucks kaum Rechenzeit beansprucht und einfach auf den abgelegten Wert zugreift. Dies ist in der Tat der Fall.

```
> (force fib37)
39088169
```

2.4.2 Memoizing

Die oben beschriebene „Merkfähigkeit" von `force` drückt ein sehr wichtiges Programmierkonzept aus. Man nennt es *Memoizing* oder *Memoization*. Memoizing kann auch bei modernen Computeralgebrasystemen, wie etwa Maple, per Kommando zugeschaltet werden.

Etwas verallgemeinert besagt es, dass zur Berechnung eines Ausdrucks zuerst nachgeschaut wird, ob dessen Wert bereits vorliegt. Ist das der Fall, wird dieser Wert verwendet. Anderenfalls wird er berechnet und (für einen eventuellen späteren wiederholten Gebrauch) vermerkt. Zur Verwaltung der betrachteten Werte darf man zunächst an eine Tabelle denken.

Memoizing kann man sehr einfach nachbilden. Zur Implementierung unserer Tabellenvorstellung verwenden wir den in Racket vorhandenen Datentyp *Hash Table*. Eine Hash Table implementiert eine Abbildung von Schlüsseln (key) auf Werte (value). Sowohl Schlüssel als auch Werte können beliebige Racket-Datentypen sein, z. B. Zahl, Zeichenkette, Paar und Liste. `(make-hash)` erzeugt eine Hash Table. Der Zugriff auf einen bestimmten Wert geschieht über dessen Schlüssel.

Ein sehr leistungsfähiges Sprachelement für Hash Tables ist `hash-ref!`. Es nimmt eine Hash Table, einen Schlüsselwert und eine nullstellige Prozedur. `hash-ref!` liefert den mit dem Schlüssel assoziierten Wert, falls es einen gibt. Andererfalls wird der in die nullstellige Prozedur „eingewickelte" Ausdruck evaluiert. Dessen Ergebnis wird dem betreffenden Schlüssel als Wert zugewiesen.

```
(define fibs (make-hash))  // Erzeugung der Hash Table fibs

(define fib-mem1
  (lambda (n)
    (if (< n 2)
        1
        (+ (hash-ref! fibs (- n 1) (lambda () (fib-mem1 (- n 1))))
```

```
          (hash-ref! fibs (- n 2) (lambda () (fib-mem1 (- n 2)))))))
```

`(fib-mem1 37)` liefert das korrekte Resultat 39088169 in deutlich kürzerer Rechenzeit als `(fib 37)`.

Aufgabe 2.12:
Erklären Sie, warum das so ist. Geben Sie für `(fib 6)` eine Baumdarstellung an, die den gesamten Berechnungsprozess visualisiert.

Fibonacci-Zahlen können nun auch für deutlich größere Argumente berechnet werden.

```
> (map (lambda (n) (hash-ref fibs n)) '(0 1 2 3 4 5 6 7 8 9 10))
'(1 1 2 3 5 8 13 21 34 55 89)
```

zeigt die Werte der ersten 11 Schlüssel der Hash Table `fibs`.

In der Definition von `fib-mem1` wird auch eine Schwäche des verwendeten Vorgehens deutlich: Immer, wenn wir eine (rekursive) Prozedur memoizieren, müssen wir deren Defintionstext verändern. Das schreit geradezu nach Fehlern und Inkonsistenzen.

Deshalb entwickeln wir einen Operator, der den Definitionstext unverändert nimmt und die zugehörige memoisierte Prozedur zurückgibt. Außerdem stört die globale Variable für die Hash Table, da deren Wert außerhalb von `fib-mem` verändert werden könnte. `memo` leistet das Gewünschte in Gestalt einer Prozedur höherer Ordnung.

```
(define memo
  (lambda (fn)
    (let ((table (make-hash)))
      (lambda p
        (hash-ref!
         table
         p
         (lambda () (apply fn p)))))))
```

Die Anwendung von `memo` auf eine zu memoisierende Prozedur gibt eine Prozedur zurück, deren Aufruf in einer Umgebung stattfindet, die die betreffende Hash Table enthält. Nach außen ist diese Hash Table nicht sichtbar.

Wir steuern nun geradewegs auf einen Fehler zu, wenn wir meinen, dass
`(define fib-pseudo-mem (memo fib))` eine Prozedur mit den gewünschten Eigenschaften definiert. Aufrufe der Form `(fib-pseudeo-mem 37)` erfordern die von `fib` bekannten langen Rechenzeiten. Dies liegt daran, dass `fib` als Argument von `memo` vor dessen Anwendung evaluiert wird. Da `fib` zur klassischen Fibonacci-Prozedur evaluiert, gibt es kein Memoizing.

Um die (gierige) Evaluation des Arguments von `memo` zu verhindern, definieren wir die *Sonderform* `define-memoized`:

```
(define-syntax define-memoized
```

```
  (syntax-rules ()
    ((define-memoized id
       (lambda (p ...) body))
     (define id
       (memo (lambda (p ...) body))))))
```

Entspricht das nach der Zeile `(syntax-rules ()` angegebene Muster dem Aufruf, werden die durch Pattern Matching vorgenommenen Variablenbindungen wie bei einer textuellen Transformation (copy and paste) im Zielausdruck (die letzten beiden Zeilen) verwendet, bevor schließlich die Evaluation dieses Zielausdrucks stattfindet.

Für unser Beispiel benötigen wir die folgende Definition:

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

Gegenüber der Standarddefinition mit `define` haben wir hier `define-memoized`. Im Code von `fib-mem` gibt es keinen Unterschied zu `fib`.

Der Ausdruck `(fib-mem 37)` wird deutlich schneller berechnet als `(fib 37)`. Die Berechnung großer Fibonacci-Zahlen erfordert keine spürbare Bearbeitungszeit:

```
> (map (lambda (n) (fib-mem n)) '(30 40 50 60 70 80 90 100))
'(1346269
  165580141
  20365011074
  2504730781961
  308061521170129
  37889062373143906
  4660046610375530309
  573147844013817084101)
```

Die vorgestellte Technik zum Memoisieren von Prozeduren kann auf mehrstellige Funktionen ausgeweitet werden. Dies ist in der Makro-Definition von `define-memoized` bereits vorgesehen, erkennbar an `(p ...)`.

Wir brauchen also nichts zu verändern, wenn wir beispielsweise die Prozedur für die Ackermann-Peter-Funktion memoisieren möchten:

```
(define-memoized ap-mem
  (lambda (n m)
    (cond
      ((= n 0)(+ m 1))
      ((= m 0)(ap-mem (- n 1) 1))
      (else
```

```
        (ap-mem (- n 1)(ap-mem n (- m 1)))))))
```

Für (ap-mem 3 11) erhalten wir 16381.

2.4.3 Potenziell Unendliches: Streams

Häufig wird behauptet, man könne mit dem Computer nur Werte endlicher Dimension verarbeiten. Hiernach müssten unendliche Mengen für die Computerarbeit absolut tabu sein, denn man hat ja stets nur einen endlichen Speicher zur Verfügung, auch wenn er noch so groß ist.

In der Tat können Computer nicht sämtliche reellen Zahlen speichern, es gibt einfach zu viele davon. Die Unendlichkeit der Menge der reellen Zahlen übersteigt sogar die der natürlichen Zahlen. Man spricht von *überabzählbar* im Gegensatz zu *abzählbar* oder *potenziell unendlichen* Mengen. Bei letzteren ist der Computer nicht so hilflos, wie man denken könnte.

Streams (Ströme) sind potenziell unendliche Daten. In einem Programmiersystem können sie nur dann repräsentiert werden, wenn man das Konzept der *verzögerten Evaluation* (lazy evaluation), mit dem wir uns in Absch. 2.4.1 beschäftigt haben, einsetzt. Andernfalls würde sich ein nicht terminierender (nicht anhaltender) Prozess ergeben, wenn man versucht, derartige Daten zu erzeugen.

Im Racket-Manual kann man nachschlagen, welche Sprachelemente für Streams bereitgestellt werden, wenn man sie mit (require racket/stream) anfordert. Es ist aber auch sehr lehrreich, einige von ihnen selbst zu implementieren.

Typische Anwendungen für Streams sind Zahlenfolgen. Für die Folge der natürlichen Zahlen (ab einschl. der Zahl Null) gilt:

$$\begin{aligned}(0, 1, 2, 3, 4, ...) &= 0 \circ (1, 2, 3, 4, ...)\\ \text{(nat.-Zahlen-ab-0)} &= \text{Folge aus 0 und (nat.-Zahlen-ab-1)}\\ \texttt{(nat-stream 0)} &= \texttt{(stream-cons 0 (nat-stream (+ 0 1))}\end{aligned}$$

In dieser Übersicht kann man gut erkennen, dass Streams – ähnlich wie Listen – durch ein Paar aus dem ersten Glied (head) des Streams und dem entsprechenden Reststream (tail) charakterisiert werden, wobei der cdr-Teil mit delay „eingeschläfert“ werden muss. Will man mit dem Reststream operieren, ist ein force erforderlich:

```
(define stream-first car)
(define stream-rest
  (lambda (stm)
    (force (cdr stm))))
```

Analog zur Listen-Datenstruktur erwarten wir stream-cons, dessen Einsatz wir in unserem oben begonnenen Beispiel vorstellen. Für das Anfangsglied der Folge der natürlichen

Zahlen – hier ist das die Null – verwenden wir den Parameter `start` und erhalten aus der letzten Zeile unmittelbar die folgende Prozedur:

```
(define nat-stream
  (lambda (start)
    (stream-cons start (nat-stream (+ start 1)))))
```

Der Aufruf `(nat-stream 0)` liefert die Folge der natürlichen Zahlen.

```
(define natzahlen (nat-stream 0))
> natzahlen
'(0 . #<promise:...>)
```

An dieser Ausgabe sieht man noch einmal sehr schön die Architektur eines Streams als Paar bestehend aus einem Kopfglied und einem Versprechen zur Berechnung des Reststreams.

Wir wollen uns die ersten 20 Glieder der Folge der natürlichen Zahlen ansehen. Zu diesem Zweck entwickeln wir eine Prozedur `print-stream`, die einen Stream und die Anzahl der anzuzeigenden Elemente ausgibt.

```
(define print-stream
  (lambda (stm n)
    (if (= n 0)
        (printf "...~n")
        (begin
          (printf "~a, " (stream-first stm))
          (print-stream (stream-rest stm) (- n 1))))))

> (print-stream natzahlen 20)
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, ...
```

Das nächste Beispiel ist eine Zahlenfolge, die natürliche Zufallszahlen aus dem Intervall $[0, 100]$ enthält. Wie bei den natürlichen Zahlen beschreiben wir das Wunschresultat:

$$(12, 100, 2, 34, 41, ...) = 12 \circ (100, 2, 34, 41, ...)$$
$$(\text{Zufallsfolge}) = \text{Folge aus 12 und (Zufallsfolge)}$$
$$\texttt{(random-stream)} = \texttt{(stream-cons (random 101) (random-stream))}$$

Die folgende Prozedur leistet das Gewünschte.

```
(define random-stream
  (lambda ()
    (stream-cons (random 101) (random-stream))))
```

In der Definition von `random-stream` sieht man sehr gut, wie notwendig es ist, die Evaluation des zweiten Arguments von `stream-cons` zu unterdrücken.

```
> (print-stream (random-stream) 10)
```

```
80, 88, 55, 43, 33, 31, 14, 95, 37, 89, ...
> (print-stream (random-stream) 10)
27, 57, 13, 0, 44, 56, 3, 17, 12, 94, ...
> (print-stream (random-stream) 10)
94, 42, 49, 10, 29, 10, 100, 56, 12, 43, ...
```

Verwendung einer expliziten Bildungsvorschrift

Eine Zahlenfolge kann auch durch eine einstellige Funktionen $f : \mathcal{N} \mapsto \mathcal{R}$ *explizit* definiert werden. Wenn wir die Folge der natürlichen Zahlen hernehmen und f auf jedes Glied anwenden, erhalten wir die gewünschte Folge.

Mit `stream-map` realisieren wir diese Transformation einer Folge in eine andere. Analog zu `map` nimmt `stream-map` eine einstellige Funktion und einen Stream. Als Ergebnis liefert `stream-map` den oben beschriebenen Resultat-Stream. In folgendem Beispiel ist das die Folge der geraden Zahlen.

```
(define evenstream (stream-map (lambda (x)(* 2 x)) natzahlen))
> (print-stream evenstream 15)
0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, ...
```

Aufgabe 2.13:
Implementieren Sie `stream-map` und stellen Sie `oddstream`, d. h. die Folge der ungeraden Zahlen, bereit.

Operationen mit Streams

Streams sind unendliche Objekte, die als Eingaben von Prozeduren weiterverarbeitet werden können. Enthalten sie Zahlen, wie in unseren Beispielen, so kann man mit ihnen unter Anwendung der bekannten Definitionen rechnen. Wir definieren exemplarisch eine zweistellige Operation `stream+` als gliedweise Addition, die zwei Summandenstreams nimmt und den zugehörigen Summenstream zurückgibt.

```
(define stream+
  (lambda (stm1 stm2)
    (stream-cons
     (+ (stream-first stm1) (stream-first stm2))
     (stream+ (stream-rest stm1) (stream-rest stm2)))))
```

Wir demonstrieren die Anwendung von `stream+` für die Summe der geraden und der ungeraden Zahlen.

```
> (print-stream oddstream 18)
1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31, 33, 35, ...
```

```
> (print-stream evenstream 18)
0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28, 30, 32, 34, ...
> (print-stream (stream+ oddstream evenstream) 18)
1, 5, 9, 13, 17, 21, 25, 29, 33, 37, 41, 45, 49, 53, 57, 61, 65, 69, ...
```

Aufgabe 2.14:
Definieren Sie `stream-` nach dem Vorbild von `stream+`.

Aufgabe 2.15:
Definieren Sie `stream-` unter Verwendung von `stream-map` und `stream+`.

2.5 Der λ-Kalkül

2.5.1 Definition und Evaluation von λ-Ausdrücken

Es ist nun an der Zeit, die theoretische Basis funktionaler Sprachen etwas genauer zu beleuchten. Dabei handelt es sich vor allem um den 1941 von Alonzo Church (1903-1995) entwickelten λ-Kalkül, auf den wir bereits in der Fußnote auf S. 19 kurz hingewiesen haben. Das Motiv zu dessen Erforschung entsprang der Suche nach einer theoretischen Fundierung des *Berechenbarkeitsbegriffs*. Man wollte genau die Klasse von Problemen bestimmen, die „mit dem Computer" gelöst werden können.

Der Ansatz von Church war in dieser Zeit keineswegs der einzige. Später stellte sich heraus, dass diese sehr verschiedenartigen Theorien die gleiche Problemklasse beschrieben. Dies stärkte den Berechenbarkeitsbegriff enorm. Knapp 20 Jahre später stellte sich heraus, dass auf der Grundlage des Lambda-Kalküls eine Programmiersprache, nämlich Lisp, definiert werden kann. John McCarthy (1927-2011) entwickelte das erste lauffähige Lisp-System.

Bis auf einige Modifikationen (aus Effizienzgründen) beruht die Implementation funktionaler Sprachen auf diesem λ-Kalkül. Trotz weitgehender Übereinstimmung sind Kalkül und Implementation nicht identisch, so dass wir beim Entwickeln und Testen von Racket-Prozeduren ggf. bedenken müssen, dass es sich um eine Repräsentation eines (leicht) modifizierten λ-Kalküls handelt.

Church hatte die Idee, die Klasse aller Ausdrücke, sog. λ-Ausdrücke, anzugeben, die man aus gegebenen *Grundbausteinen* konstruieren kann und für die bestimmte *Rechenregeln* gelten.

Wir definieren zunächst die **Syntax von λ-Ausdrücken**:

1. Der Vorrat an Grundbausteinen ist eine (abzählbar) unendliche Menge von *Variablen* $\{x_1, x_2, \ldots\}$. Diese Variablen sind λ-Ausdrücke.

2. Durch *Anwendung* eines λ-Ausdrucks M auf einen λ-Ausdruck N, geschrieben als $(M\ N)$, entsteht ein neuer λ-Ausdruck. Zur Ermittlung des Ergebnisausdrucks sind die oben angekündigten Rechen- oder Reduktionsregeln erforderlich.
3. Schließlich können λ-Ausdrücke auch durch *funktionale Abstraktion* erzeugt werden: Sind x eine Variable und M ein λ-Ausdruck, so ist auch $(\lambda\ (x)\ M)$ ein λ-Ausdruck.

In Racket sind uns Symbole für Variablen wohl bekannt. Ebenso geläufig sind Prozeduranwendungen, wie `(cons (car '(a b)) (cdr '(x y z))) = (a y z)`.
`(lambda (x)(* x x))` ist ein Beispiel einer funktionalen Abstraktion, nämlich für das Quadrieren.

Dem aufmerksamen Leser wird bei der Einführung des λ-Kalküls nicht entgangen sein, dass einige Sprachbestandteile, die wir in Racket zu schätzen gelernt haben, fehlen. Dazu gehört `define`, zur Benennung von Werten, vor allem für Prozeduren. Weiterhin fällt auf, dass der λ-Kalkül ausschließlich die Definition einstelliger Funktionen erlaubt. In Racket gibt es n-stellige ($n \geq 0$) und sogar variabelstellige Prozeduren. Wir werden zeigen, dass dies mit den theoretischen Grundlagen vereinbar ist und es weiterer Zusätze nicht bedarf.

Reduktionsregeln

Im Folgenden befassen wir uns mit den oben angekündigten Rechenregeln, also der Auswertungs- oder Reduktionsstrategie: Welchen Wert beschreibt ein bestimmter λ-Ausdruck? Zu welchem Ausdruck kann ein gegebener Ausdruck reduziert (evaluiert) werden? Wie „rechnet“ man mit λ-Ausdrücken?

β-Regel: Es seien E und M λ-Ausdrücke, wobei M in E nicht gebunden vorkommen darf. x sei eine Variable. Dann beschreibt

$$((\lambda\ (x)\ E)\ M) \overset{\beta}{\Rightarrow} E[M/x]$$

das Resultat der Anwendung von $(\lambda\ (x)\ E)$ auf M als Ergebnis der Ersetzung aller Vorkommen x in E durch M. Man sagt kurz: „E mit M für x“. In der Tat handelt es sich um eine reine Textmanipulation.

Wir wenden die β-Regel auf folgenden λ-Ausdruck an:

$$\underbrace{((\lambda\ (x)\ ((x\ d)\ \underbrace{(\underbrace{(\lambda\ (y)\ (x\ y))}_{f_2}\ a)}_{(f_2\ a)}))}_{f_1}\ b)$$

Die geschweiften Klammern helfen uns, die Struktur des Ausdrucks besser zu erfassen.

Bei der nun folgenden schrittweisen Reduktion arbeiten wir von innen nach außen: Als Erstes wenden wir die mit f_2 bezeichnete Funktion auf a an, indem jedes Vorkommen der gebundenen Variablen y in $(x\ y)$ durch a ersetzt wird. Das Ergebnis ist offensichtlich $(x\ a)$. Im zweiten Schritt schreibt man b für alle x in $((x\ d)(x\ a))$, was zu $((b\ d)(b\ a))$ führt.

$$\begin{aligned} ((\lambda\ (x)\ ((x\ d)((\lambda\ (y)\ (x\ y))\ a)))\ b) &\overset{\beta}{\Rightarrow} ((\lambda\ (x)\ ((x\ d)\ (x\ a)))\ b) \\ &\overset{\beta}{\Rightarrow} ((b\ d)(b\ a)) \end{aligned}$$

Arbeitet man von außen nach innen, so entsteht das gleiche Resultat. Die Substitution beginnt mit b für alle x in $((x\ d)((\lambda\ (y)\ (x\ y))\ a))$.

$$\begin{aligned} ((\lambda\ (x)\ ((x\ d)((\lambda\ (y)\ (x\ y))\ a)))\ b) &\overset{\beta}{\Rightarrow} ((b\ d)((\lambda\ (y)\ (b\ y))\ a)) \\ &\overset{\beta}{\Rightarrow} ((b\ d)(b\ a)) \end{aligned}$$

Die Anwendung der β-Regel bewirkt eine (bedeutungserhaltende) Transformation, engl.: meaning preserving transformation. Man nennt das reasoning about programs by symbolic reduction.

In der β-Regel wurde durch die Forderung „M darf in E nicht gebunden vorkommen" sichergestellt, dass infolge der auszuführende Substitution eine ursprünglich ungebundene Variable nicht in den Gültigkeitsbereich einer λ-Bindung gerät. Dieses Problem wird mit folgendem Beispiel illustriert:

$$((\lambda\ (x)\ (\lambda\ (y)\ (x\ y)))(\mathbf{y}\ w)) \overset{\beta}{\Rightarrow} (\lambda\ (y)\ ((\mathbf{y}\ w)\ y))\ \text{ist falsch!}$$

y ist eine gebundene Variable, hingegen kommt **y** im Vorgabeausdruck frei vor. Nach der (unberechtigten) Anwendung der β-Regel gerät **y** in den Einflussbereich von $\lambda\ (y)$ und wird somit gebunden.

Dies gilt auch für folgenden Ausdruck, dessen Reduktion durch fehlerhaft angewandte β-Regel 16 ergibt:

$$(((\lambda\ (x)\ (\lambda\ (y)\ (*\ x\ y)))\ y)\ 4) \overset{\beta}{\Rightarrow} ((\lambda\ (y)\ (*\ y\ y))\ 4) \overset{\beta}{\Rightarrow} 16.$$

In Racket wird dieser Ausdruck jedoch korrekt evaluiert.

```
(define y 1)
> (((lambda (x)(lambda (y)(* x y))) y) 4)
4
```

Wie soll man sich nun verhalten, wenn die Voraussetzung zur Anwendung der β-Regel nicht erfüllt ist? Da die Wahl der Namen gebundener Variablen in einer Funktionsabstraktion keine Rolle spielt, kann eine geeignete Umbenennung vorgenommen werden, ohne dass die Bedeutung des Ausdrucks dadurch verändert wird. Dies führt uns zur

α-Konvention: Wenn x' in E *nicht frei* vorkommt, gilt:

$$(\lambda\ (x)\ E) \stackrel{\alpha}{\longleftrightarrow} (\lambda\ (x')\ E[x'/x]).$$

Exemplarisch wenden wir auf $(\lambda\ (y)\ (x\ y))$ in $((\lambda\ (x)\ (\lambda\ (y)\ (x\ y)))(y\ w))$ vor der β-Regel die α-Konvention an:

$$\begin{aligned} ((\lambda\ (x)\ (\lambda\ (y)\ (x\ y)))(y\ w)) &\stackrel{\alpha}{\longleftrightarrow} ((\lambda\ (x)\ (\lambda\ (z)\ (x\ z)))(y\ w)) \\ &\stackrel{\beta}{\Rightarrow} (\lambda\ (z)\ ((y\ w)\ z)) \end{aligned}$$

Für unser obiges Racket-Beispiel ergibt sich mit `(define y 1)`:

$$(((\lambda\ (x)\ (\lambda\ (y)\ (*\ x\ y)))\ y)\ 4) \stackrel{\alpha}{\longleftrightarrow} (((\lambda\ (x)\ (\lambda\ (z)\ (*\ x\ z)))\ y)\ 4) \stackrel{\beta}{\Rightarrow} ((\lambda\ (z)\ (*\ y\ z))\ 4) \stackrel{\beta}{\Rightarrow} 4.$$

Aufgabe 2.16:
Schreiben Sie eine Racket-Prozedur, die alle freien Variablen eines gegebenen λ-Ausdrucks als Liste zurückgibt.

Da im λ-Kalkül alle Ausdrücke (Werte) entweder Variablen oder Funktionen von genau einem Argument sind, ähnelt die folgende Regel eher einem Programmiertrick.

η-Umwandlung: Dies ist eine strikte Vereinfachungsregel, mit

$$\begin{aligned} (\lambda\ (z)\ ((\lambda\ (x)\ E)\ z)) &\stackrel{\eta}{\Rightarrow} (\lambda\ (x)\ E) \text{ und} \\ (\lambda\ (x)\ (E\ x)) &\stackrel{\eta}{\Rightarrow} E. \end{aligned}$$

Der oben angegebene Ausdruck $(\lambda\ (z)\ ((\lambda\ (x)\ E)\ z))$ ist eine Funktionsabstraktion, deren Rumpf aus einer Anwendung besteht. Wendet man die β-Regel auf einen Ausdruck dieser Art an, z. B. auf $(\lambda\ (z)\ ((\lambda\ (x)\ (+\ x\ 3))\ z))$, so ergibt sich $(\lambda\ (z)\ (+\ z\ 3))$. Nach der Umbenennung der gebundenen Variablen z zu x erhalten wir erwartungsgemäß $(\lambda\ (x)\ (+\ x\ 3))$.

Ein reduzierbarer Ausdruck (REDEX = reducible expression) wird solange[9] reduziert, bis er keine reduzierbaren Teilausdrücke mehr enthält. Man spricht dann von *Normalform.*

Die schrittweise Reduktion wirft zwei Fragen auf:

1. In welcher Reihenfolge sollten die Ersetzungen (Anwendung der Reduktionsregeln) durchgeführt werden, wenn es im betrachteten Ausdruck mehrere Anwendungsstellen gibt?

[9] Dies gilt nur für den klassischen λ-Kalkül. Implementationen für funktionsorientierte Programmiersprachen weichen davon im Allgemeinen etwas ab.

2. Erhält man in jedem Falle ein Resultat? Mit anderen Worten: Besitzt jeder REDEX eine Normalform?

Wir geben zuerst die Antwort auf die zweite Frage: Es gibt Ausdrücke, die *keine* Normalform besitzen. Das Musterbeispiel ist $\Omega = ((\lambda\ (x)\ (x\ x))(\lambda\ (x)\ (x\ x)))$.

Wegen $((\lambda\ (x)\ (x\ x))(\lambda\ (x)\ (x\ x))) \overset{\beta}{\Rightarrow} ((\lambda\ (x)\ (x\ x))(\lambda\ (x)\ (x\ x)))$ wird Ω durch Anwendung der β-Regel reproduziert. Es entsteht ein REDEX, der wiederum der β-Regel unterworfen werden kann usw. Wir erhalten offenbar keine Normalform für Ω.

Die Antwort auf die erste Frage gibt das CHURCH-ROSSER-Theorem: Falls ein REDEX eine Normalform besitzt, ist gleichgültig, welche Reduktionsreihenfolge man wählt. Auch wenn auf verschiedenen Wegen verschiedene Zwischenresultate entstehen, ist das Endergebnis in jedem Falle das gleiche.

Auch bei der Reduktion des Ausdrucks $((\lambda\ (x)\ ((x\ d)((\lambda\ (y)\ (x\ y))\ a)))\ b)$, s. S. 82, haben wir für zwei Reduktionsrichtungen das gleiche Ergebnis erhalten. Das Theorem beweist jedoch die obige Aussage im Allgemeinen.

Es liegt auf der Hand, dass die beliebige Wahl der Reduktionsreihenfolge vorzügliche Möglichkeiten für *parallele* Verarbeitung bietet.

Aufgabe 2.17:
Verwenden Sie zur Berechnung von $((\lambda\ (v)\ ((\lambda\ (a)\ (a\ v))(\lambda\ (x)\ (x\ v))))\ b)$ unterschiedliche Reduktionsreihenfolgen.

2.5.2 Zur Implementation funktionsorientierter Sprachen

Die Reduktion, genauer die β-Reduktion, ist ein *Termersetzungsverfahren*, bei der eine Funktionsanwendung nach folgender Regel vereinfacht wird: Das Funktionsargument wird unverändert hergenommen und an allen vorkommenden Stellen der gebundenen Variablen der Funktionsabstraktion eingesetzt.

Der λ-Kalkül fordert mit der β-Regel eine sog. *leftmost reduction* oder *normal order reduction*. Der Name kommt daher, dass die Reduktion einer Anwendung stets mit dem (am weitesten links stehenden) Operator beginnt und erst danach der Operand reduziert wird. Die leftmost reduction garantiert eine eindeutige Reduktion einer Funktionsanwendung, falls ein solches Resultat überhaupt existiert.

Im Ausdruck (in Racket-Schreibweise)

$$((\lambda\ (x)\ (+\ x\ x\ x))((\lambda\ (x)\ (*\ x\ x\ x))\ 12345))$$

wird also – gemäß β-Regel – die Anwendung $((\lambda\ (x)\ (*\ x\ x\ x))\ 12345)$ zunächst unverändert für jedes x in $(+\ x\ x\ x)$ eingesetzt. In der Sprechweise der Programmierung entspricht das einer *namensmäßigen Parametervermittlung (call by name)*. Der Ausdruck

wird zu $(+\ ((\lambda\ (x)\ (*\ x\ x\ x))\ 12345)\ ((\lambda\ (x)\ (*\ x\ x\ x))\ 12345)\ ((\lambda\ (x)\ (*\ x\ x\ x))\ 12345))$ reduziert, woraus sich ein großer Nachteil für die praktische Verarbeitung ergibt: Der (rechenaufwendige) Teilausdruck $((\lambda\ (x)\ (*\ x\ x\ x))\ 12345)$ muss – nach der Einsetzung – mehrfach (dreimal) ausgewertet werden. Aus der Sicht der Programmierung bedeutet das einen Effizienzverlust, was die Programmabarbeitung spürbar verlangsamt.

Grundsätzlich gibt es zwei Ideen zur Überwindung dieses Problems. Die erste besteht darin, dass wir die Reduktionsvorschrift für Anwendungen ein wenig modifizieren: Reduziere zuerst das Argument! Der so entstandene Ausdruck wird anschließend der β-Regel unterworfen. Man nennt diese Form *applicative order reduction*, was einer *wertmäßigen Parametervermittlung* (*call by value*) gleichkommt. Die theoretische Fundierung liefert der λ-Kalkül mit Wertaufruf.

Bei gängigen Implementationen funktionsorientierter Sprachen wird nicht nur der Operand (Argument), sondern auch der Operator reduziert, *bevor* die entsprechende Einsetzung stattfindet. Die Implementation erfolgt außerdem so, dass die Reduktion nicht erst beim Erreichen einer Normalform (falls vorhanden) stoppt, sondern schon früher, wenn ein definierter Ergebnistyp vorliegt.

Aufgabe 2.18:
Notieren Sie für $((\lambda\ (v)\ ((\lambda\ (a)\ (a\ v))(\lambda\ (x)\ (x\ v))))\ b)$ die vollständige applicative order reduction.

Das Ergebnis für $((\lambda\ (v)\ ((\lambda\ (a)\ (a\ v))(\lambda\ (x)\ (x\ v))))\ b)$ stimmt mit dem durch normal order reduction gefundenen Resultat überein. Leider ist das nicht immer so, wie das folgende Beispiel zeigt:

$$((\lambda\ (y)\ z)((\lambda\ (x)\ (x\ x))(\lambda\ (x)\ (x\ x))))$$

Bei normal order reduction werden alle Vorkommen der gebundenen Variablen y in z durch den Ausdruck $((\lambda\ (x)\ (x\ x))(\lambda\ (x)\ (x\ x)))$ ersetzt, *ohne* ihn vorher zu evaluieren. Da es in z keine einzige solche Stelle gibt, wird der gesamte Ausdruck zu z reduziert.

Applicative order reduction ist hier nicht erfolgreich: Die Berechnung terminiert nicht, da $((\lambda\ (x)\ (x\ x))(\lambda\ (x)\ (x\ x)))$ zuerst evaluiert wird.

Aufgabe 2.19:
Evaluieren Sie $((\lambda\ (x)\ (x\ x))(\lambda\ (x)\ (x\ x)))$ mit Racket. Konzentrieren Sie sich auf den Stop-Knopf und sichern Sie vor diesem Experiment wichtige Daten!

Offensichtlich liefert die effizientere applicative-order-Strategie nicht immer ein Ergebnis, auch wenn es existiert (und mit normal order reduction gefunden wird). Für die bekannten Racket-Implementationen hat man sich dennoch entschieden, die applicative order reduction zu verwenden, wohl wissend, dass sie eben für sehr seltene Beispiele nicht funktioniert.

Die zweite Möglichkeit, dem Problem der Mehrfachevaluation zu begegnen, ist die verzögerte Evaluation, *lazy evaluation*, mit der wir uns in Abschn. 2.4.3 beschäftigt haben. Man vermeidet wiederholte Berechnungen des Arguments, indem man den zugehörigen Ausdruck durch den (einmal) berechneten Wert überschreibt. Man weiß, dass diese *call-by-need*-Vermittlung wenigstens theoretisch genauso effizient ist wie call by value und die gewünschte Reduktion leistet.

2.5.3 Currying

Wir haben gesehen, dass der λ-Kalkül die Definition *einstelliger* Funktionen erlaubt und keine anderen. In Racket ist bekanntlich viel mehr möglich. Wie kann man aus dem λ-Kalkül die Möglichkeit ableiten, mehrstellige Funktionen zu definieren? Oder anders gefragt: Wie kann man die Definition mehrstelliger Funktionen auf einstellige zurückführen?

Die Antwort darauf haben zwei Herren, nämlich Moses Schönfinkel (1889-1942) und Haskell Curry (1900-1982), unabhängig voneinander gefunden.

Ausgangspunkt ist die in Übung 2.3 auf S. 58 entwickelte zweistellige Prozedur `compose`. Der Aufruf `((compose add1 sqrt) 9)` liefert `4`.

Eine curryfizierte Version dieser Prozedur erhält man entweder durch die Anwendung der in Racket vorhandenen Prozedur `curry`, wie in `(curry compose)`, oder indem man die Curryfizierung „manuell“ vornimmt:

```
(define compose-curry
  (lambda (f)
    (lambda (g)
      (lambda (x)
        (f (g x))))))

> (((compose-curry add1) sqrt) 9)
4
```

Im Aufrufbeispiel sieht man, dass die `add1`-Prozedur nach Anwendung der `sqrt`-Prozedur auf die Zahl `9` angewandt wird. Bis auf `define`) sind ausschließlich einstellige Prozeduren in Aktion.

Vielleicht weil die Bezeichnungen „Schönfinkeln“ und (engl.) „Schönfinkel-ing“ für diese Funktionstransformation irgendwie unanständig klingen, haben sich Currying und manchmal auch Curryfication, deutsch: *Curryfizieren*, durchgesetzt.

Allgemein wird durch Currying eine n-stellige Funktion ($n > 1$) der Form

$$f : A_1 \times A_2 \times \ldots \times A_n \mapsto B$$

überführt in eine zugehörige, einstellige Funktion (ein Funktional) der Gestalt

$$f' : A_1 \mapsto (A_2 \mapsto (\ldots (A_n \mapsto B) \ldots)).$$

Folglich wird durch die Verwendung von Prozeduren mit mehreren Parametern in Racket der λ-Kalkül als theoretische Basis nicht verlassen.

2.5.4 Y combinator

Nun wollen wir noch die letzte „Unstimmigkeit" zwischen Racket und dem zugrunde liegenden λ-Kalkül aufklären und fragen, ob die Verwendung von `define` (und damit von `set!`) überhaupt notwendig ist? Im λ-Kalkül kommen Wertzuweisungen, z. B. mit `define`, nicht vor.

Natürlich besteht nicht für alle Prozeduren die Notwendigkeit der Benennung. Beispielsweise kann das Quadrieren ebenso mit einer anonymen Prozedur stattfinden.

```
> ((lambda (n) (* n n)) 8)
64
```

Aber wie soll es gelingen, *rekursive* Prozeduren wie etwa

```
(define fak
  (lambda (n)
    (if (= n 0)
      1
      (* n (fak (- n 1))))))

> (fak 5)
120
```

ohne deren Benennung zu definieren?

In der Tat kann man eine Funktion Y_{normal} angeben, die wie ein „Rekursivmacher" für eine als λ-Ausdruck gegebene Funktion f arbeitet. In Y_{normal} treffen wir wieder auf Ω, s. S. 84.

$$Y_{normal} = (\lambda\ (f)((\lambda\ (x)\ (f\ (x\ x)))(\lambda\ (x)\ (f\ (x\ x)))))$$

Von einer solchen Funktion Y wird gefordert, dass sie die Fixpunkt-Eigenschaft

$$(Y\ f)\ =\ (f\ (Y\ f))$$

erfüllt. Wir rechnen dies (mit normal order reduction) nach:

$$\begin{aligned}
(Y\ f) &= ((\lambda\ (f)((\lambda\ (x)\ (f\ (x\ x)))(\lambda\ (x)\ (f\ (x\ x)))))\ f) \\
&\overset{\beta}{\Rightarrow} ((\lambda\ (x)\ (f\ (x\ x)))(\lambda\ (x)\ (f\ (x\ x)))) \\
&\overset{\beta}{\Rightarrow} (f\ ((\lambda\ (x)\ (f\ (x\ x)))(\lambda\ (x)\ (f\ (x\ x))))) \\
&= (f\ (Y\ f))
\end{aligned}$$

Die Fixpunkteigenschaft ist offensichtlich genau das, was wir brauchen, um eine Funktion „rekursiv zu machen“. Für die Fakultätsfunktion könnte das folgender Racket-Aufruf leisten.

```
(Y-normal
  (lambda (fak)
    (lambda (n)
      (if (= n 0)
        1
        (* n (fak (- n 1)))))))
```

Leider ist das nicht der Fall. Da Racket-Ausdrücke mit applicative order reduction reduziert werden, liefert der zugehörige Aufruf

```
(define Y-normal
  (lambda (f)
    ((lambda (x) (f (x x)))
     (lambda (x) (f (x x))))))
```

kein Ergebnis. Man braucht also wieder den Stop-Knopf!

Aufgabe 2.20:
Berechnen Sie diese Anwendung von Y_{normal} auf dem Papier. Benutzen Sie normal order reduction.

Für Racket benötigt man einen sog. *applicative order Y combinator*,

```
(define Y
  (lambda (f)
    ((lambda (x)
       (f (lambda (z) ((x x) z))))
     (lambda (x)
       (f (lambda (z) ((x x) z)))))))
```

der die gewünschte Wirkung als „Rekursivmacher“ erzielt.

Um den Aufruf besser lesbar zu halten, verwenden wir das kurze Y und zeigen weiter unten, dass dessen Definition (mit `define`) prinzipiell nicht notwendig ist.

```
> (Y
    (lambda (fak)
      (lambda (n)
        (if (= n 0)
          1
          (* n (fak (- n 1)))))))
#<procedure>
```

Das Ergebnis ist erwartungsgemäß eine Prozedur, die, auf 5 angewandt, $5! = 120$ liefert.

```
> ((Y
     (lambda (fak)
       (lambda (n)
         (if (= n 0)
           1
           (* n (fak (- n 1)))))))
   5)
120
```

Um abschließend zu zeigen, dass man wirklich ohne `define` auskommt, ersetzen wir Y durch den entsprechenden λ-Ausdruck.

```
> (((lambda (f)
    ((lambda (x)
       (f (lambda (z) ((x x) z))))
      (lambda (x)
       (f (lambda (z) ((x x) z))))))
  (lambda (fak)
    (lambda (n)
      (if (= n 0)
          1
          (* n (fak (- n 1)))))))
 5)
120
```

Aufgabe 2.21:
Versuchen Sie, den Aufruf zur Erzeugung der Racket-Prozedur für die rekursiv definierte Fakultätsfunktion mit Bleistift und Papier nachzuvollziehen.

Damit haben wir uns davon überzeugen können, dass man mit den Mitteln des λ-Kalküls mehrstellige und rekursive Funktionen definieren und anwenden kann. Doch wo sind die Konstanten (Zahlen, Wahrheitswerte, ...) und die vielen anderen Sprachelemente, die wir in Racket – auf sauberer theoretischer Basis – verwenden? Kaum zu glauben, aber auch diese können im λ-Kalkül konstruiert werden. Dies ist allerdings Gegenstand der Berechenbarkeitstheorie.

2.6 Umgebungsmodell

2.6.1 Umgebungsbegriff

In den vorangehenden Abschnitten haben wir genau beschrieben, wie beliebig komplexe Racket-Ausdrücke ausgewertet werden. Dabei sind wir davon ausgegangen, dass das Racket-System sämtliche Variablen, die zur Benennung von Werten verwendet werden,

kennt. Die Frage nach der Art und Weise, wie die vom Nutzer definierten Variablen-Bindungen eingerichtet bzw. zur Auswertung von Ausdrücken bereitgestellt werden, haben wir noch nicht gestellt. Das soll im Folgenden nachgeholt werden.

Grundsätzlich kann man davon ausgehen, dass zahlreiche globale Variablen nach dem Start des Systems existieren. Abb. 2.1 zeigt einen winzigen Ausschnitt aus dieser sog. *globalen Umgebung*. Einige typische Repräsentanten, wie Ziffern, `#t`, `#f`, die Variable `pi`, Operationssymbole, wie `+`, und Zeichen, wie `#\a`, sind angegeben. Es sind die Grundbausteine, aus denen komplexere Objekte, wie z.B. ganze Zahlen, gebildet werden können.

Zahlen, Zeichen, ..., Variablennamen	Wert
`3`	`3`
`pi`	`3.141592653589793`
`#t`	`#t`
`+`	`<procedure +>`
`#\a`	`#\a`

Abbildung 2.1: globale Umgebung (Ausschnitt)

Wie aus Abb. 2.1 hervorgeht, passt es sehr gut, sich diese Name-Wert-Liste als Tabelle vorzustellen.

Bei erweiterbaren Sprachen, wie eben Racket, ist die Entscheidung eher subjektiv, welcher Sprachumfang als *initial* angesehen werden soll. Beim Systemstart und während des Dialogs mit dem Nutzer können die gewünschten Variablenbindungen bereitgestellt werden.

Eine Tabelle für Bindungen, wie sie etwa nach dem Systemstart eingerichtet wird, nennen wir *Rahmen* (frame). Ein Rahmen kann beliebig viele Bindungen (Tabellenzeilen) enthalten. Jedes Racket-Symbol ist *höchstens einmal* als Variablenname in dieser Tabelle enthalten.

Eine Folge (Liste) von Rahmen wird als *Umgebung* (environment) bezeichnet. In Abb. 2.1 besteht die globale Umgebung aus genau einem Rahmen, in dem die initialen Variablen gebunden sind. Da es nur einen Rahmen gibt, stellt dieser auch die *globale Systemumgebung* dar.

Im Allgemeinen ergibt sich die globale Umgebung aus der Rahmenfolge für nutzer- und vordefinierte Variablenbindungen

- nach dem Systemstart
- nach dem Laden vordefinierter Variablen und
- nach jeder Definition einer Variablen mit `define`[10].

[10] s. Abschn. 2.6.3

Abb. 2.2 auf S. 92 illustriert vier Umgebungen $E_1 = F_1 \to F_3 \to F_4$, $E_2 = F_2 \to F_3 \to F_4$, $E_3 = F_3 \to F_4$ und $E_4 = F_4$. Dabei handelt es sich um eine fiktive Konstellation, die praktisch so nicht vorkommen kann. Die darin eingetragenen Umgebungen können gut diskutiert werden.

Der von oben auf den jeweiligen Rahmen verweisende Pfeil bestimmt den Beginn der Umgebung. Sie endet bei dem Rahmen, der keinen Nachfolger besitzt.

2.6.2 Bestimmung eines Variablenwerts

Wie findet man den Wert einer Variablen in diesem Umgebungsmodell? Für jeden Racket-Ausdruck steht fest, in welcher Umgebung er ausgewertet wird. Dies führt zu der aufgeworfenen Frage nach den einzelnen Werten sämtlicher Variablen im betrachteten Ausdruck.

Das Verfahren zur Evaluation einer Variablen x in einer Umgebung E lässt sich folgendermaßen beschreiben:

1. Suche x im ersten, zweiten, dritten, ... Rahmen von E.
2. Beende die Suche mit Erfolg bei der ersten Fundstelle von x oder erfolglos am Ende von E.
3. Gib im Erfolgsfall den im Rahmen angegebenen Wert von x zurück bzw. – bei Misserfolg – eine Fehlermeldung: `reference to undefined identifier: x` bzw. `variable x is not bound`.

Wir wenden das Verfahren auf den folgenden mit A bezeichneten Ausdruck

```
(+ (* a y) z)
```

an und legen das in Abb. 2.2 auf S. 92 dargestellte Umgebungsmodell zugrunde.

In E_1 hat A keinen Wert, denn für das Symbol `a` gibt es keine Bindung, weder in F_1 noch in F_3 und F_4. Der Wert (`17`) von `y` wird in F_1 gefunden, der von `z` ist ebenfalls `17` und findet sich in F_3.

In E_2 hat A den Wert `89` = `(+ (* 2 43) 3)`. `a` findet man in F_2, `y` in F_3 und `z` (erstmalig) in F_2. Man sagt: Die Bindung von `z` in F_3 wird verschattet. Die Werte für `+` und `*` werden in E_4 gefunden.

2.6.3 Herstellung einer Variablenbindung

Eine Variablen-Definition mittels `define` sorgt dafür, dass die zugehörige Bindung als Name-Wert-Paar in einen (bestimmten) Rahmen dauerhaft eingetragen wird. Da jeder Bezeichner (identifier) als Variablenname in einem Rahmen[11] nur höchstens einmal vor-

[11] Hier steht Rahmen, nicht etwa „Umgebung“!

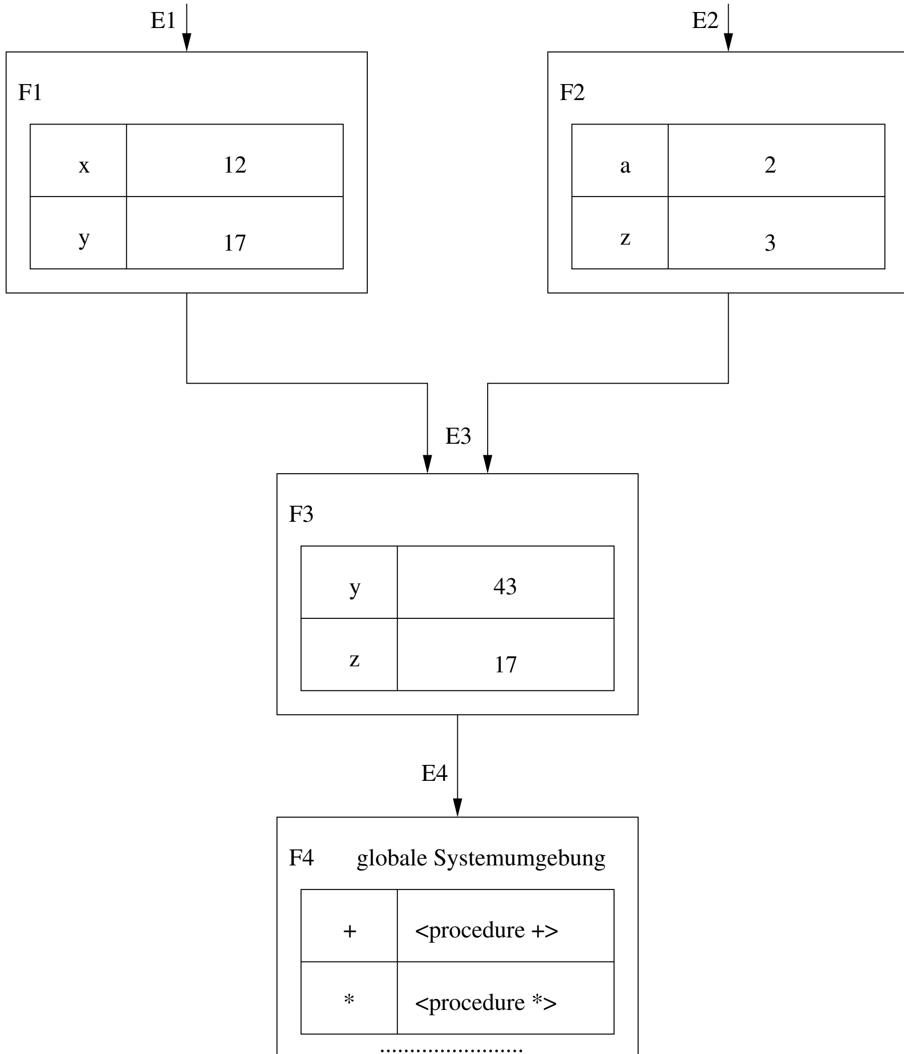

Abbildung 2.2: Umgebungsmodell mit den Umgebungen E_1, E_2, E_3 und E_4

kommen darf, spielt es keine Rolle, in welcher Tabellenzeile die Bindung steht.

Mehrfache Definitionen ein und derselben Variablen werden im DrRacket-System (Definitionsfenster) nicht zugelassen (duplicate definition for identifier). Eine Wertveränderung einer eingerichteten Variablenbindung mittels `set!` ist hingegen innerhalb des betreffenden Moduls (z. B. im Definitionsfenster) möglich.

Nach dem folgenden Dialog steht die Variable `hallo` im Rahmen für nutzerdefinierte Variablen F_2 und damit in der globalen Umgebung E_1, s. Abb. 2.3.

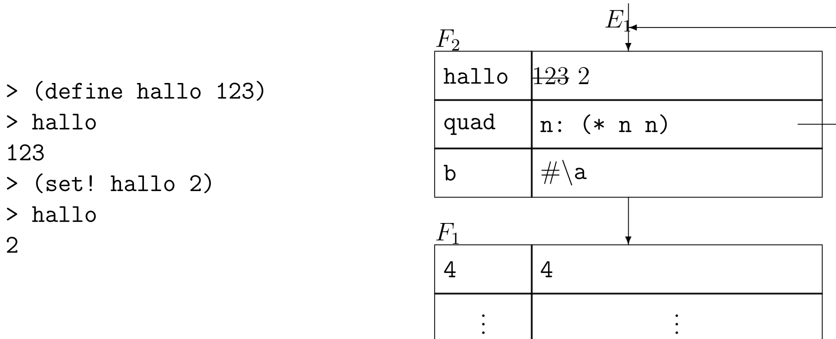

Abbildung 2.3: globale Umgebung (E_1): nutzerdef. (F_2) und eingebaute Variablen (F_1)

Der in Abb. 2.3 dargestellte Rahmen F_2 enthält – neben der für `hallo` – zwei weitere Bindungen. Der Wert von `quad` ist eine Prozedur: das Resultat der Evaluation des λ-Ausdrucks `(lambda (n)(* n n))`.

Bei der Evaluation der `define`-Sonderform wird bekanntlich das zweite Argument ausgewertet. Der in der Tabelle gespeicherte Wert ist das Ergebnis dieser umgebungsbezogenen Evaluation des betrachteten Ausdrucks. Zahlen, Zeichen, Zeichenketten, die beiden Wahrheitswerte und die leere Liste evaluieren zu sich selbst. Welchen Wert liefert die Evaluation von `(lambda (n)(* n n))` in E_1? Das Ergebnis ist eine Prozedur, genauer: eine sog. *closure*, die drei Informationen enthält:

1. den/die formalen Parameter,
2. den Prozedurkörper (Funktionsabstraktion) und
3. die Umgebung, in der der Prozedurkörper (bei Aufruf dieser Prozedur) auszuwerten ist.

Der Wert von `quad` in Abb. 2.3 ist also eine closure mit genau einem Parameter `n` und dem Prozedurkörper `(* n n)`, der in der Umgebung E_1 ausgewertet werden muss. Die zugehörige Umgebung wird durch einen Pfeil, der hier auf E_1 zeigt, festgelegt. Dies ist genau die Umgebung, in der die Prozedur `quad` definiert wurde. Es mag zunächst abwegig

erscheinen, dass dieser Pfeil auf eine andere als die globale Definitionsumgebung zeigen könnte. Weiter unten werden wir sehen, dass dies nicht nur möglich, sondern für das Verständnis der Auswertung von Ausdrücken von grundlegender Bedeutung ist.

Abschließend werfen wir noch einmal einen Blick auf Abb. 2.2 auf S. 92. Wir möchten klären, wie die dort angegebenen Umgebungen entstanden sein könnten. F_3 erhält man nach dem Systemstart durch:

```
(define z 17)
(define y 43)
```

Der Rahmen F_3 und damit die Umgebung E_3 bleiben solange bestehen, bis die Arbeit mit dem Racket-System beendet wird. Generell muss nicht jeder Rahmen diese Eigenschaft besitzen: Wenn wir annehmen, dass die Gültigkeiten von F_1 und F_2 nur auf die Evaluation eines Ausdrucks beschränkt sind, könnten sie beispielsweise mit den folgenden `let`-Ausdrücken hergestellt worden sein.

```
> (let ([x 12][y 17])
    (printf "x=~a y=~a z=~a" x y z))
x=12 y=17 z=17

> (let ([a 2][z 3])
    (printf "a=~a y=~a z=~a" a y z))
a=2 y=43 z=3
```

Nach Evaluation der angegebenen Ausdrücke sind die beiden Rahmen wieder verschwunden.

2.6.4 Prozeduranwendung im Umgebungsmodell

Mit Hilfe des Umgebungsmodells können wir nun analysieren, wie die Evaluation eines Prozeduraufrufes abläuft. Dabei greifen wir auf die Beispielumgebung aus Abb. 2.3 aus Abschn. 2.6.3 zurück. Im Rahmen F_2 wurde durch Evaluation der Prozedurdefinition

```
(define quad
  (lambda (n)
    (* n n)))
```

die Variable `quad` eingetragen und an die closure gebunden, die bei der Auswertung des angegebenen λ-Ausdrucks in der Definitionsumgebung zurückgegeben wird.

Der Aufruf

```
> (quad 2)
```

löst nun folgende Verarbeitung aus:

1. Suche den Wert von `quad` und den von `2`.
   Letzterer ist `2` und findet sich in F_1. Als Wert von `quad` entnehmen wir in F_2 eine closure, die aus den folgenden drei Komponenten besteht:
   - Parameter: `n`
   - Prozedurkörper: `(* n n)`
   - Zugehörige Umgebung: E_1

   Diese werden nun für die Auswertung des o. g. Ausdrucks benötigt.
2. Es wird ein neuer Rahmen F_3 gebildet. In F_3 wird der Parameter der Prozedur als Variable `n` (erste Information aus der closure) mit dem Wert des Arguments beim Aufruf, also `2`, eintragen. Dieser Rahmen wird der zu `quad` gehörenden Umgebung E_1 (dritte Information aus der closure) *vorangestellt*.
   Die neue Umgebung $E_2 = F_3 \to E_1$ ist nun die, in der der Prozedurrumpf (zweite Information aus der closure) evaluiert wird, s. Abb. 2.4.

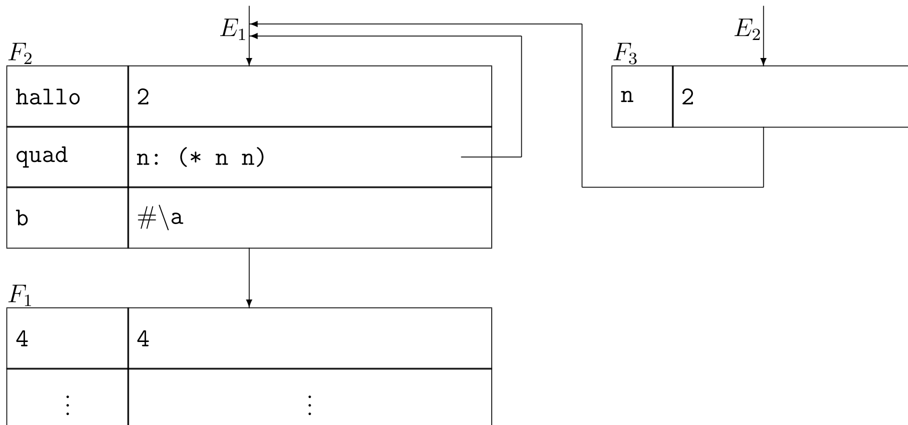

Abbildung 2.4: Umgebungsmodell zur Auswertung von `(quad 2)`

3. Der Ausdruck `(* n n)` wird in E_2 ausgewertet. Das Ergebnis ist offensichtlich `4`.
4. Mit Abschluss der Evaluation der Prozeduranwendung wird die ursprüngliche Umgebung E_1 als die gültige globale Umgebung wieder hergestellt. D. h. der Rahmen F_3 steht danach nicht mehr zur Verfügung.

Aufgabe 2.22:
Ergänzen Sie das Umgebungsmodell in Abb. 2.4 um den Eintrag, der durch die Auswertung von `(define res (quad 2))` hervorgerufen wird. Machen Sie sich dabei klar, in welcher Umgebung der jeweils betrachtete Ausdruck evaluiert wird.

Aufgabe 2.23:
Analysieren Sie die Definition von `hyp` und den Aufruf `(hyp 2 3)` mit dem Umgebungsmodell.

```
(define hyp
  (lambda (kat1 kat2)
    (sqrt (+ (* kat1 kat1) (* kat2 kat2)))))
```

Aufgabe 2.24:
Machen Sie sich mit dem Umgebungsmodell klar, weshalb die Prozedur `quad` auch dann korrekt arbeitet, wenn man den formalen Parameter `n` in `quad` umbenennt, so dass sich die folgende Definition ergibt.

```
(define quad
  (lambda (quad)
    (* quad quad)))
```

Aufgabe 2.25:
Warum führt der Aufruf `(fak 3)` nach Umbenennung des Parameters `n` zu `fak` in der Definition

```
(define fak
  (lambda (n)
    (if (= n 0)
      1
      (* n (fak (- n 1))))))
```

zum Abbruch mit einer Fehlermeldung wie `application: not procedure; expected a procedure that can be applied to arguments given: 3 arguments...: 2`.
Stellen Sie den Mauszeiger im Definitionsfenster von DrRacket auf den formalen Parameter `n` und nutzen Sie die DrRacket-Unterstützung bei der Umbenennung von Variablen. Wenn Sie versuchen, `n` in `fak` umzubenennen, erhalten Sie die Warnung *The new name you have chosen, "fak", conflicts with an already established name in this scope.*

2.6.5 Prozeduren mit erweiterter Definitionsumgebung

Im Abschn. 2.2.2 über Prozeduren höherer Ordnung haben wir Prozeduren kennengelernt, die eine Prozedur zurückgeben können. Eine solche ist beispielsweise `k-add`, die eine natürliche Zahl k als Eingabe erwartet und eine einstellige Prozedur zurückgibt, die ihrerseits zu einer eingegebenen Zahl n die Summe $n + k$ ausgibt.

```
(define k-add
  (lambda (k)
    (lambda (n)
      (+ k n))))
```

Aus Abschn. 2.6.4 wissen wir, dass als erstes die Variable `k-add` im Rahmen F_2 eingetragen wird. Ihr Wert ist die closure mit den in Abb. 2.5 angegebenen drei Komponenten.

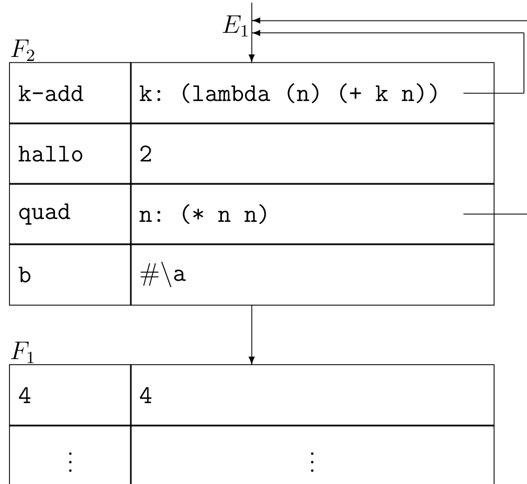

Abbildung 2.5: Umgebungsmodell nach der Definition von `k-add`

Bis hierher gibt es also nichts Neues. Interessant ist nun der Aufruf

```
> (define 3-add (k-add 3))
```

Die Anwendung `(k-add 3)` wird folgendermaßen evaluiert.

1. Erzeuge einen Rahmen F_3 mit $k = 3$ und bilde die Umgebung $E_2 = F_3 \to E_1$. (E_1 deshalb, weil dies die zu `k-add` gehörige Umgebung ist.)
2. Evaluiere den Körper von `k-add`, also `(lambda (n)(+ k n))`, in E_2. Das Ergebnis ist eine closure, deren Definitionsumgebung nicht E_1, sondern E_2 ist.
3. Trage die Variable `3-add` mit diesem Wert in F_2 ein.

Das zugehörige Umgebungsmodell zeigt Abb. 2.6.

Nun wenden wir uns einem Aufrufbeispiel für `3-add` zu.

```
> (3-add 7)
10
```

Die so definierte Prozedur `3-add` arbeitet wie ein „3-Addierer“.

Das in Abb. 2.6 dargestellte Umgebungsmodell wird temporär um die Umgebung $E_3 = F_4 \to E_2$ erweitert, s. Abb. 2.7. In F_4 wird $n = 7$ eingetragen. Danach wird der Ausdruck `(+ k n)` in E_3 evaluiert.

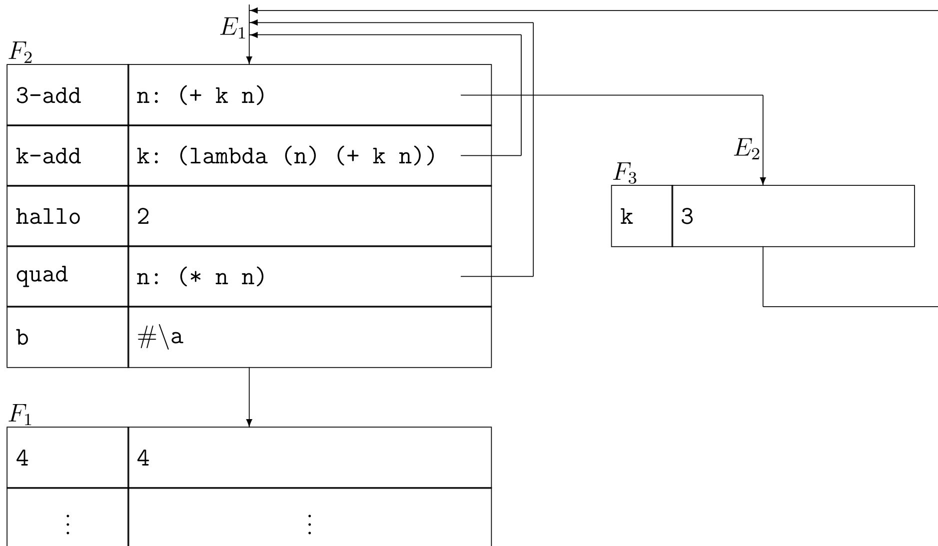

Abbildung 2.6: Umgebungsmodell nach der Definition von `3-add`

Aufgabe 2.26:
Ergänzen Sie das Umgebungsmodell in Abb. 2.6 um eine Prozedur `10-add`, also um einen 10-Addierer, und spielen Sie danach den Aufruf `(10-add 4)` durch.

Die Bearbeitung von Aufg. 2.26 wird dringend empfohlen. Dabei wird sich herausstellen, dass es zwei voneinander völlig unabhängige Rahmen mit $k = 3$ für `3-add` und $k = 10$ für `10-add` gibt. Sie bilden jeweils das Anfangsglied der Umgebungen, die in der entsprechenden closure gespeichert werden. Die k's werden also völlig separat verwaltet und könnten nur von der jeweils zugehörigen Prozedur wertmäßig verändert werden.

Ob `3-add` und `10-add` physisch identischen Code für den Prozedurkörper verwenden, hängt von den Details der Racket-Implementierung ab.

Aufgabe 2.27:
Analysieren Sie den Aufruf `(g 9)` $= 36$, mit folgender Definition von `g` mit dem Umgebungsmodell.

```
(define g
  (lambda (m)
    ((lambda (a b)
       (set! a (+ a 1))
       (+ (* a b b) m))
     2
     3)))
```

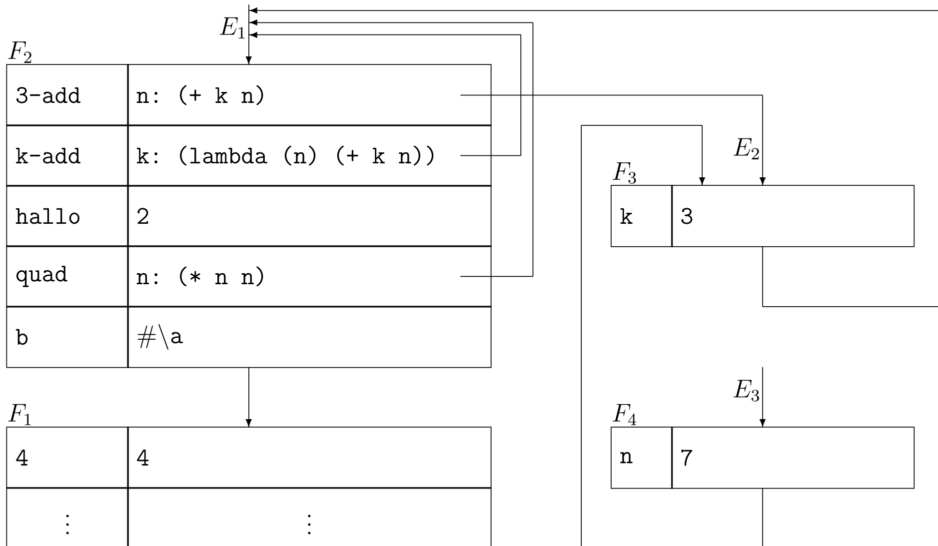

Abbildung 2.7: Umgebungsmodell unmittelbar nach dem Aufruf von `(3-add 7)`

In diesem `define`-Ausdruck kommen eine Prozedurdefinition und eine Prozeduranwendung vor. Beachten Sie, dass die Anwendung erst bei obigem Aufruf und nicht etwa bei der Definition der Prozedur ausgeführt wird.

2.6.6 Prozeduren mit lokalem Zustand

Wir betrachten die beiden Prozeduren `f1` und `f2` mit folgenden Definitionen.

```
(define f1
  (lambda (n)
    (let ([a 2][b 3])
      (set! a (+ a 1))
      (+ (* a n n) b))))
```

```
(define f2
  (let ([a 2][b 3])
    (lambda (n)
      (set! a (+ a 1))
      (+ (* a n n) b))))
```

Warum liefert `(f2 4)` bei wiederholtem Aufruf unterschiedliche Resultate, `(f1 4)` hingegen immer `51`?

Aus der Sicht guten Programmierstils sind die beiden Definitionen keinesfalls vorbildlich! Sie wurden so gestaltet, um ganz bestimmte Effekte studieren zu können.

Zuerst lösen wir den mit `let` eingebrachten syntaktischen Zucker auf. Dabei ist zu beachten, dass aus `(let ([a 2][b 3]) ...)` eine *Anwendung* (nicht etwa eine Prozedur oder closure) entsteht, nämlich `((lambda (a b) ...) 2 3)`. Die allgemeine Regel heißt:

```
(let ([v1 e1][v2 e2] ... [vn en]) E)
   => ((lambda (v1 v2 ... vn) E) e1 e2 ... en)
```

Die Verwendung von `let` bzw. `letrec` für lokale Gültigkeitsbereiche ist oft sehr bequem und wurde bereits auf S. 23 beschrieben.

Nach dem Ersetzen von `let` haben wir es also mit den folgenden Definitionen zu tun.

```
(define f1
  (lambda (n)
    ((lambda (a b)
       (set! a (+ a 1))
       (+ (* a n n) b))
     2
     3)))
```

```
(define f2
  ((lambda (a b)
     (lambda (n)
       (set! a (+ a 1))
       (+ (* a n n) b)))
   2
   3))
```

Analyse von `f1`

Wir beginnen unsere Analyse bei `f1`. Nach der Definition ergibt sich die in Abb. 2.8 skizzierte Umgebung E_1. Der Wert von `f1` ist also eine closure.

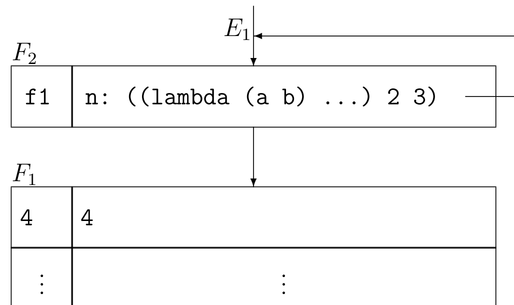

Abbildung 2.8: globale Umgebung (E_1): nutzerdef. (F_2) und eingebaute Variablen (F_1)

Bei Aufruf `(f1 4)` wird der Rumpf der Prozedur, also die Anwendung

```
((lambda (a b)
   (set! a (+ a 1))
   (+ (* a n n) b))
 2
 3)
```

in der Umgebung E_2 ausgewertet, nachdem der Rahmen F_3 für die Bindung von `n` eingerichtet wurde. Danach wird F_4 für die Bindungen von `a` und `b` aufgesetzt und die Umgebung E_3 gebildet. Das zugehörige Umgebungsmodell zeigt Abb. 2.9.

Die beiden Ausdrücke

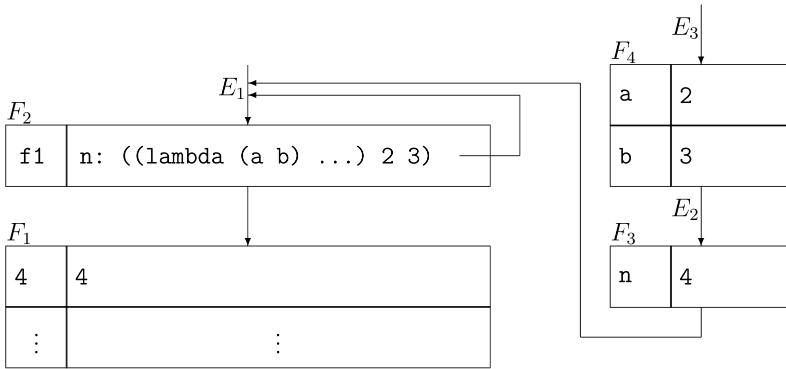

Abbildung 2.9: Umgebungsmodell für den Aufruf `(f1 4)`

```
(set! a (+ a 1))
(+ (* a n n) b))
```

werden nun in Umgebung E_3 evaluiert. Dabei wird der Wert von `a` zuerst inkrementiert. Das Ergebnis ist `(+ (* 3 4 4) 3)`=51. Danach nimmt das Umgebungsmodell wieder die in Abb. 2.8 dargestellte Gestalt an.

Wiederholt man den Aufruf unverändert, wird der beschriebene Ablauf reproduziert und das gleiche Resultat ausgegeben.

Aufgabe 2.28:
Erklären Sie, weshalb `(f1 4)` die Zahl 30 ausgibt, wenn man alle in der Definition von `f1` vorkommenden `n` in `a` umbenennt. Scheuen Sie nicht die Mühe, ein dementsprechendes Umgebungsmodell zu skizzieren.

Analyse von `f2`

Die Analyse der Definition von `f2` führt zu einem ganz anderen Umgebungsmodell, wie man aus Abb. 2.10 ersehen kann.

Zuerst wird die Anwendung `((lambda (a b) ...) 2 3)` in der globalen Umgebung E_1 ausgeführt. Dies erzeugt Umgebung E_2 mit dem Rahmen F_3 für die Bindungen von `a` und `b`. Der λ-Ausdruck `(lambda (n) ...)` wird in E_2 evaluiert. Das Ergebnis ist eine closure, die als Wert von `f2` in F_2 eingetragen wird. Die zur closure gehörende Umgebung ist $E_2 = F_3 \to F_2 \to F_1$.

Der Aufruf `(f2 4)` wird natürlich in der globalen Umgebung E_1 ausgewertet. Abb. 2.11 zeigt das zu diesem Aufruf gehörende Umgebungsmodell.

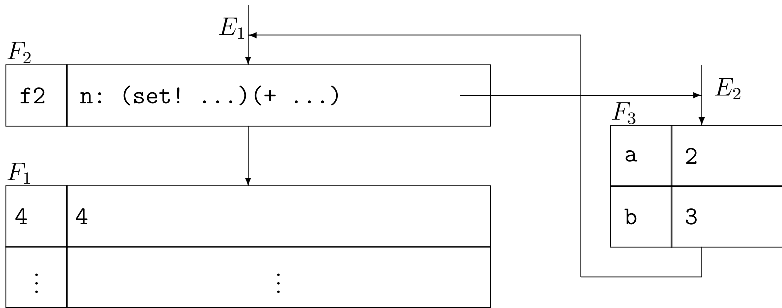

Abbildung 2.10: globale Umgebung (E_1) nach Definition von `f2`

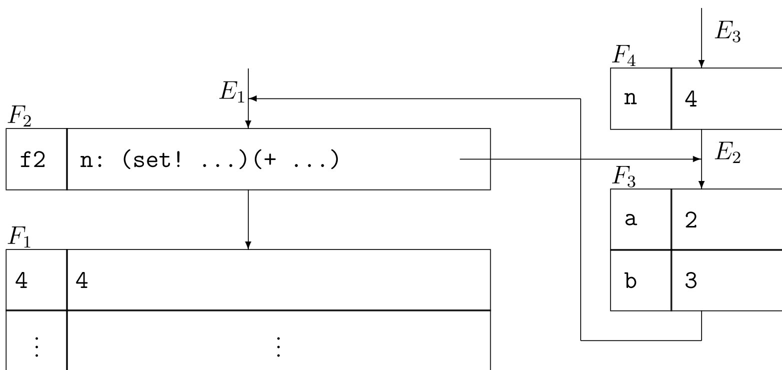

Abbildung 2.11: Umgebungsmodell beim *ersten* Aufruf von `(f2 4)`

In F_2 bzw. F_1 findet man die Werte von `f2` bzw. `4`. Für den Parameter `n` von `f2` richten wir zeitweilig den Rahmen F_4, mit $n = 4$, ein.

Da die zu `f2` gehörende Umgebung E_2 ist, zeigt der F_4 verlassende Pfeil auf E_2. Der Körper von `f2` wird nun in $E_3 = F_4 \to F_3 \to F_2 \to F_1$ evaluiert. Das Ergebnis ist 51.

Aufgrund der Wertveränderung von `a` nach `(set! a (+ a 1))` wird der entsprechende Eintrag in F_3 modifiziert. Darin unterscheidet sich das Umgebungsmodell nach Abschluss der Evaluation von `(f2 4)` gegenüber dem in Abb. 2.10.

Der Rahmen F_4 und damit auch die Umgebung E_3 gehen verloren. Beim nächsten Aufruf `(f2 4)` werden sie wieder aufgebaut. Abb. 2.12 zeigt die Umgebung E_3 (mit dem neuen Wert für `a` in F_3), in der `(f2 4)` beim zweiten Aufruf ausgewertet wird.

Das Ergebnis ist `(+ (* 4 4 4) 3)` = 67. Weitere Aufrufe (ohne vorherigen Neudefinition von `f2`) liefern 83, 99 und 115.

Auf diese Weise lassen sich Zustände, die bestimmte Eigenschaften eines Objektes kap-

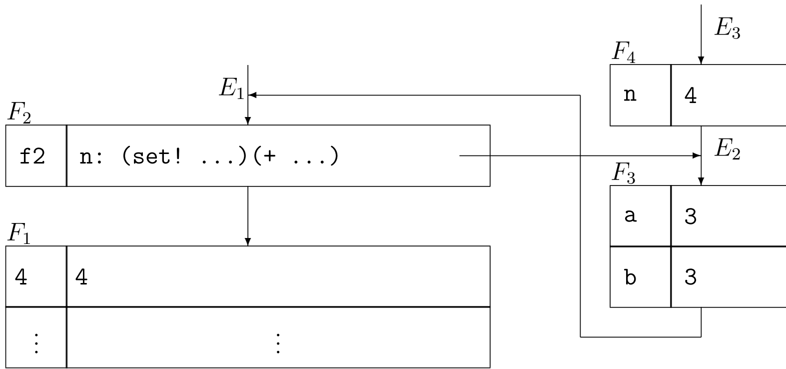

Abbildung 2.12: Umgebungsmodell beim *zweiten* Aufruf von `(f2 4)`

seln, mit Funktionen modellieren.

Aufgabe 2.29:
Nehmen Sie in der Definition von `f2` die (zugegeben unsinnige) Umbenennung aller vorkommenden `n` zu `a` vor und erklären Sie, weshalb danach auch bei sämtlichen Aufruf-Wiederholungen die Zahl 128 als Wert von `(f2 4)` ausgegeben wird.

http://www.springer.com/978-3-658-14133-2

Programmierparadigmen
Eine Einführung auf der Grundlage von Racket
Wagenknecht, C.
2016, X, 244 S. 41 Abb., Softcover
ISBN: 978-3-658-14133-2