8 Programmieren mit Akkumulatoren

Bei den rekursiven Prozeduren der vergangenen Kapitel war der Wert eines rekursiven Aufrufs stets unabhängig vom Kontext: Die Fakultät von 4 „wußte nicht“, daß sie später noch mit 5 multipliziert wird, die Summe der Zahlen von 1 bis 4 „wußte nicht“, daß später noch 5 dazuaddiert wird, etc. Manche Probleme sind aber so formuliert, daß bei der Berechnung ein Zwischenergebnis mitgeführt und aktualisiert wird. Die Konstruktionsanleitungen für Prozeduren, die Listen oder natürliche Zahlen verarbeiten, führen aber bei direkter Anwendung zu Prozeduren, die kein Zwischenergebnisses mitführen können: Solche Probleme erfordern deshalb eine neue Programmiertechnik, das Programmieren mit *Akkumulatoren*, und entsprechend angepaßte Konstruktionsanleitungen.

8.1 Zwischenergebnisse mitführen

Gefragt ist eine Prozedur, welche eine Liste invertiert, also die Reihenfolge ihrer Elemente umdreht:

```
; Liste umdrehen
(: invert ((list %a) -> (list %a)))

(check-expect (invert empty) empty)
(check-expect (invert (list 1 2 3 4)) (list 4 3 2 1))
```

Gerüst und Schablone sehen wie folgt aus:

```
(define invert
  (lambda (lis)
    (cond
      ((empty? lis) ...)
      ((pair? lis)
       ... (first lis) ...
       ... (invert (rest lis)) ...))))
```

Der Ausdruck `(invert (rest lis))` liefert den Rest der Liste in umgekehrter Reihenfolge. Falls `lis`, wie im zweiten Testfall, also die Liste `#<list 1 2 3 4>` ist, so ist der invertierte Rest `#<list 4 3 2>`. Das gewünschte Ergebnis `#<list 4 3 2 1>` entsteht durch das Anhängen des ersten Elements *hinten* an die Liste. Durch Wunschdenken wird eine Prozedur `append-element` angenommen, die hinten an eine Liste ein Element anhängt:

```
; Element an Liste anhängen
(: append-element ((list %a) %a -> (list %a)))
```

Mit Hilfe von `append-element` läßt sich `invert` leicht vervollständigen:

```
(define invert
  (lambda (lis)
    (cond
      ((empty? lis) empty)
      ((pair? lis)
       (append-element (invert (rest lis))
                       (first lis))))))
```

Die Prozedur `append-element` ist ganz ähnlich der Prozedur `concatenate` aus Abschnitt 7.2. Zunächst Testfälle:

```
(check-expect (append-element (list 1 2 3) 4) (list 1 2 3 4))
(check-expect (append-element empty 4) (list 4))
```

Gerüst und Schablone:

```
(define append-element
  (lambda (lis el)
    (cond
      ((empty? lis) ...)
      ((pair? lis)
       ... (first lis) ...
       ... (append-element (rest lis) el) ...))))
```

Die Schablone läßt sich leicht vervollständigen:

```
(define append-element
  (lambda (lis el)
    (cond
      ((empty? lis) (list el))
      ((pair? lis)
       (make-pair (first lis)
                  (append-element (rest lis) el))))))
```

Doch zurück zu `invert`. Obwohl die zu erledigende Aufgabe einfach erscheint, dauert schon das Invertieren von Listen der Länge 1000 eine ganze Weile.[1] Tatsächlich ist es so, daß z.B. das Invertieren einer Liste der Länge 400 *mehr* als doppelt so lang wie das Invertieren einer Liste der Länge 200 benötigt. Das liegt daran, daß `invert` bei jedem rekursiven Aufruf `append-element` aufruft, und `append-element` selbst macht soviele rekursive Aufrufe wie die Liste lang ist. Das wiederum heißt aber, daß die Gesamtanzahl

[1] So war es zumindest zur Zeit der Drucklegung dieses Buchs auf handelsüblichen Computern. Ggf. müssen es auf moderneren Rechnern Listen der Länge 10000 sein, um das Problem deutlich zu machen.

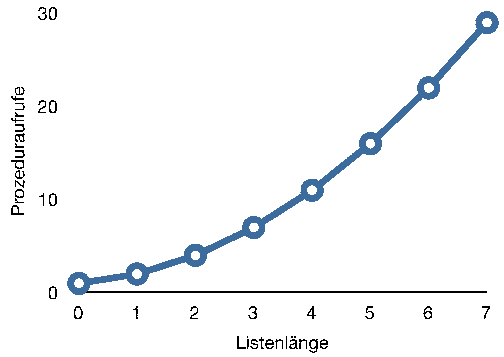


Abbildung 8.1 Prozeduraufrufe bei `invert`

der Prozeduraufrufe für das Invertieren einer Liste der Länge *n* so steigt wie in der Kurve in Abbildung 8.1 gezeigt, also offenbar stärker als linear: Das erklärt das überproportionale Ansteigen der Rechenzeit. (Dafür ist auch Aufgabe 8.1 relevant.) Dies ist für so eine einfache Aufgabe inakzeptabel: Listen der Länge 10000 sind nichts ungewöhnliches, und das Invertieren sollte dem Computer leichtfallen.

Tatsächlich gibt es eine bessere Methode, eine Liste umzudrehen: Die obige `invert`-Prozedur konstruiert die Ergebnisliste, indem stets Elemente *hinten* angehängt werden. Das entspricht nicht der „natürlichen“ Konstruktion von Listen mit `make-pair`, das ein Element *vorn* anhängt. Das Ergebnis ließe sich aber durch Anhängen vorn ganz einfach konstruieren, und zwar, indem in folgender Reihenfolge Zwischenergebnisse berechnet werden, wie in folgendem Beispiel für den Testfall `(invert (list 1 2 3 4))`:

```
#<empty-list>
#<list 1>
#<list 2 1>
#<list 3 2 1>
#<list 4 3 2 1>
```

Jedes Zwischenergebnis entsteht aus dem vorhergehenden, indem ein Element vorn an die Liste darüber angehängt wird. Dies geschieht in der Reihenfolge, in der die Elemente in der ursprünglichen Liste auftreten: scheinbar einfach. Allerdings erlaubt die normale Konstruktionsanleitung für Listen nicht, dieses Zwischenergebnis mitzuführen: Das Ergebnis des rekursiven Aufrufs `(invert (rest lis))` ist unabhängig vom Wert von `(first lis)`. Damit aber ist es der Prozedur aus der normalen Konstruktionsanleitung unmöglich, die obige Folge von Zwischenergebnissen nachzuvollziehen, da von einem Zwischenergebnis zum nächsten gerade `(first lis)` vorn angehängt wird. Für diesen speziellen Fall – wenn eine Berechnung das Mitführen von Zwischenergebnissen erfordert – muß die normale Konstruktionsanleitung deshalb angepaßt werden.

Dieses Problem läßt sich durch Mitführen des Zwischenergebnisses in einem separaten Parameter lösen, dem sogenannten *Akkumulator*. Dazu wird eine Hilfsprozedur `invert-helper` definiert, die neben der Eingabeliste diesen Akkumulator konsumiert:

```
(: invert-helper ((list %a) (list %a) -> (list %a)))

(define invert-helper
  (lambda (lis acc)
    ...))
```

Die Liste `lis` ist nach wie vor die bestimmende Eingabe, es greifen also die entsprechenden Konstruktionsanleitungen für gemischte und zusammengesetzte Daten:

```
(define invert-helper
  (lambda (lis acc)
    (cond
      ((empty? lis) ...)
      ((pair? lis)
       ... (first lis) ...
       ... (invert-helper (rest lis) ...) ...))))
```

Wenn `invert-helper` aufgerufen wird, sind die noch zu verarbeitenden Elemente der ursprünglichen Liste in `lis`, und das Zwischenergebnis, was aus den Elementen davor berechnet wurde, ist in `acc`. Wenn `lis` leer ist, sind alle Elemente verarbeitet und das Zwischenergebnis ist das Endergebnis:

```
(define invert-helper
  (lambda (lis acc)
    (cond
      ((empty? lis) acc)
      ((pair? lis)
       ... (first lis) ...
       ... (invert-helper (rest lis) ...) ...))))
```

Für den rekursiven Aufruf muß noch ein neuer Wert für `acc` übergeben werden. Dieser entsteht, wie im Beispiel zu sehen ist, dadurch, daß an den Akkumulator das erste Element der Liste vorn angehängt wird:

```
(define invert-helper
  (lambda (lis acc)
    (cond
      ((empty? lis) acc)
      ((pair? lis)
       ... (invert-helper (rest lis)
                          (make-pair (first lis) acc)) ...))))
```

Da der rekursive Aufrufs von `invert-helper` schließlich direkt das Endergebnis zurückgegeben wird, ist damit die Prozedur auch schon fertig:

```
(define invert-helper
  (lambda (lis acc)
```

```
(cond
  ((empty? lis) acc)
  ((pair? lis)
   (invert-helper (rest lis)
                  (make-pair (first lis) acc))))))
```

Die neue Hilfsprozedur `invert-helper` paßt nicht auf den Vertrag von `invert`; `invert` muß also separat definiert werden und den passenden Anfangswert für `acc` übergeben:

```
(define invert
  (lambda (lis)
    (invert-helper lis empty)))
```

Da `invert-helper` ausschließlich als Hilfsprozedur zu `invert` benutzt wird und ohne `invert` keinen Nutzen hat, ist es nicht notwendig, einen separaten Vertrag für `invert-helper` anzugeben.

Die neue Version von `invert` kommt ohne `append-element` aus. Der Beispielaufruf von oben führt zu folgender Auswertung im Substitutionsmodell, die sich auch im Stepper gut nachvollziehen läßt:

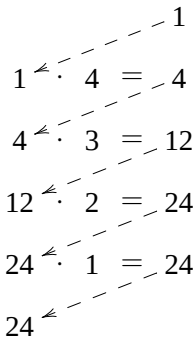
```
(invert (list 1 2 3 4))
⟹ ... ⟹ (invert-helper #<list 1 2 3 4> empty)
⟹ ... ⟹ (cond ((empty? #<list 1 2 3 4>) ...) ((pair? #<list 1 2 3 4>) ...))
⟹ ... ⟹ (invert-helper (rest #<list 1 2 3 4>) (make-pair (first #<list 1 2 3 4>) empty))
⟹ ... ⟹ (invert-helper #<list 2 3 4> (make-pair 1 empty))
⟹ ... ⟹ (invert-helper #<list 2 3 4> #<list 1>)
⟹ ... ⟹ (cond ((empty? #<list 2 3 4>) ...) ((pair? #<list 2 3 4>) ...))
⟹ ... ⟹ (invert-helper (rest #<list 2 3 4>) (make-pair (first #<list 2 3 4>) #<list 1>))
⟹ ... ⟹ (invert-helper #<list 3 4> (make-pair 2 #<list 1>))
⟹ ... ⟹ (invert-helper #<list 3 4> #<list 2 1>)
⟹ ... ⟹ (cond ((empty? #<list 3 4>) ...) ((pair? #<list 3 4>) ...))
⟹ ... ⟹ (invert-helper (rest #<list 3 4>) (make-pair (first #<list 3 4>) #<list 2 1>))
⟹ ... ⟹ (invert-helper #<list 4> (make-pair 3 #<list 2 1>))
⟹ ... ⟹ (invert-helper #<list 4> #<list 3 2 1>)
⟹ ... ⟹ (cond ((empty? #<list 4>) ...) ((pair? #<list 4>) ...))
⟹ ... ⟹ (invert-helper (rest #<list 4>) (make-pair (first #<list 4>) empty))
⟹ ... ⟹ (invert-helper #<empty-list> (make-pair 4 #<list 3 2 1>))
⟹ ... ⟹ (invert-helper #<empty-list> #<list 4 3 2 1>)
⟹ ... ⟹ (cond ((empty? #<empty-list>) #<list 4 3 2 1>) ((pair? #<empty-list>) ...))
⟹ #<list 4 3 2 1>
```

Tatsächlich arbeitet die neue Prozedur auch effizienter: Da `invert-helper` nicht bei jedem Aufruf selbst wieder eine Prozedur aufruft, die eine komplette Liste verarbeitet, steigt die Rechenzeit nur noch proportional zur Länge der Liste.

Da die Prozedur `invert` generell nützlich ist, ist sie unter dem Namen `reverse` fest eingebaut.

Akkumulatoren, die Zwischenergebnisse verwalten, sind bei der Lösung einer Reihe von Problemen nützlich. Zum Beispiel könnte eine Prozedur, welche die Fakultät *n*! einer Zahl

n berechnet, vorgehen, indem sie das Produkt $n \cdot \ldots \cdot 1$ schrittweise von links her ausrechnet:



Der Anfangswert für das Zwischenergebnis ist 1, also gerade die Fakultät von 0.

Für die Konstruktion wird die Schablone für Prozeduren, die natürliche Zahlen verarbeiten, um einen Akkumulator erweitert:

```
; Fakultät berechnen

(: ! (natural -> natural))

(check-expect (! 0) 1)
(check-expect (! 3) 6)
(check-expect (! 5) 120)

(define !
  (lambda (n)
    (!-helper n 1)))

(define !-helper
  (lambda (n acc)
    (if (= n 0)
        acc
        ... (!-helper (- n 1) ... acc ...) ...)))
```

Wie aus der Beispielrechnung ersichtlich ist, wird aus dem „alten“ Zwischenergebnis das „neue“ Zwischenergebnis, indem jeweils mit `n` multipliziert wird:

```
(define !-helper
  (lambda (n acc)
    (if (= n 0)
        acc
        ... (!-helper (- n 1) (* n acc)) ...)))
```

Wie schon bei `invert` ist es nicht notwendig, daß für die noch verbleibenden Ellipsen etwas eingesetzt wird; das Programm ist bereits fertig:

```
(define !-helper
  (lambda (n acc)
    (if (= n 0)
        acc
        (!-helper (- n 1) (* acc n)))))
```

Tatsächlich ist es bei Prozeduren mit Akkumulator grundsätzlich nicht notwendig, für die Ellipsen am Schluß etwas einzusetzen. Bei der normalen Schablone für Prozeduren, die Listen bzw. natürliche Zahlen verarbeiten, wird für diese Ellipsen Code eingesetzt, was das erste Element der Liste mit dem Ergebnis des rekursiven Ausdrucks zum Rückgabewert kombiniert. Dies ist beim Einsatz eines Akkumulators aber nicht notwendig, da das erste Element der Liste bereits in die Berechnung des nächsten Zwischenergebnisses eingeht und dieses Zwischenergebnis beim letzten Aufruf bereits das Endergebnis ist.

8.2 Schablonen für Prozeduren mit Akkumulator

Aus den beiden Beispielen des vorgehenden Abschnitts ergeben sich direkt Schablonen für Prozeduren mit Akkumulator. Zunächst die Schablone für Prozeduren mit Akkumulator, die Listen konsumieren:

```
(: proc ((list elem) -> ...))

(define proc
  (lambda (lis)
    (proc-helper lis z)))

(define proc-helper
  (lambda (lis acc)
    (cond
      ((empty? lis) acc)
      ((pair? lis)
       (proc-helper (rest lis)
                    (... (first lis) ... acc ...))))))
```

Hier ist *proc* der Name der zu definierenden Prozedur und *proc*`-helper` der Name der Hilfsprozedur mit Akkumulator. Der Anfangswert für den Akkumulator – also das initiale Zwischenergebnis – ist der Wert von *z*. Der Ausdruck, der für `(... (first lis) ... acc ...)` einzusetzen ist, macht aus dem alten Zwischenergebnis `acc` das neue Zwischenergebnis.

Die Schablone für Prozeduren mit Akkumulator, die natürliche Zahlen konsumieren, ist analog:

```
(: proc (natural -> ...))

(define proc
```

```
  (lambda (n)
    (proc-helper n z)))

(define proc-helper
  (lambda (n acc)
    (if (= n 0)
        acc
        (proc-helper (- n 1) (... acc ...)))))
```

Wieder ist *z* der Ausdruck für das initiale Zwischenergebnis und für `(... acc ...)` ist ein Ausdruck einzusetzen, der aus dem alten Zwischenergebnis `acc` ein neues macht.

8.3 Kontext und Endrekursion

Ein Vergleich der beiden Versionen der Fakultätsfunktion von S. 74 und S. 108 zeigt, daß Formulierungen mit und ohne Akkumulator unterschiedliche Berechnungsprozesse erzeugen. Hier ein Prozeß mit Akkumulator:

```
(! 4)
⟹ (!-helper 4 1)
⟹ (if (= 4 0) 1 (!-helper (- 4 1) (* 1 4)))
⟹ (if #f 1 (!-helper (- 4 1) (* 1 4)))
⟹ (!-helper (- 4 1) (* 1 4))
⟹ (!-helper 3 4)
⟹ (if (= 3 0) 4 (!-helper (- 3 1) (* 4 3)))
⟹ (if #f 4 (!-helper (- 3 1) (* 4 3)))
⟹ (!-helper (- 3 1) (* 4 3))
⟹ (!-helper 2 12)
⟹ (if (= 2 0) 12 (!-helper (- 2 1) (* 12 2)))
⟹ (if #f 12 (!-helper (- 2 1) (* 12 2)))
⟹ (!-helper (- 2 1) (* 12 2))
⟹ (!-helper 1 24)
⟹ (if (= 1 0) 24 (!-helper (- 1 1) (* 24 1)))
⟹ (if #f 24 (!-helper (- 1 1) (* 24 1)))
⟹ (!-helper (- 1 1) (* 1 24))
⟹ (!-helper 0 24)
⟹ (if (= 0 0) 24 (!-helper (- 0 1) (* 24 0)))
⟹ (if #t 24 (!-helper (- 0 1) (* 24 0)))
⟹ 24
```

Demgegenüber hier der Prozeß ohne Akkumulator:

```
(! 4)
⟹ (if (= 4 0) 1 (* 4 (! (- 4 1))))
⟹ (if #f 1 (* 4 (! (- 4 1))))
```

```
⟹ (* 4 (! (- 4 1)))
⟹ (* 4 (! 3))
⟹ (* 4 (if (= 3 0) 1 (* 3 (! (- 3 1)))))
⟹ (* 4 (if #f 1 (* 3 (! (- 3 1)))))
⟹ (* 4 (* 3 (! (- 3 1))))
⟹ (* 4 (* 3 (! 2)))
...
⟹ (* 4 (* 3 (* 2 (! 1))))
⟹ (* 4 (* 3 (* 2 (if (= 1 0) 1 (* 1 (! (- 1 1)))))))
⟹ (* 4 (* 3 (* 2 (if #f ... (* 1 (! (- 1 1)))))))
⟹ (* 4 (* 3 (* 2 (* 1 (! (- 1 1))))))
⟹ (* 4 (* 3 (* 2 (* 1 (! 0)))))
⟹ (* 4 (* 3 (* 2 (* 1 (if (= 0 0) 1 (* 0 (! (- 0 1))))))))
⟹ (* 4 (* 3 (* 2 (* 1 (if #t 1 (* 0 (! (- 0 1))))))))
⟹ (* 4 (* 3 (* 2 (* 1 1))))
⟹ (* 4 (* 3 (* 2 1)))
⟹ (* 4 (* 3 2))
⟹ (* 4 6)
⟹ 24
```

Es ist deutlich sichtbar, daß die Version ohne Akkumulator alle Multiplikationen bis zum Schluß „aufstaut“. Das heißt aber auch, daß im Laufe des Berechnungsprozesses Ausdrücke auftauchen, die desto größer werden je größer das Argument von `!` ist: Bei `(! 100)` werden zum Beispiel 100 Multiplikationen aufgestaut.

Die Version mit Akkumulator hingegen scheint in der Größe der zwischenzeitlich auftretenden Ausdrücke begrenzt zu sein. Tatsächlich stellt sich das Wachstum der Version ohne Akkumulator bei der Version mit Akkumulator nicht ein.

Der Grund dafür sind die Schablonen: In der Schablone für Prozeduren ohne Akkumulator steht `... (`*proc* `(- n 1)) ...`, das heißt, um den rekursiven Aufruf von *proc* wird noch etwas „herumgewickelt“, oder, anders gesagt, mit dem Ergebnis des rekursiven Aufrufs passiert noch etwas. Das, was mit dem Ergebnis noch passiert, heißt der *Kontext* des Aufrufs. Bei `!` ist der vollständige Ausdruck `(* n (! (- n 1)))`. Wenn aus diesem Ausdruck der rekursive Aufruf `(! (- n 1))` herausgenommen wird, bleibt der Kontext `(* n ○)`, wobei ○ markiert, wo der Aufruf entfernt wurde. Tatsächlich wird in der Literatur diese Markierung *Loch* genannt und `[]` geschrieben. Der Kontext `(* n [])` macht deutlich, daß mit Ergebnis eines Aufrufs, der später für `[]` eingesetzt wird, noch `n` multipliziert wird. Dementsprechend stauen sich in der Reduktionsfolge die Multiplikationen mit den verschiedenen Werten von `n`.

Bei der Fakultäts-Prozedur mit Akkumulator ist der Ausdruck, zu dem der Rumpf bei $n \neq 0$ reduziert wird, `(!-helper (- n 1) (* n acc))`. Der Kontext des Aufrufs von `!-helper` innerhalb dieses Ausdrucks ist `[]`, also *leer – nichts* passiert mehr mit dem Rückgabewert von `!-helper`, und damit stauen sich auch bei der Reduktion keine Kontexte an. Solche Prozeduraufrufe ohne Kontext heißen *endrekursiv* – eben, weil nach dem

rekursiven Aufruf „Ende“ ist.[2] Die Berechnungsprozesse, die von endrekursiven Aufrufen generiert werden, heißen auch *iterative* Prozesse.

8.4 Das Phänomen der umgedrehten Liste

Die beiden Varianten der Fakultäts-Prozedur berechnen zwar beide stets das gleiche Ergebnis. Die beiden Reduktionsfolgen für `(! 4)` aus dem vorigen Abschnitt zeigen allerdings, daß die beiden Prozeduren bei der Berechnung unterschiedlich vorgehen: Während die Variante ohne Akkumulator „von rechts“ multipliziert, also folgendermaßen auswertet:

$$4 \cdot (3 \cdot (2 \cdot (1 \cdot 1)))$$

multipliziert die Variante mit Akkumulator „von links“:

$$(((1 \cdot 4) \cdot 3) \cdot 2) \cdot 1$$

Die Multiplikationen passieren also in umgekehrter Reihenfolge. Dies macht bei der Fakultät keinen Unterschied, da die Multiplikation assoziativ ist. Diese Assoziativität ist jedoch nicht immer gegeben – insbesondere nicht bei Prozeduren, die Listen zurückgeben. Hier zum Beispiel eine Prozedur, die eine Zahl *n* konsumiert und eine absteigende Liste der Zahlen von *n* bis 1 zurückliefert:

```
; Liste der Zahlen von n bis 1 generieren
(: build-list (natural -> (list natural)))

(check-expect (build-list 0) empty)
(check-expect (build-list 3) (list 3 2 1))

(define build-list
  (lambda (n)
    (if (= n 0)
        empty
        (make-pair n (build-list (- n 1))))))
```

Die direkte Übersetzung in eine Variante mit Akkumulator liefert:

```
(define build-list
  (lambda (n)
    (build-list-helper n empty)))

(define build-list-helper
  (lambda (n acc)
    (if (= n 0)
        acc
        (build-list-helper (- n 1) (make-pair n acc)))))
```

[2] Das Konzept des Aufrufs ohne Kontext ist nicht auf rekursive Aufrufe beschränkt. Im Englischen heißen solche Aufrufe allgemeiner *tail calls* (also ohne „recursive“).

Diese Variante ist inkorrekt: Sie liefert z.B. für `(build-list 3)` das Ergebnis `#<list 1 2 3>`, die Elemente der Liste sind also in umgekehrter Reihenfolge. Da schon die Fakultätsprozedur mit Akkumulator die Multiplikationen gegenüber der Variante ohne Akkumulator in umgekehrter Reihenfolge durchgeführt hat, war dies allerdings zu erwarten, und ist ein generelles Phänomen bei der Berechnung von Listen-Ausgaben mit Akkumulator. Das Problem kann durch das Umdrehen der Ergebnisliste gelöst werden:

```
(define build-list-helper
  (lambda (n acc)
    (if (= n 0)
        (reverse acc)
        (build-list-helper (- n 1) (make-pair n acc)))))
```

Anmerkungen

Bei der Auswertung von Programmen durch den Computer wird für die Verwaltung von Kontexten Speicherplatz benötigt: Bei rekursiven Prozeduren ohne Akkumulator wächst dieser Speicherplatz mit der Größe der Argumente. Entsprechend wird prinzipiell kein Speicherplatz benötigt, wenn kein Kontext anfällt. In Scheme wird auch tatsächlich kein Speicherplatz für endrekursive Aufrufe verbraucht; dies ist allerdings bei vielen anderen Programmiersprachen nicht der Fall. Mehr dazu in Kapitel 17.

Aufgaben

Aufgabe 8.1 Entwicklen Sie eine Formel für die Anzahl der rekursiven Aufrufe in der ersten Version von `invert!` (Hinweis: Greifen Sie auf die Gauß'sche Summenformel zurück.)

Aufgabe 8.2 Schreiben Sie eine Prozedur `list-sum+product`, die eine Liste von Zahlen konsumiert und eine zweielementige Liste zurückgibt, deren erstes Element die Summe der Listenelemente und deren zweites Element ihr Produkt ist. Schreiben Sie zwei Varianten der Prozedur: eine ohne Akkumulator und eine mit zwei Akkumulatoren.

Aufgabe 8.3 Schreiben Sie eine Prozedur, die als Eingabe eine Liste von Kursen einer Aktie (als Zahlen) eines Tages akzeptiert (nach Tageszeit aufsteigend sortiert), und als Rückgabewert den höchstmöglichen Gewinn liefert, die durch den Kauf und folgenden Verkauf der Aktie an diesem Tag erreicht werden kann.

Hinweis: Diese Prozedur benötigt zwei Akkumulatoren.

Aufgabe 8.4 Schreibe zu der Prozedur `power` aus Aufgabe 6.9 eine endrekursive Variante.

Aufgabe 8.5 Identifizieren Sie die Kontexte der Aufrufe der Prozeduren namens `p` in folgenden Ausdrücken:

```
(+ (p (- n 1)) 1)
(p (- n 1) acc)
(* (p (rest lis)) b)
(+ (* 2 (p (- n 1))) 1)
(p (- n 1) (* acc n))
(f (p n))
(+ (f (p n)) 5)
(p (f (- n 1)) (* n (h n)))
(+ (f (p n)) (h n))
```

Welche Aufrufe sind endrekursiv bzw. *tail calls*?