Informatik I

6. Rekursion auf Zahlen und Endrekursion

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

11. November 2010

Informatik I

11. November 2010 — 6. Rekursion auf Zahlen und Endrekursion

6.1 Die Sorte der natürlichen Zahlen

Eine Vereinfachung von Listen

Erinnern wir uns an `liste` (beliebige Elementsorte):

```
(define-record-procedures leere-liste
  make-leere-liste leer?
  ())
(: make-leere-liste (-> leere-liste))
(define leer (make-leere-liste))
(define liste
  (signature
   (mixed leere-liste nll                )))
(define-record-procedures nll
  kons nichtleer?
  (kopf rumpf))
(: kons (%a liste  -> liste    ))
(: kopf (nll -> %a))
(: rumpf (nll   -> liste                 ))
```

Eine Vereinfachung von Listen

Wir werfen das Feld für den Kopf weg:

```
(define-record-procedures leere-liste
  make-leere-liste leer?
  ())
(: make-leere-liste (-> leere-liste))
(define leer (make-leere-liste))
(define liste
  (signature
   (mixed leere-liste nll                )))
(define-record-procedures nll
  kons nichtleer?
  (     rumpf))
(: kons (   liste  -> liste    ))

(: rumpf (nll   -> liste                 ))
```

Eine Vereinfachung von Listen

Jetzt nehmen wir ein paar Umbenennungen vor:

```
(define-record-procedures null-sorte
  make-null-sorte  null?
  ())
(: make-null-sorte  (-> null-sorte))
(define null (make-null-sorte ))
(define natürlich
  (signature
   (mixed null-sorte  nichtnull-sorte)))
(define-record-procedures nichtnull-sorte
  sukz nichtnull?
  (     präd))
(: sukz (natürlich -> natürlich))

(: präd (nichtnull-sorte -> natürlich))
```

Natürliche Zahlen: Bemerkungen

- Wir haben soeben die Sorte der natürlichen Zahlen als gemischte Sorte aus zwei Recordsorten selbst definiert, genauso wie wir das vorher für `zahlenliste` und andere Listensorten getan haben.
- Wir haben wieder deutsche Namen verwendet, um Kollisionen auszuschließen: `null`, Sorte `natürlich`, Konstruktor `sukz`essor (Nachfolger), Selektor `präd`ezessor (Vorgänger).
- Die Definition ist sehr ähnlich wie die für Listen, aber sogar noch einfacher!
- Für theoretische (mathematische) Überlegungen zu den natürlichen Zahlen ist diese Vorgehensweise genau die richtige!

Konstruktionsanleitung 7 (Listen)

Erinnern wir uns an die Schablone für eine Prozedur, die eine (Zahlen-)Liste konsumiert:

```
(: p ((zahlenliste) -> τ))
(define p
  (lambda (l)
    (cond
      ((leer? l) ...)
      ((nichtleer?  l)
       ... (kopf l)
       ... (p (rumpf l)) ...))))
```

- Fülle zuerst den `leer?`-Zweig aus.
- Fülle dann den `nichtleer?`-Zweig aus unter der Annahme, dass der rekursive Aufruf `(p (rumpf l))` das gewünschte Ergebnis für den Rest der Liste liefert.

Übersetzung auf natürliche Zahlen

```
(: p ((natürlich) -> τ))
(define p
  (lambda (n)
    (cond
      ((null? n) ...)
      ((nichtnull?  n)
        ... (p (präd n)) ...))))
```

Rekursion für natürliche Zahlen entwickeln

Wie geht es jetzt weiter?
Wir könnten jetzt längere Zeit mit den selbst definierten natürlichen Zahlen weiterarbeiten, wie wir das für Listen getan haben. Das tun wir nicht! Warum?

- Einmal reicht! Wir würden nichts Neues über das Selbst-Bauen von Sorten lernen.
- Im Falle der natürlichen Zahlen wäre es gar zu künstlich und ineffizient, nicht die eingebaute Sorte zu verwenden.

Warum dann die letzten Folien?

- Wir verstehen, dass die natürlichen Zahlen konzeptionell eine gemischte Sorte aus zwei Recordsorten, sehr ähnlich zu den Listen, sind, und können dementsprechende Konstruktionsanleitungen wiederverwenden.

Weg von den selbst definierten natürlichen Zahlen

Erster Schritt: Sorte `natural` verwenden

Die Konstruktoren und der Selektor für natürliche Zahlen sind:

```
; Konstruktor für Null
(: zero natural)
(define zero 0)
; Konstruktor für Nachfolger
(: succ (natural -> natural))
(define succ
  (lambda (x) (+ x 1)))
; Selektor
(: posnat (natural -> boolean)) ; Hilfsprädikat
(define posnat (lambda (n) (positive? n)))
(: pred ((predicate posnat) -> natural))
(define pred
  (lambda (x) (- x 1)))
```

Weg von den selbst definierten natürlichen Zahlen

Zweiter Schritt: Welche Namen werden verwendet?

- Das „Sortenprädikat“ `zero?` ist eingebaut und wir werden es verwenden.
- Statt `pred n` werden wir `(- n 1)` verwenden.
- Wir werden kein „Sortenprädikat“ `nichtnull?` verwenden; stattdessen ein „else“.

Wir werden dies in Kürze sehen . . .

6.2 Prozeduren, die natürliche Zahlen konsumieren

- Beispiel
- Konstruktionsanleitung

Beispiel: die Fakultätsfunktion

Die Fakultätsfunktion ist folgendermaßen definiert:

$$n! \quad := \quad n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 1$$

Das heißt,

$$\begin{aligned} 1! &= 1 \\ 2! &= 2 \\ 3! &= 6 \\ 4! &= 24 \\ &\vdots \end{aligned}$$

Ansatz für gemischte Daten

Die natürlichen Zahlen sind konzeptionell eine gemischte Sorte, also verwenden wir den Ansatz für gemischte Daten:

```
; n! berechnen
(: factorial (natural -> natural))
(define factorial
  (lambda (n)
    (cond
      ((zero? n) ...)
      (else      ...))))
```

Beachte: kein „Sortenprädikat“ `nichtnull?`, sondern ein `else`.

if statt cond

Da wir es bei natürlichen Zahlen immer mit zwei Fällen (Mischung aus `null-sorte` und `nichtnull-sorte`) zu tun haben, verwenden wir `if` anstelle von `cond`:

```
; n! berechnen
(: factorial (natural -> natural))
(define factorial
  (lambda (n)
    (if (zero? n)
        ...
        ...)))
```

Weiter mit Ansatz für zusammengesetzte Daten

Im `else`-Zweig gilt `(> n 0)`, also ist die Vorgängerfunktion auf `n` anwendbar. Wir wenden also den Selektor auf das Argument an, wie immer bei zusammengesetzten Daten (`nichtnull-sorte` ist als Record definiert):

```
; n! berechnen
(: factorial (natural -> natural))
(define factorial
  (lambda (n)
    (if (zero? n)
        ...
        (... (- n 1) ...))))
```

Anstatt `(pred n)` schreiben wir `(- n 1)`.

Rekursiver Aufruf

Ganz genau wie bei Listen rufen wir `factorial` nun rekursiv auf:

```
; n! berechnen
(: factorial (natural -> natural))
(define factorial
  (lambda (n)
    (if (zero? n)
        ...
        ... (factorial (- n 1)) ...)))
```

- Die Definition war: $n! := n \cdot (n-1) \cdot (n-2) \cdots 1$
- Daher ist `(factorial n)` gleich `(* n (factorial (- n 1)))`.

Fertiger „else“-Zweig

```
; n! berechnen
(: factorial (natural -> natural))
(define factorial
  (lambda (n)
    (if (zero? n)
        ...
        (* n (factorial (- n 1))))))
```

Es fehlt der (Basis-) Fall `(factorial 0)` ...

(factorial 0)

Um (factorial 0) zu „berechnen“, betrachten wir die Berechnung von (factorial 1) und schauen nach, wie (factorial 0) definiert sein müsste, damit das Richtige herauskommt:

```
   (factorial 1)
=> (if (zero? 1) ... (* 1 (factorial (- 1 1))))
=> (* 1 (factorial (- 1 1)))
=> (* 1 (factorial 0))
=> (* 1 (if (zero? 0) ... (* 0 (factorial (- 0 1)))))
=> (* 1 ...)
   ; die einzige Möglichkeit ist ... = 1
=> 1
```

Damit (factorial 1) gemäß rekursiver Prozedur richtig berechnet wird, muss (factorial 0) als 1 definiert sein.

0! in der Mathematik

0! ist in der Mathematik in der Tat als 1 definiert.

Die Fakultät von 0 ist das leere Produkt, welches als das neutrale Element der Multiplikation definiert ist, also 1.

Auch in unserer Prozedur brauchen wir, dass `(factorial 0)` als das neutrale Element der Multiplikation definiert ist.

Die fertige Prozedur

```
; n! berechnen
(: factorial (natural -> natural))
(define factorial
  (lambda (n)
    (if (zero? n)
        1
        (* n (factorial (- n 1))))))
```

Konstruktionsanleitung 8 (natürliche Zahlen)

Eine Prozedur, die eine natürliche Zahl konsumiert, hat folgende Schablone:

```
(: p (natural -> τ ))
(define p
  (lambda (n)
    (if (zero? n)
        ...
        ... (p (- n 1)) ...)))
```

- Fülle zuerst den `zero?`-Zweig aus.
- Fülle dann den „else“-Zweig aus unter der Annahme, dass `(p (- n 1))`, der rekursive Aufruf von *p*, das gewünschte Ergebnis für den Vorgänger der natürlichen Zahl liefert.

6.3 Nichttermination

Nichttermination

Eine rekursive Funktion, die nicht eine der Konstruktionsanleitungen für Listen oder natürliche Zahlen verwendet, muss nicht immer ein Ergebnis liefern.

Nichttermination

Beispiel: direkter rekursiver Aufruf

```
; nichts tun
(: waste-time (natural -> natural))
(define waste-time
  (lambda (x)
    (waste-time x)))
```

Terminiert nicht:

```
   (waste-time 0)
=> (waste-time 0)
=> (waste-time 0)
=> ...
```

Nichttermination

Beispiel: Fehler in der Schablone (Selektor vergessen)

```
; n! nicht berechnen
(: notfac (natural -> natural))
(define notfac
  (lambda (n)
    (if (zero? n)
        1
        (* n (notfac n)))))
```

Terminiert nicht:

```
   (notfac 2)
=> (* 2 (notfac 2))
=> (* 2 (* 2 (notfac 2)))
=> (* 2 (* 2 (* 2 (notfac 2))))
=> (* 2 (* 2 (* 2 (* 2 (notfac 2)))))
=> ...
```

Nichttermination

Beispiel: Fehler in der Schablone (Fallunterscheidung/Basisfall vergessen)

```
; n! nicht berechnen
(: notfac2 (natural -> natural))
(define notfac2
  (lambda (n)
    (* n (notfac2 (- n 1)))))
```

Terminiert nicht:

```
   (notfac2 4)
=> (* 4 (notfac2 3))
=> (* 4 (* 3 (notfac2 2)))
=> (* 4 (* 3 (* 2 (notfac2 1))))
=> (* 4 (* 3 (* 2 (* 1 (notfac2 0)))))
=> (* 4 (* 3 (* 2 (* 1 (* 0 (notfac2 -1)))))) ...
```

Terminierende Berechnungsprozesse

- Ein Berechnungsprozess terminiert, falls er nach endlich vielen Berechnungsschritten mit einem Wert endet.
- Andernfalls ist der Berechnungsprozess nichtterminierend.
- Eine Prozedur, die nach der Konstruktionsanleitung für Listen bzw. für natürliche Zahlen geschrieben ist, terminiert für jede signaturgemäße Eingabe.
- Im Allgemeinen ist das Problem der Nichttermination nicht lösbar (siehe Informatik III, 3. Semester).
- Man kann „Termination“ oder „Terminierung“ sagen.

6.4 Rekursion auf mehreren Argumenten

- Rekursion auf dem ersten Argument
- Rekursion auf dem zweiten Argument
- Rekursion auf beiden Argumenten

Rekursion auf mehreren Argumenten

- Manche Prozeduren konsumieren mehrere natürliche Zahlen, z.B. die eingebauten arithmetischen Operatoren.
- Wir werden jetzt betrachten, wie in diesen Fällen rekursive Prozeduren konstruiert werden können.
- Um ein einfaches Beispiel zu haben, nehmen wir die natürlichen Zahlen und definieren die binären `plus`, `minus`, `times`. Wir tun also so, als gäbe es +, −, und ∗ noch nicht.
- Wir werden die Problematik später auch an Listen wiedersehen.

Unser Ausgangspunkt

Die Konstruktoren und der Selektor für natürliche Zahlen sind:

```
; Konstruktor für Null
(: zero natural)
(define zero 0)
; Konstruktor für Nachfolger
(: succ (natural -> natural))
(define succ
  (lambda (x) (+ x 1)))
; Selektor
(: posnat (natural -> boolean)) ;Hilfsprädikat
(define posnat (lambda (n) (positive? n)))
(: pred ((predicate posnat) -> natural))
(define pred
  (lambda (x) (- x 1)))
```

Von nun ab verwenden wir + und - nicht mehr.

Die Prozedur plus

Wir beginnen mit Signatur, Gerüst und Testfällen:

```
; plus: zwei natürliche Zahlen addieren
(: plus (natural natural -> natural))
(define
  plus
  (lambda (n m)
    (...)))
(check-expect (plus zero zero) zero)
(check-expect (plus (succ zero) zero) (succ zero))
(check-expect
   (plus (succ (succ (succ zero)))
         (succ (succ zero)))
   (succ (succ (succ (succ (succ zero))))))
```

Konstruktionsanleitung anwenden

Wir wenden die Konstruktionsanleitung für natürliche Zahlen nur auf das erste Argument an:

```
(define
  plus
  (lambda (n m)
    (if (zero? n)
        ...
        (... (plus (pred n) m) ...)
        )))
```

Das zweite Argument lassen wir im rekursiven Aufruf unverändert.

Vollständige Prozedur

Wir haben Glück! Es gibt nicht mehr viel zu tun:

```
(define
  plus
  (lambda (n m)
    (if (zero? n)
        m
        (succ (plus (pred n) m))
        )))
```

`plus` lässt sich sehr gut mittels Rekursion auf dem ersten Argument definieren.

Konstruktionsanleitung anwenden

Nun eine alternative Definition:
Wir wenden die Konstruktionsanleitung für natürliche Zahlen nur auf das zweite Argument an:

```
(define
  plus
  (lambda (n m)
    (if (zero? m)
        ...
        (... (plus n (pred m)) ...)
        )))
```

Das erste Argument lassen wir im rekursiven Aufruf unverändert.

Vollständige Prozedur

Wir haben Glück! Es gibt nicht mehr viel zu tun:

```
(define
  plus
  (lambda (n m)
    (if (zero? m)
        n
        (succ (plus n (pred m)))
        )))
```

`plus` lässt sich sehr gut mittels Rekursion auf dem zweiten Argument definieren.
Da `plus` kommutativ ist, ist das nicht überraschend.

Noch eine alternative Definition

Nochmals das Gerüst:

```
(define
  plus
  (lambda (n m)
    (...)))
```

Wir führen jetzt zunächst eine Verzweigung über `n` ein.

Verzweigung über n

```
(define
  plus
  (lambda (n m)
    (if (zero? n)
        ...
        ...
        )
    ))
```

Dann ersetzen wir jede Ellipse durch eine Verzweigung über m.

Verzweigung über m

```
(define
  plus
  (lambda (n m)
    (if (zero? n)
        (if (zero? m)
            ...
            ...
            )
        (if (zero? m)
            ...
            ...
            )
        )
    ))
```

Man könnte jetzt versuchen, wo möglich rekursive Aufrufe einzuführen.
Aber vielleicht lassen sich einige der vier Zweige schnell verarzten ...

Drei Zweige sind einfach

```
(define
  plus
  (lambda (n m)
    (if (zero? n)
        (if (zero? m)
            zero
            m
            )
        (if (zero? m)
            n
            ...
            )
        )
    ))
```

Jetzt der rekursive Aufruf auf beiden Argumenten ...

Der rekursive Aufruf

```
(define
  plus
  (lambda (n m)
    (if (zero? n)
        (if (zero? m)
            zero
            m
            )
        (if (zero? m)
            n
            (... (plus (pred n) (pred m)) ...)
            )
        )
    ))
```

Vollständige Prozedur

```
(define
  plus
  (lambda (n m)
    (if (zero? n)
        (if (zero? m) zero m)
        (if (zero? m)
            n
            (succ (succ (plus (pred n) (pred m))))
            )
        )
    ))
```

Dies lässt sich noch vereinfachen . . .

Vollständige Prozedur vereinfacht

```
(define
  plus
  (lambda (n m)
    (if (zero? n)
        m
        (if (zero? m)
            n
            (succ (succ (plus (pred n) (pred m))))
            )
        )
    ))
```

`plus` lässt sich gut mittels Rekursion auf beiden Argumenten definieren.

Bemerkungen

- Wir haben gesehen: Rekursion auf einem der Argumente, oder auf beiden. Alle drei Ansätze haben für `plus` gut funktioniert.
- Es kann auch komplizierter werden: unterschiedliche rekursive Aufrufe in verschiedenen Zweigen, Hilfsprozeduren ...
- Wenn ein Ansatz zu kompliziert wird, ist er wahrscheinlich nicht der am besten geeignete. Dann abbrechen und einen anderen versuchen!

Übung

Schreiben Sie eine Prozedur für `minus`. Versuchen Sie alle drei obigen Ansätze:

- Rekursion auf dem zweiten Argument funktioniert gut.
- Rekursion auf beiden Argumenten geht am besten.
- Rekursion auf dem ersten Argument ist sehr schwierig. Der erste, der sich mit einer Lösung per Email meldet, bekommt vom Dozenten eine Dose Spam.

Im Betreuten Programmieren wird es eine ähnliche Aufgabe geben.

6.5 Endrekursion, Iteration, Akkumulatoren

- Platzprobleme
- Entwurf einer Hilfsfunktion
- Endrekursion und Iteration
- Listen

Größe von Prozessen

Der Berechnungsprozess für `factorial` wächst mit der Größe der Eingabe:

```
=> (factorial 10)
=> (* 10 (factorial 9))
=> (* 10 (* 9 (factorial 8)))
=> (* 10 (* 9 (* 8 (factorial 7))))
=> (* 10 (* 9 (* 8 (* 7 (factorial 6)))))
=> (* 10 (* 9 (* 8 (* 7 (* 6 (factorial 5))))))
=> (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (factorial 4)))))))
=> (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (* 4 (factorial 3))))))))
=> (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (* 4 (* 3 (factorial 2)))))))))
=> (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 (factorial 1))))))))))
=> (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 (* 1 (factorial 0)))))))))))
=> (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 (* 1 1))))))))))
=> (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 1))))))))))
=> (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (* 4 (* 3 2))))))))
=> (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (* 4 6)))))))
=> (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 24))))))
=> (* 10 (* 9 (* 8 (* 7 (* 6 120)))))
=> (* 10 (* 9 (* 8 (* 7 720))))
=> (* 10 (* 9 (* 8 5040)))
=> (* 10 (* 9 40320))
=> (* 10 362880)
=> 3268800
```

Platzverbrauch eines Berechnungsprozesses

- Die eigentliche arithmetische Berechnung (eingebaute Prozedur `*`) kann erst beginnen, wenn der letzte rekursive Aufruf `(factorial 0)` stattgefunden hat. Dabei haben sich 10 Multiplikationen angesammelt.
- Die Größe des Ausdrucks ist also proportional („bis auf Kleinigkeiten“) zum Wert des Arguments, und zugleich zum Platzverbrauch für die Auswertung in der Maschine.
- Es wäre schön, wenn wir den Berechnungsprozess so ändern könnten, dass die Ausdrücke eine bestimmte Größe nicht überschreiten, so dass der Platzverbrauch konstant wäre.

Wie Platz sparen?

- Irgendwie müssten wir dafür sorgen, dass die Auswertung der Multiplikationen früher beginnen kann, sodass sich diese nicht ansammeln.
- Z.B. der Ausdruck

  ```
  (* 10 (* 9 (* 8 (* 7 (factorial 6)))))
  ```

  kann erst auswertet werden, wenn der Wert von `(factorial 6)` vorliegt.
- Ausnutzen der Assoziativität von `*` liefert

  ```
  (* (* (* (* 10 9) 8) 7) (factorial 6))
  ```

  Also könnte man die Multiplikationen vorziehen:

  ```
  (* 5040 (factorial 6))
  ```

Wie Multiplikationen vorziehen?

- Wir sind immer noch bei der Berechnung von `(factorial 10)`, die nach einigen Schritten `(factorial 6)` aufruft.
- Wir müssen die Definition von `factorial` so aufzäumen, dass in dem Moment, in dem `(factorial 6)` aufgerufen wird, `(* (* (* 10 9) 8) 7)` $= 5040$ schon ausgerechnet ist, statt dass sich diese Multiplikationen ansammeln. Der Aufruf von `(factorial 6)` muss 5040 als „Zusatzinformation“ mitbekommen.
- `factorial` „mit Zusatzinformation“? Das wird natürlich eine ganz neue Funktion, die ein Argument mehr hat als `factorial`. Wir nennen diese Hilfsfunktion `it-factorial-1`.

Die Aufrufe von `it-factorial-1`

Die Aufrufe von `it-factorial-1` bei der Berechnung von `(it-factorial-1 10 ...)` sind:

```
(it-factorial-1  10       1); 1
(it-factorial-1   9      10); 10
(it-factorial-1   8      90); (* 10 9)
(it-factorial-1   7     720); (* (* 10 9) 8)
(it-factorial-1   6    5040); (* (* (* 10 9) 8) 7)
(it-factorial-1   5   30240);
(it-factorial-1   4  151200);
(it-factorial-1   3  604800);
(it-factorial-1   2 1814400);
(it-factorial-1   1 3628800); (* ... (* 10 ... 3) 2)
(it-factorial-1   0 3628800); (* ... (* 10 ... 2) 1)
```

Die Aufrufe von `it-factorial-1` allgemein

Die Aufrufe von `it-factorial-1` bei der Berechnung von (`it-factorial-1` m 1) sind:

(`it-factorial-1`	m	1)
(`it-factorial-1`	$(m-1)$	m)
(`it-factorial-1`	$(m-2)$	$(m \cdot (m-1))$)
	$\vdots$	
(`it-factorial-1`	1	$(m \cdot (m-1) \cdot \ldots \cdot 3 \cdot 2)$)
(`it-factorial-1`	0	$(m \cdot (m-1) \cdot \ldots \cdot 3 \cdot 2 \cdot 1)$)

d.h., für alle $i \in \{0, \ldots, m\}$,

(`it-factorial-1`	$(m-i)$	$\frac{m!}{(m-i)!}$)

Was soll `it-factorial-1` berechnen?

Was soll ein Aufruf von `(it-factorial-1` n k`)` als Ergebnis liefern?
Zwei Überlegungen:

- `(factorial` n`)` berechnet $n!$. Dann sollte `(it-factorial-1` n k`)` nicht dasselbe berechnen, sondern irgendwie das zusätzliche Argument k verwenden. Idee?
  `(it-factorial-1` n k`)` sollte $n! \cdot k$ berechnen.
- Wie oben gesehen, haben wir Aufrufe der Form `(it-factorial-1` $(m-i)$ $\frac{m!}{(m-i)!}$`)`, die letztlich „dazu dienen“ sollen, $m!$ zu berechnen. Mit welchem Ergebnis wäre denn am besten „gedient“? Mit $m!$.

Gemäß erster Überlegung berechnet `(it-factorial-1` $(m-i)$ $\frac{m!}{(m-i)!}$`)` den Wert $(m-i)! \cdot \frac{m!}{(m-i)!}$, d.h., $m!$, genau wie gemäß der zweiten Überlegung.

Anfang und Ende

`(it-factorial-1` m `1)` berechnet $m! \cdot 1$, also $m!$.

`(it-factorial-1 0` $m!$`)` berechnet $0! \cdot m!$, also $m!$.

Daraus können wir ablesen, dass `(it-factorial-1 0` k`)` als Ergebnis einfach das Argument k zurückgeben sollte:

```
; Produkt n! * k berechnen
(: it-factorial-1 (natural natural -> natural))
(define it-factorial-1
  (lambda (n k)
    (if (zero? n)
        k
        ...)))
```

Das zweite Argument ist ein Akkumulator: es enthält ein Zwischenergebnis der Berechnung, das am Ende der rekursiven Aufrufe als Endergebnis zurückgegeben wird.

Der rekursive Aufruf in der Definition

Wir haben gesehen: es wird für alle $i \in \{0, \ldots, m\}$

$$(\texttt{it-factorial-1}\ (m - i)\ \tfrac{m!}{(m-i)!})$$

aufgerufen, d.h. für gegebenes $n \geq 0$ folgt auf

$$(\texttt{it-factorial-1}\ (n - i)\ \tfrac{m!}{(m-i)!})$$

im nächsten Schritt

$$(\texttt{it-factorial-1}\ (m - (i + 1))\ \tfrac{m!}{(m-(i+1))!}).$$

Letzteres formen wir um

$$(\texttt{it-factorial-1}\ ((m - i) - 1))\ \tfrac{m!}{((m-i)-1))!})$$
$$(\texttt{it-factorial-1}\ ((n - i) - 1))\ ((n - i) \cdot \tfrac{m!}{(m-i)!}))$$

Definition von it-factorial-1

Somit haben wir also hergeleitet, wie it-factorial-1 definiert sein muss:

```
; Produkt n! * k berechnen
(: it-factorial-1 (natural natural -> natural))
(define it-factorial-1
  (lambda (n k)
    (if (zero? n)
        k
        (it-factorial-1 (- n 1) (* n k)))))
```

Aufruf der Hilfsfunktion

```
; Produkt der Zahlen 1 .. n berechnen
(: it-factorial (natural -> natural))
(define it-factorial
  (lambda (n)
    (it-factorial-1 n 1)))
```

Berechnungsprozess

```
   (it-factorial 3)
=> (it-factorial-1 3 1)
=> (if (zero? 3)
       1
       (it-factorial-1 (- 3 1) (* 3 1)))
=> (it-factorial-1 2 3)
=> (if (zero? 2)
       3
       (it-factorial-1 (- 2 1) (* 2 3)))
=> (it-factorial-1 1 6)
=> (if (zero? 1)
       6
       (it-factorial-1 (- 1 1) (* 1 6)))
=> (it-factorial-1 0 6)
=> (if (zero? 0)
       6
       (it-factorial-1 (- 0 1) (* 0 6)))
=> 6
```

Ein Berechnungsprozess konstanter Größe

```
=> (it-factorial-1 15             1)
=> (it-factorial-1 14            15)
=> (it-factorial-1 13           210)
=> (it-factorial-1 12          2730)
=> (it-factorial-1 11         32760)
=> (it-factorial-1 10        360360)
=> (it-factorial-1  9       3603600)
=> (it-factorial-1  8      32432400)
=> (it-factorial-1  7     259459200)
=> (it-factorial-1  6    1816214400)
=> (it-factorial-1  5   10897286400)
=> (it-factorial-1  4   54486432000)
=> (it-factorial-1  3  217945728000)
=> (it-factorial-1  2  653837184000)
=> (it-factorial-1  1 1307674368000)
=> (it-factorial-1  0 1307674368000)
=> 1307674368000
```

Größe des Berechnungsprozesses bleibt konstant.

Endrekursion und Iteration

Ein Berechnungsprozess ist iterativ, falls seine Größe konstant bleibt.
(Später, für das imperative Programmieren, werden wir den Begriff „iterativ“ noch etwas anders definieren/charakterisieren.)
Ein Funktionsaufruf ist endrekursiv (engl. tail-recursive), falls er den vorangehenden Funktionsaufruf vollständig ersetzt.
Beispiel:

```
(define it-factorial-1
  (lambda (n result)
    (if (zero? n)
        result
        (it-factorial-1 (- n 1) (* n result)))))
```

Alternative Beschreibungen von Endrekursion

- Ein Funktionsaufruf ist endrekursiv (engl. tail-recursive), falls er den vorangehenden Funktionsaufruf vollständig ersetzt.
- Der zweite Zweig der Verzweigung enthält nicht den rekursiven Aufruf als einen (echten) Teil, sondern der zweite Zweig besteht nur aus diesem rekursiven Aufruf.
- Das letzte, was die Prozedur tut, ist, sich selbst rekursiv aufzurufen (diese Ausdrucksweise wird vor allem dem imperativen Programmieren gerecht, siehe später).

Gegenbeispiel

Größe des Berechnungsprozesses ist nicht konstant (hier: proportional zur Eingabe)

```
(define factorial
  (lambda (n)
    (if (zero? n)
        1
        (* n (factorial (- n 1))))))
```

Prozeduren mit Akkumulatoren

Prozeduren mit Akkumulatoren sind wichtig, da ihre Berechnungsprozesse iterativ sind!

Konstruktionsanleitung: Prozedur mit Akkumulator

Konsumieren von natürlichen Zahlen

```
(: proc (natural -> τ))
(define proc
  (lambda (n)
    (proc-helper n initial)))

(: proc-helper (natural τ -> τ))
(define proc-helper
  (lambda (n acc)
    (if (zero? n)
        acc
        (proc-helper (- n 1) (... n ... acc ...)))))
```

Dabei muss initial ein „neutrales, initiales Zwischenergebnis“ sein, und `(... n ... acc ...)` berechnet das nächste Zwischenergebnis.

Konstruktionsanleitung: Prozedur mit Akkumulator

Konsumieren von Listen

```
(: proc ((list σ) -> τ))
(define proc
  (lambda (l)
    (proc-helper l initial)))

(: proc-helper ((list σ) τ -> τ))
(define proc-helper
  (lambda (l acc)
    (if (empty? l)
        acc
        (proc-helper (rest l)
                     (... (first l) ... acc ...)))))
```

Dabei muss initial ein „neutrales, initiales Zwischenergebnis“ sein, und `(... (first l) ... acc ...)` berechnet das nächste Zwischenergebnis.

Hinweis zur Konstruktionsanleitung

Diese Konstruktionsanleitung ist bei weitem nicht so konkret wie frühere Konstruktionsanleitungen.
Eine Prozedur mit Akkumulator zu schreiben, ist nicht einfach und erfordert in recht frühem Stadium ein genaues Verständnis des Problems.
Sie müssen die Frage stellen:

- Um das Resultat für n zu berechnen, kann man sinnvollerweise irgendein Zwischenergebnis aus n und $n-1$ berechnen?
- Um das Resultat für eine Liste zu berechnen, kann man sinnvollerweise irgendein Zwischenergebnis aus dem ersten und zweiten Element berechnen?

Falls ja, ist das Problem ein Kandidat für eine Prozedur mit Akkumulator.

Beispiel: Prozedur mit Akkumulator (Listen)

Hier eine Prozedur, die eine Liste invertiert, also die z.B. bei Eingabe `#<list 1 2 3>` das Ergebnis `#<list 3 2 1>` berechnet.

```
(: invert ((list %a) -> (list %a)))
(define invert
  (lambda (l)
    (invert-helper l empty)))

(: invert-helper ((list %a) (list %a) -> (list %a)))
(define invert-helper
  (lambda (l acc)
    (if (empty? l)
        acc
        (invert-helper (rest l) (cons (first l) acc)))))
```

Zusammenfassung

- Natürliche Zahlen als gemischte, rekursive Sorte
- Rekursion: natürliche Zahlen als Argumente
- Nichttermination
- Rekursion auf mehreren Argumenten
- Endrekursion, Iteration, Akkumulatoren
- Endrekursion, Iteration, Akkumulatoren auf Listen