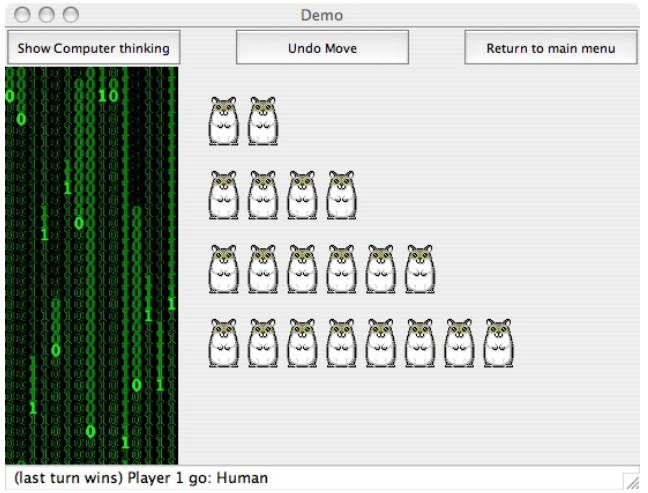
Fallstudie: Nim Spiel



- Angeblich chinesischen Ursprungs (*Jianshizi*)
- Interessant für Spieltheorie: vollständig analysierbar
- Frühzeitig „computerisiert“
  - 1939 Nimatron (Weltausstellung New York)
  - 1951 Nimrod (Berliner Handelsmesse)

Das Nim Spiel

Regeln

Es gibt zwei Stapel von Münzen. Zu Beginn enthalten beide Stapel gleich viele Münzen.

Die beiden Spieler ziehen abwechselnd. In einem Zug entfernt der Spieler beliebig viele Münzen von einem Stapel seiner Wahl. Es gewinnt der Spieler, der die letzte Münze aufnimmt.

Aufgaben

- Erstelle Datenstrukturen für den Spielstand und für einen Zug.
- Erstelle eine Prozedur, die einen Zug ausführt.

3 Rekursion

3.1 Polymorphe Verträge

Beobachtung: Manche Prozeduren arbeiten unabhängig von der Sorte ihrer Argumente, dh. sie verhalten sich **polymorph** (vom griechischen $\pi o \lambda \upsilon \mu o \rho \varphi o \varsigma$, vielgestalt).

Beispiele:

```
; Die Identität: Argument unverändert zurückgeben
(: identity (%value -> %value))
(define identity
  (lambda (x)
    x))

; Die konstante Funktion: Das erste Argument unverändert zurückgeben
(: const (%value %value -> %value))
(define const
  (lambda (x y)
    x))
```

```
; Projektion: ein Argument auswählen
(: proj ((one-of 1 2) %value %value -> %value))
(define proj
  (lambda (i x1 x2)
    (cond
     ((= i 1) x1)
     ((= i 2) x2))))
```

Kennzeichen: der Vertrag der Prozeduren enthält Vertragsvariable, die jeden Wert akzeptieren

3.1.1 Parametrisch polymorphe Verträge

Die Verträge mit `%value` sind unbefriedigend und ungenau!

Betrachte

```
(: identity (%value -> %value))
```

Aussage: wenn das Argument von `identity` irgendein Wert ist, so ist das Ergebnis auch irgendein (anderer) Wert.

Aber:

- `(identity 1)` liefert `number`
- `(identity "xyz")` liefert `string`
- `(identity #f)` liefert `boolean`
- dh. `identity` angewendet auf Wert der Sorte `A` liefert einen Wert der Sorte `A`!

3.1.2 Beispiele für parametrisch polymorphe Verträge

```
; Die Identität: Argument unverändert zurückgeben
(: identity (%a -> %a))
(define identity
  (lambda (x) x))

; Die konstante Funktion: Das erste Argument unverändert zurückgeben
(: const (%a %b -> %a))
(define const
  (lambda (x y) x))

; Projektion: ein Argument auswählen
(: proj ((one-of 1 2) %a %a -> %a))
(define proj
  (lambda (i x1 x2)
    (cond
     ((= i 1) x1)
     ((= i 2) x2))))
```

3.1.3 Schreibweise für parametrisch polymorphe Verträge

- Eine Prozedur, die einen Teil ihrer Argumente immer gleichartig behandelt, kann einen *parametrisch polymorphen Vertrag* erhalten.
- Der polymorphe Vertrag verwendet eine *Sortenvariable* (wie z.B. `%a`, `%b`, ...) um über eine konkrete Sorte zu abstrahieren.
- Der parametrisch polymorphe Vertrag fasst alle Verträge zusammen, wo Sorten konsistent für die Sortenvariablen eingesetzt werden. (Vertragsinstanzen)
- Beispiel: Wenn p den Vertrag (`number %a %a -> %a`) erfüllt, dann erfüllt p alle folgenden Verträge
  - `number number number -> number`
  - `number string string -> string`
  - `number boolean boolean -> boolean`
  - `number cookie cookie -> cookie`

3.1.4 Paare

```
; Ein Paar ist ein Wert
;  (make-mypair a b)
; wobei a und b jeweils beliebige Werte sind.
```

Die zugehörige Record-Definition

```
(define-record-procedures mypair
  make-mypair mypair?
  (myfirst myrest))
```

liefert Prozeduren mit folgenden Verträgen

```
(: make-mypair (%value %value -> mypair))
(: mypair? (%value -> boolean))
(: myfirst (mypair -> %value))
(: myrest (mypair -> %value))
```

3.1.5 Paare - verbessert

```
; Ein Paar von A und B ist ein Wert
;  (make-mypair a b)
; wobei a und b jeweils Werte aus A bzw. B sind.
```

Die zugehörige parametrische Record-Definition

```
(define-record-procedures-parametric mypair mypair-of
  make-mypair mypair?
  (myfirst myrest))
```

liefert Prozeduren mit folgenden Verträgen

```
(: make-mypair (%a %b -> (mypair-of %a %b)))
(: mypair? (%value -> boolean))
(: myfirst ((mypair-of %a %b) -> %a))
(: myrest ((mypair-of %a %b) -> %b))
```

3.2 Listen

3.2.1 Die leere Liste

```
; Leere Liste
; empty : empty
```

Verwendung

```
   empty
=> #<empty-list>
```

Das Typprädikat `empty?`

```
   (empty? empty)
=> #t
   (empty? "wasanderes")
=> #f
```

3.2.2 Listen aus Paaren und `empty`

```
; Eine Liste von As ist eins der folgenden
;  - die leere Liste
;  - ein Paar (aus einem A-Wert und einer Liste von As)
; Name: (mylist-of %a)
```

d.h. eine `(mylist-of %a)` verwendet `make-mypair` mit der Vertragsinstanz

```
(: make-mypair (%a (mylist-of %a) -> (mylist-of %a)))
```

- Beobachtung: Liste tritt in der Definition von Liste auf (innerhalb eines Paars).
- Liste ist ein rekursiver Datentyp definiert durch eine rekursive Definition.

3.2.3 Vertragdefinition für Listen

- Standardvertrag ohne Parameter

```
(define mylist (mixed (predicate empty?) mypair))
```

- "Liste von A" ist ein gemischter Datentyp.
- Der Vertrag für (`mylist` %a) ist *parametrisch*, daher muss er durch eine Lambda-Abstraktion definiert werden!

```
(define mylist-of
  (lambda (elem-contract)
    (contract
     (mixed (predicate empty?)
            (mypair-of elem-contract (mylist-of elem-contract))))))
```

3.2.4 Vordefinierte Paare und Listen

Ab Sprachlevel "Die Macht der Abstraktion":

```
; Eine Liste von As ist eins der folgenden
;  - die leere Liste
;  - ein Paar (aus einem Wert und einer Liste von As)
; Name: (list %value)
```

dabei verwenden wir folgende Vereinbarung

```
; Ein Paar ist ein Wert
;  (make-pair x y)
; wobei x ein Wert aus A ist und y eine Liste von As.
(: make-pair (%a (list %a) -> (list %a)))
```

3.2.5 Beispiele für Listen

```
empty
(make-pair 1 empty)
(make-pair 1 (make-pair 2 empty))
(make-pair 1 (make-pair 2 (make-pair 3 empty)))
```

Liefern die Ausgabe

```
#<empty-list>
#<record:pair 1 #<empty-list>>
#<record:pair 1 #<record:pair 2 #<empty-list>>>
#<record:pair 1 #<record:pair 2 #<record:pair 3 #<empty-list>>>>
```

Die Sorten der Listenelemente müssen nicht gleich sein:

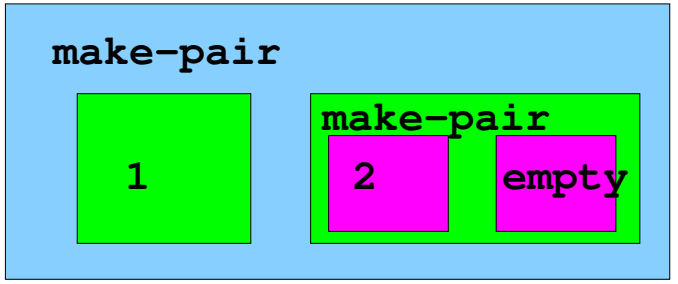
```
   (make-pair 1 (make-pair #t (make-pair "Ende" empty)))
=> #<record:pair 1 #<record:pair #t #<record:pair "Ende" #<empty-list>>>>
```

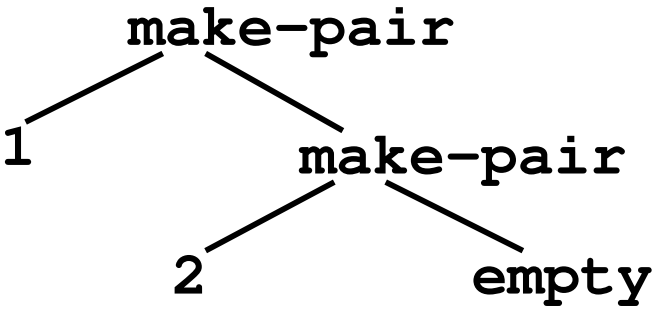
3.2.6 Visualisierung

Die Liste

```
(make-pair 1 (make-pair 2 empty))
```

kann wie folgt dargestellt werden:





3.2.7 Operationen auf Listen

Typprädikate `empty?`, `pair?`

Konstruktor `empty`

liefert die leere Liste

Konstruktor `(: make-pair (%value (list %value) -> (list %value)))`

erweitert die Liste um ein neues *Kopfelement*

Selektor `(: first ((list %value) -> %value))`

liefert das erste Element (Kopfelement)

Selektor `(: rest ((list %value) -> (list %value)))`

liefert den Rest der Liste, d.h. die Liste *ohne* das Kopfelement

3.2.8 Beispiele für die Grundoperationen auf Listen

```
(define liste-1 empty)
(define liste-2 (make-pair 1 liste-1))
(define liste-3 (make-pair 2 liste-2))
(define liste-4 (make-pair 4 liste-3))
```

```
   (first liste-2)
=> 1
   (rest liste-2)
=> #<empty-list>
   (first liste-3)
=> 2
   (rest liste-3)
=> #<record:pair 1 #<empty-list>>
```

3.2.9 Listenverträge

- Standardlistenvertrag: `%value`
  Liste mit beliebigen Elementen
- Andere Verträge für die Listenelemente möglich:
  - `(list number)`
  - `(list string)`
  - `(list (list boolean))`
  - `(list (mixed number string))`
- **Definition:** Die Liste `xs` erfüllt den Vertrag `(list` c`)` genau dann, wenn
  - `xs` ist leer oder
  - `xs` ist ein Paar, so dass
    - `(first xs)` erfüllt den Vertrag c und
    - `(rest xs)` erfüllt den Vertrag `(list` c`)`.

3.3 Prozeduren, die Listen konsumieren

3.3.1 Summe der Listenelemente

```
; Elemente einer Liste addieren
(: list-sum ((list number) -> number))
(define list-sum
  (lambda (xs)
    (cond
     ((empty? xs) ...)
     ((pair? xs)  (... (first xs) ... (rest xs) ...)))))
```

Vorgehensweise bis hierher:

- Konstruktionsanleitung für Prozeduren
- Konstruktionsanleitung für gemischte Daten (Liste ist gemischter Datentyp)

3.3.2 Neuer Schritt: Komponente besitzt Listentyp

- Aus `(: xs (list number))`
  und `(: rest ((list number) -> (list number)))`
- ergibt sich `(rest xs) : (list number)`
- Standardansatz: verwende das Ergebnis von `(list-sum (rest xs))`

```
; Elemente einer Liste addieren
(: list-sum ((list number) -> number))
(define list-sum
  (lambda (xs)
    (cond
     ((empty? xs) ...)
     ((pair? xs)  (... (first xs) ... (list-sum (rest xs)) ...)))))
```

3.3.3 Ausfüllen der Schablone

```
; Elemente einer Liste addieren
(: list-sum ((list number) -> number))
(define list-sum
  (lambda (xs)
    (cond
     ((empty? xs) 0)
     ((pair? xs)  (+ (first xs) (list-sum (rest xs)))))))
```

3.3.4 Länge einer Liste

```
; Anzahl der Elemente einer Liste bestimmen
(: list-length ((list %a) -> number))
(define list-length
  (lambda (l)
    (cond
     ((empty? l)
      ...)
     ((pair? l)
      (... (first l) ... (list-length (rest l)) ...)))))
```

- Gerüst und Schablone genau wie bei `list-sum`

Definition:

```
(define list-length
  (lambda (l)
    (cond
     ((empty? l)
      0)
     ((pair? l)
      (+ 1 (list-length (rest l)))))))
```

Tests:

```
(check-expect (list-length empty) 0)
(check-expect (list-length (liste-3)) 3)
```

- vordefiniert als: `length`
- parametrisch polymorpher Vertrag
  `(: length ((list %a) -> number))`
  die Länge einer Liste ist unabhängig von Sorte der Listenelemente.

Berechnungsprozess

```
   (list-length (make-pair 1 (make-pair 2 empty))))
=> (cond
     ((empty? (make-pair 1 (make-pair 2 empty)))        ; #f
      0)
     ((pair? (make-pair 1 (make-pair 2 empty)))         ; #t
      (+ 1 (list-length (rest (make-pair 1 (make-pair 2 empty)))))))
=> (+ 1 (list-length (rest (make-pair 1 (make-pair 2 empty)))))
=> (+ 1 (list-length (make-pair 2 empty)))
=> (+ 1 (+ 1 (list-length empty)))
=> (+ 1 (+ 1 0))
=> 2
```

Konstruktionsanleitung 7 (Listen)

Eine Prozedur, die eine Liste konsumiert, hat folgende Schablone:

```
(: p ((list c) -> t))
(define p
  (lambda (l)
    (cond
      ((empty? l) ...)
      ((pair?  l)
       ... (first l)
       ... (p (rest l)) ...))))
```

- Fülle zuerst den `empty?` Zweig aus.
- Fülle dann den `pair?` Zweig aus unter der Annahme, dass
  - der **rekursive Aufruf** `(p (rest l))` das gewünschte Ergebnis für den Rest der Liste liefert und
  - `(first l)` den Vertrag c erfüllt.

Mantra #7 (Prozeduren über Listen)

Befolge für Prozeduren, die Listen konsumieren, zuerst die Konstruktionsanleitung und schreibe Vertrag, Gerüst und Schablone auf, *vor tieferem Nachdenken* über die Aufgabenstellung!

Mantra #8 (Flaches Denken)

Denke *niemals* rekursiv über einen rekursiven Prozess nach!

Quelle http://rambleon.org/wp-content/uploads/2007/11/recursion.jpg

3.4 Zusammenfassung

- Polymorphie
- Definition von Listen
- Listenverträge
- Rekursion: Listen als Argumente

3.5 Rekursion über Zahlen

3.5.1 Natürliche Zahlen als gemischter, rekursiver Datentyp

```
; Eine natürliche Zahl ist eins der folgenden
;  - die Null
;  - der Nachfolger einer natürlichen Zahl
; Name: natural
```

Die Konstruktoren für natürliche Zahlen sind demnach

```
; Konstruktor für Null
(: zero natural)
(define zero
  0)
; Konstruktor für Nachfolger
(: succ (natural -> natural))
(define succ
  (lambda (x) (+ x 1)))
```

3.5.2 Selektor für natürliche Zahlen

Der Selektor von `natural` ist nur für natürliche Zahlen größer 0 definiert: die Vorgängerfunktion

```
; Selektor für Nachfolger von natürlichen Zahlen
(: pred (natural -> natural))
(define pred
  (lambda (x) (- x 1)))
```

3.5.3 Prozeduren, die natürliche Zahlen konsumieren

Beispiel: die Fakultätsfunktion

$$
\begin{aligned}
n! &= n \cdot (n-1) \cdot (n-2) \cdots 1 \\
1! &= 1 \\
2! &= 2 \\
3! &= 6 \\
4! &= 24 \\
&\vdots
\end{aligned}
$$

3.5.4 Beispiel: Fakultätsfunktion

Verwende den Ansatz für gemischte Daten ...

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

Anmerkung

- Immer zwei Fälle: Verwende `if` anstelle von `cond`
- Im `else`-Zweig ist `(> n 0)`, also ist die Vorgängerfunktion auf `n` anwendbar

3.5.5 Beispiel: Fakultätsfunktion (Forts.)

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

Anmerkung:

- `(pred n)` ist gleich `(- n 1)`
- Da `(pred n)` wieder die Sorte `natural` hat, ist `factorial` darauf anwendbar.

3.5.6 Beispiel: Fakultätsfunktion (Forts.)

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

Anmerkung:

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

3.5.7 Beispiel: Fakultätsfunktion (Forts.)

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

Anmerkung:

- Es fehlt der (Basis-) Fall `(factorial 0)`.
- Berechne diesen Fall durch Auswertung von `(factorial 1)` mit dem Substitutionsmodell.

3.5.8 `(factorial 1)`

```
   (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
```

3.5.9 Beispiel: Fakultätsfunktion (Ergebnis)

```
; 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 -> t))
(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.

3.6 Nichttermination

- Eine rekursive Funktion, die nicht eine der Konstruktionsanleitungen für Listen oder natürliche Zahlen verwendet, muss nicht immer ein Ergebnis liefern.
- Beispiel: direkter rekursiver Aufruf

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

- Beispiel: Fehler in der Schablone (Selektor vergessen)

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

- Beispiel: Fehler in der Schablone (Fallunterscheidung vergessen)

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

Nichtterminierende Berechnungsprozesse

Alle drei Definitionen erzeugen nichtterminierende Berechnungsprozesse:

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

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

```
   (notfac2 42)
=> (* 42 (notfac2 41))
=> (* 42 (* 41 (notfac2 40)))
=> (* 42 (* 41 (* 40 (notfac2 39))))
=> (* 42 (* 41 (* 40 (* 39 (notfac2 38)))))
=> ...
```

3.6.1 Terminierende Berechnungsprozesse

- Ein Berechnungsprozess **terminiert**, falls er nach endlich vielen Berechnungsschritten mit einem Wert endet.
- Anderenfalls ist der Berechnungsprozess **nichtterminierend**.
- **Eine Prozedur, die nach der Konstruktionsanleitung für Listen bzw. für natürliche Zahlen geschrieben ist, terminiert für jede vertragsgemäße Eingabe.**

3.7 Iterative Berechnungsprozesse

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

```
   (factorial 15)
=> (* 15 (factorial 14))
=> (* 15 (* 14 (factorial 13)))
=> (* 15 (* 14 (* 13 (factorial 12))))
=> (* 15 (* 14 (* 13 (* 12 (factorial 11)))))
=> (* 15 (* 14 (* 13 (* 12 (* 11 (factorial 10))))))
=> (* 15 (* 14 (* 13 (* 12 (* 11 (* 10 (factorial 9)))))))
=> (* 15 (* 14 (* 13 (* 12 (* 11 (* 10 (* 9 (factorial 8))))))))
=> (* 15 (* 14 (* 13 (* 12 (* 11 (* 10 (* 9 (* 8 (factorial 7)))))))))
=> (* 15 (* 14 (* 13 (* 12 (* 11 (* 10 (* 9 (* 8 (* 7 (factorial 6))))))))))
=> (* 15 (* 14 (* 13 (* 12 (* 11 (* 10 (* 9 (* 8 (* 7 (* 6 (factorial 5)))))))))))
=> (* 15 (* 14 (* 13 (* 12 (* 11 (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (factorial 4))))))))))))
=> (* 15 (* 14 (* 13 (* 12 (* 11 (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (* 4 (factorial 3)))))))))))))
=> (* 15 (* 14 (* 13 (* 12 (* 11 (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (* 4 (* 3 (factorial
=> (* 15 (* 14 (* 13 (* 12 (* 11 (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 (facto
=> (* 15 (* 14 (* 13 (* 12 (* 11 (* 10 (* 9 (* 8 (* 7 (* 6 (* 5 (* 4 (* 3 (* 2 (* 1
=> ...
```

3.7.1 Platzverbrauch eines Berechnungsprozesses

- Die Größe des Ausdrucks ist proportional zum Platzverbrauch für die Auswertung in der Maschine.
- Verbesserungsidee: Ändern des Berechnungsprozesses, so dass die Ausdrücke eine bestimmte Größe nicht überschreiten $\Rightarrow$ **konstanter Platzverbrauch**.
- Der Ausdruck

  ```
  (* 15 (* 14 (* 13 (* 12 (* 11 (* 10 (* 9 (* 8 (* 7 (factorial 6))))))))))
  ```

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

  ```
  (* (* (* (* (* (* (* (* (* 15 14) 13) 12) 11) 10) 9) 8) 7) (factorial 6))
  ```
- Multiplikationen könnten vorgezogen werden:

  ```
  (* 1816214400 (factorial 6)))))))))))
  ```

3.7.2 Alternative Definition der Fakultätsfunktion

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

- Ansatz: Definiere Hilfsfunktion `it-factorial-1`
- Das neue, zweite Argument ist ein **Akkumulator**, es schleift ein Zwischenergebnis der Berechnung durch den Berechnungsprozess.
- Ein Aufruf `(it-factorial-1` n r`)` berechnet $n! \cdot r$.
- Also liefert `(it-factorial-1` n 1`)` gerade $n! \cdot 1 = n!$.

3.7.3 Iterative Definition der Hilfsfunktion

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

- Die Prozedur `it-factorial-1` besitzt nur **iterative Berechnungsprozesse**.

3.7.4 Iterativer Berechnungsprozess

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

3.7.5 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
```

$\Rightarrow$ Grösse des iterativen Berechnungsprozesses bleibt konstant

3.7.6 Endrekursion und Iteration

- Ein Berechnungsprozess ist **iterativ**, falls seine Größe konstant bleibt.
- Ein Funktionsaufruf ist **endrekursiv**, falls er den vorangehenden Funktionsaufruf vollständig ersetzt.
  Beispiel
  - Endrekursiv: nur **iterative Berechnungsprozesse**, **Größe konstant**

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

  - Nicht endrekursiv: **rekursive Berechnungsprozesse**,
    Größe nicht konstant (hier: proportional zur Eingabe)

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

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

3.7.7 Konstruktionsanleitung: Prozedur mit Akkumulator (Zahlen)

Prozedur mit Akkumulator, die natürliche Zahlen konsumiert

```
(define result (contract ...))
(: proc (natural -> result))
(define proc
  (lambda (n)
    (proc-helper n result-for-0)))

(: proc-helper (natural result -> result))
(define proc-helper
  (lambda (n acc)
    (if (= n 0)
        acc
        (proc-helper (- n 1) (... n ... acc ...)))))
```

Dabei berechnet result-for-0 das Ergebnis für $n = 0$ und `(... n ... acc ...)` berechnet das nächste Zwischenergebnis.

3.7.8 Konstruktionsanleitung: Prozedur mit Akkumulator (Listen)

Prozedur mit Akkumulator, die Listen konsumiert

```
(define elem (contract ...))
(define result (contract ...))
(: proc ((list elem) -> result))
(define proc
  (lambda (l)
    (proc-helper l result-for-empty)))

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

Dabei berechnet result-for-empty das Ergebnis für die leere Liste und `(... (first l) ... acc ...)` berechnet das nächste Zwischenergebnis.

3.7.9 Beispiel: Prozedur mit Akkumulator (Listen)

Invertieren einer Liste

```
(: 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) (make-pair (first l) acc)))))
```

3.8 Zusammenfassung

- Natürliche Zahlen als gemischter, rekursiver Datentyp
- Rekursion: natürliche Zahlen als Argumente
- Nichttermination
- Endrekursion, Iteration, Akkumulatoren