Informatik I

10. Türme von Hanoi (Listen)

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

9. Dezember 2010

Informatik I

9. Dezember 2010 — 10. Türme von Hanoi (Listen)

10.1 Eingebaute Listen

10.2 Türme von Hanoi

Praktische Programme mit Listen

Dieses Kapitel ist sehr eng angelehnt an Kapitel 7 in unserem Lehrbuch [KS07]: “Praktische Programme mit Listen”.

Es ist allerdings etwas abgespeckt (manches wurde schon früher behandelt), daher bleibt nur ein einziges “praktisches Programm mit Listen” übrig.

10.1 Eingebaute Listen

list und append

Die eingebaute Prozedur `list` erlaubt es, Listen aus ihren Elementen ohne Verwendung von `make-pair` zu erzeugen. Sie akzeptiert eine beliebige Anzahl von Argumenten, macht daraus eine Liste und gibt diese zurück:

```
   (list 1 2 3)
=> #<list 1 2 3>
   (list "Banane" "Apfel" "Birne")
=> #<list "Banane" "Apfel" "Birne">
```

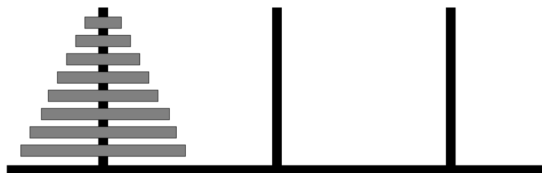
Erinnern wir uns außerdem an die eingebaute Prozedur `append`.

10.2 Türme von Hanoi

- Spielbeschreibung
- Daten
- Die Prozedur
- Hilfsprozeduren
- Fertiges Programm

Spielbeschreibung

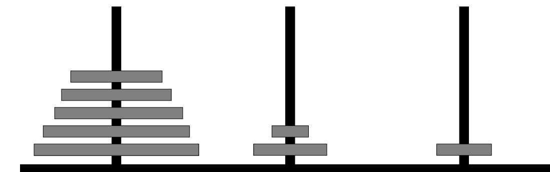
Der Spielzustand besteht aus drei Säulen 1, 2, 3, auf denen insgesamt n Scheiben unterschiedlicher Größe aufgestapelt sind.



Die Aufgabe des Puzzles ist es, den Turm auf einen der anderen Pfähle zu bewegen, allerdings unter folgenden Einschränkungen:

- Es wird nur eine Scheibe auf einmal bewegt.
- Es werden immer nur kleinere auf größere Scheiben gelegt.

Zwischenstellung



- Nicht einfach von Hand zu lösen.
- Schreiben Scheme-Programm
- Konstruktionsanleitungen helfen

Die Daten

1. die Anzahl der zu bewegenden Scheiben: `natural`
2. die Pfähle, auf denen die Scheiben sitzen: `natural` 1, 2, oder 3
3. die Spielzüge, die eine Lösung für das Puzzle beschreiben: Record, um Start- und Zielpfahl zu benennen

Record für einen Spielzug

```
; Ein Hanoi-Spielzug ist ein Wert
;  (make-hanoi-move f t)
; bei dem f und t Nummern von Pfählen sind.
(define-record-procedures hanoi-move
  make-hanoi-move hanoi-move?
  (hanoi-move-from hanoi-move-to))
(: make-hanoi-move (natural natural -> hanoi-move))
```

Folgen von Spielzügen (insbesondere die Lösung) werden durch Listen von `hanoi-move`-Records dargestellt.

Kurzbeschreibung und Signatur

Anzahl von Scheiben muss Parameter der Prozedur sein. Ausgabe ist eine Folge von geeigneten Spielzügen.

```
; Hanoi-Puzzle lösen
(: hanoi (natural -> (list hanoi-move)))
```

Gerüst:

```
(define hanoi
  (lambda (n)
    ...))
```


Konstruktionsanleitung für natürliche Zahlen

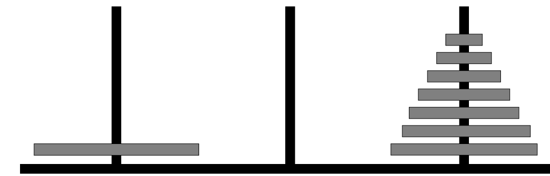
```
(define hanoi
  (lambda (n)
    (if (zero? n)
        ...
        ... (hanoi (- n 1) ...))))
```

Erster Zweig: keine Scheiben

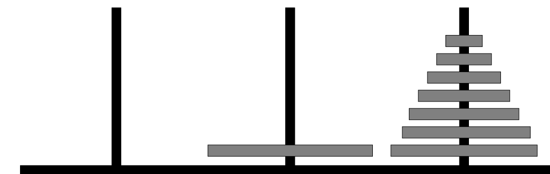
```
(define hanoi
  (lambda (n)
    (if (zero? n)
        empty
        ... (hanoi (- n 1)) ...)))
```


Zweiter Zweig: > 0 Scheiben

Nimm an, `(hanoi (- n 1))` berechnet die richtige Lösung. Dann tut sie bei `n` Scheiben folgendes:

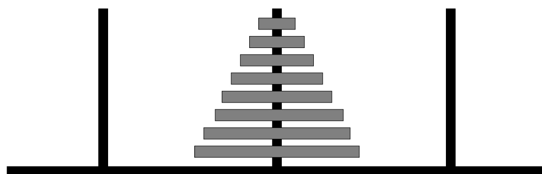


Bewege nun die größte Scheibe auf Pfahl 2:



Lösung wiederverwenden

Lösung für `(hanoi (- n 1))` wiederverwenden, aber "‘Pfahl 1"’ mit "‘Pfahl 3"’ und "‘Pfahl 3"’ mit "‘Pfahl 2"’ vertauschen:



Schönheitsfehler? Zielpfahl stimmt nicht.

Zielpfahl korrigieren

Folgendes muss man ändern:

- Kleineren Turm statt nach Pfahl 3 nach Pfahl 2 bewegen, also "‘Pfahl 3"’ mit "‘Pfahl 2"’ vertauschen;
- große Scheibe von Pfahl 1 nach Pfahl 3 bewegen;
- kleineren Turm von Pfahl 2 nach Pfahl 3 bewegen.

Systematische Umnumerieren der Spielzüge durch Vertauschen jeweils zweier Pfähle.

Weiter mit hanoi

Wir benötigen `(hanoi (- n 1))` zweimal:

```
(define hanoi
  (lambda (n)
    (if (zero? n)
        empty
        ... (hanoi (- n 1)) ...
        ... (hanoi (- n 1)) ...)))
```

Effizienz?
Wir sollten ein `let` benutzen.

let benutzen

```
(define hanoi
  (lambda (n)
    (if (zero? n)
        empty
        (let ((one-less (hanoi (- n 1))))
          (... one-less ...
           ... one-less ...)))))
```


Umnummerierung

Die Spielzüge aus `(hanoi (- n 1))` müssen geeignet umnumeriert werden.

- Erst "‘Pfahl 3"’ mit "‘Pfahl 2"’ vertauschen;
- dann "‘Pfahl 1"’ mit "‘Pfahl 2"’ vertauschen.

Das Umnumerieren ist eine Operation auf Listen von Spielzügen, die am besten von einer Hilfsprozedur erledigt werden sollte.

Hilfsprozedur zur Umnummerierung

Eingaben?

- Zugfolge (Liste von Zügen);
- Zwei Pfahlnummern (die Vertauschung)

Beschreibung und Signatur:

```
; in einer Zugfolge einen Pfahl mit einem anderen
; vertauschen
(: renumber-moves ((list hanoi-move) natural natural
                   -> (list hanoi-move)))
```

Wunschdenken

Wir wollen uns beim Schreiben von `hanoi` nicht ablenken lassen! Daher tun wir so, als wäre `renumber-moves` schon fertig, und verschieben die Fertigstellung auf später: Wunschdenken, Top-Down-Design.

Weiter mit `hanoi`

```
(define hanoi
  (lambda (n)
    (if (zero? n)
        empty
        (let ((one-less (hanoi (- n 1))))
          (... (renumber-moves one-less 3 2) ...
           ... (renumber-moves one-less 1 2) ...)))))
```


Zug mit der größten Scheibe

Nun noch der Zug mit der größten Scheibe:

```
(define hanoi
  (lambda (n)
    (if (zero? n)
        empty
        (let ((one-less (hanoi (- n 1))))
          (... (renumber-moves one-less 3 2) ...
           ... (make-hanoi-move 1 3) ...
           ... (renumber-moves one-less 1 2) ...)))))
```


Kombination

Wir müssen nun die zwei Zugfolgen und den Einzelzug kombinieren. Der Einzelzug und die Zugfolge danach einfach durch `cons`:

```
(define hanoi
  (lambda (n)
    (if (zero? n)
        empty
        (let ((one-less (hanoi (- n 1))))
          ... (renumber-moves one-less 3 2) ...
          (cons (make-hanoi-move 1 3)
                (renumber-moves one-less 1 2))))))
```

Kombination II

Beide Folgen können schließlich mit append aneinandergehängt werden:

```
(define hanoi
  (lambda (n)
    (if (zero? n)
        empty
        (let ((one-less (hanoi (- n 1))))
          (append
           (renumber-moves one-less 3 2)
           (cons (make-hanoi-move 1 3)
                 (renumber-moves one-less 1 2)))))))
```


renumber-moves

Soweit zu hanoi. Es fehlt noch renumber-moves. Erinnerung:

```
; in einer Zugfolge einen Pfahl mit einem anderen
; vertauschen
(: renumber-moves ((list hanoi-move) natural natural
                   -> (list hanoi-move)))
```

Schablone:

```
(define renumber-moves
  (lambda (moves p1 p2)
    (cond
     ((empty? moves) ...)
     ((cons? moves)
      ... (first moves) ...
      ... (renumber-moves (rest moves)
                          p1 p2) ...))))
```


Einzelnen Zug umnumerieren

Nehmen Hilfsprozedur renumber-move mit folgender Beschreibung und Signatur an (Wunschdenken):

```
; in einem Zug einen Pfahl mit einem anderen
; vertauschen
(: renumber-move (hanoi-move natural natural
                  -> hanoi-move))
```

Vervollständigen renumber-moves:

```
(define renumber-moves
  (lambda (moves p1 p2)
    (cond
     ((empty? moves) empty)
     ((cons? moves)
      (cons (renumber-move (first moves) p1 p2)
            (renumber-moves (rest moves) p1 p2))))))
```


Zurück zu renumber-move

Hat Record als Ein- und Ausgabe. Kombinieren zwei Schablonen:

```
(define renumber-move
  (lambda (move p1 p2)
    (make-hanoi-move
     ... (hanoi-move-from move) ...
     ... (hanoi-move-to move) ...)))
```

Zur Umnummerierung von (hanoi-move-from move) und (hanoi-move-to move) verwenden wir wiederum eine Hilfsprozedur namens renumber-peg (Wunschdenken!).

```
; einen Pfahl mit einem anderen vertauschen
(: renumber-peg (natural natural natural -> natural))
```

renumber-move mit renumber-peg

```
(define renumber-move
  (lambda (move p1 p2)
    (make-hanoi-move
     (renumber-peg (hanoi-move-from move) p1 p2)
     (renumber-peg (hanoi-move-to move) p1 p2))))
```


Schließlich zu renumber-peg

Gerüst:

```
(define renumber-peg
  (lambda (peg p1 p2)
    ...))
```

Wie geht es weiter? Fallunterscheidung je nach Wert von peg.
Wie viele Fälle? Drei Fälle, drei Zweige:

```
(define renumber-peg
  (lambda (peg p1 p2)
    (cond
     ((= peg p1) p2)
     ((= peg p2) p1)
     (else peg))))
```


Fertig!

Fertig ist das Hanoi-Programm. Hier ein Beispiel:

```
   (hanoi 3)
=> #<list #<record:hanoi-move 1 3>
          #<record:hanoi-move 1 2>
          #<record:hanoi-move 3 2>
          #<record:hanoi-move 1 3>
          #<record:hanoi-move 2 1>
          #<record:hanoi-move 2 3>
          #<record:hanoi-move 1 3>>
```


Mantra

Mantra #9 (Wunschdenken, Top-Down-Design)

Verschiebe Probleme, die du nicht sofort lösen kannst, in noch zu schreibende Prozeduren. Lege für diese Prozeduren Beschreibung und Signatur fest und benutze sie bereits, schreibe sie aber später.

Zusammenfassung

- Wir haben ein interessantes Programm gesehen, das auf Listen rechnet.
- Wir haben überhaupt einmal ein etwas größeres Programm gesehen.
- Wir haben gelernt, dass Wunschdenken wichtig ist, um beim Programmieren den Überblick zu behalten.

Literatur I

Herbert Klaeren and Michael Sperber.
Die Macht der Abstraktion.
Teubner Verlag, 2007.