

Kapitel 5

Datenabstraktion

Das letzte Kapitel hat sich mit dem Thema der *Datenrepräsentation* beschäftigt: Die zentrale Frage bei der Datenrepräsentation ist: „Was für Sprachmittel können dafür herhalten, um Informationen als Werte in einem Programm zu manipulieren?“ Diese Frage ist häufig schwer zu beantworten. Schlimmer noch, während der Entwicklungszeit eines Programms kann es notwendig werden, die Datenrepräsentation zu ändern. Dies kann schwerwiegende Auswirkungen auf ein Programm haben.

Zur Demonstration dient ein Programm für 2-Haufen-Nim. Es benutzt einige neue Sprachelemente.

5.1 Ein-/Ausgabe und sequentielle Auswertung

Die bisherigen Programme haben recht spärlich mit dem Benutzer interagiert: Sie wurden mit einer Eingabe versorgt, heraus kam eine Ausgabe. Für interaktive Programme ist dies nicht genug. Einige weitere Prozeduren und ein weiteres Sprachmittel erlauben *Ein- und Ausgabe* und damit Interaktion mit dem Benutzer.

Die Prozedur `display` druckt die Repräsentation ihres Parameters in der REPL aus. Der Effekt ist gar nicht so einfach zu demonstrieren, aber immerhin:

```
> (display 5)
5
```

Die Prozedur `newline` (ohne Parameter) gibt einen Zeilenvorschub aus.

```
> (newline)
```

Um wirklich sinnvolle Ein-/Ausgabe zu gestalten, ist es notwendig, mehrere davon hintereinander vorzunehmen. Die neue Spezialform `begin` erlaubt genau dies: Sie wertet jeden ihrer Operanden nacheinander aus und verwirft ihre Werte *bis auf den letzten*. Dieser letzte Wert wird zum Wert der `begin`-Form:

```
<expression> ::= (begin <sequence>)
<sequence> ::= <expression>+
```

Also:

```
> (begin 1 2 3 4 5)
5
> (begin (display 23) (newline) (display 17) (newline))
```

23

17

`begin` umschließt implizit den Rumpf von Abstraktionen und damit auch `let`-Ausdrücken:

```
> (let ((x 1)) (display x) (newline) (display (+ x x)) (newline))
1
2
```

Auch wenn diese Prozeduren zunächst (besonders für Pascal- und C-Programmierer) harmlos aussehen, ist mit `display` und `newline` ein entscheidender Faktor mit in die Programmierung gekommen: Bei allen Ausdrücken bisher hat nur ihr *Wert* interessiert, bei `display` und `newline` interessiert zum ersten Mal der Wert überhaupt nicht, sondern lediglich der *Effekt*. Mit Effekten kommt plötzlich *Reihenfolge* ins Spiel, die vorher zwar festgelegt, aber weitgehend unwichtig war.

Gelegentlich ist es wünschenswert, einen Text auszugeben. Zu diesem Zweck kann der Text in Anführungszeichen eingeschlossen an `display` übergeben werden:

```
> (display "C est si bon.")
C est si bon.
```

Soweit zur Ausgabe. Zur Eingabe kann die Prozedur `read` verwendet werden; sie wartet auf die Eingabe der Repräsentation eines Werts und gibt den entsprechenden Wert zurück:

```
> (read)
5
```

```
5
> (read)
#t
```

```
#t
> (read)
(1 . 2)
```

```
(1 . 2)
```

Eine weitere praktische Prozedur ist `error`. Sie hat beliebig viele Parameter, von denen der erste in Anführungszeichen eingeschlossen sein müssen. Bei der Auswertung von `error` werden die Repräsentationen der Parameter nacheinander ausgegeben; danach wird das Programm beendet. `Error` wird dazu benutzt, um das Programm zu beenden, wenn durch Fehler oder Fehleingaben eine sinnvolle weitere Ausführung nicht mehr möglich ist.

5.2 Syntaktischer Zucker für Conditionals

Conditionals kommen gelegentlich in geschachtelter Form vor. Ein Beispiel (wenn auch ein verworfenes) ist die erste Version von `other-peg`:

```
(define other-peg
  (lambda (peg-1 peg-2)
    (if (= peg-1 1)
        (if (= peg-2 2)
            3
```

```

2)
(if (= peg-1 2)
  (if (= peg-2 1)
    3
    1)
  (if (= peg-2 1)
    2
    1))))

```

Scheme bietet einigen syntaktischen Zucker zur Abkürzung solcher Conditionals. Zunächst einmal sind da die booleschen Junktoren `and` und `or`:

```

⟨expression⟩ ::= (and ⟨test⟩*)
⟨expression⟩ ::= (or ⟨test⟩*)

```

Dabei gelten folgende Regeln:

```

(and)
⇒
#t

```

```

(and t0 t1 ...)
⇒
(if t0 (and t1 ...) #f)

```

```

(or)
⇒
#f

```

```

(or t0 t1 ...)
⇒
(if t0 t0 (or t1 ...))

```

Des Weiteren gibt es noch eine Prozedur `not`, welche aus `#t` `#f` macht und umgekehrt.

Für geschachtelte Conditionals gibt es `cond`:

```

⟨expression⟩ ::= (cond ⟨cond clause⟩+)
⟨expression⟩ ::= (cond ⟨cond clause⟩* (else ⟨sequence⟩))
⟨cond clause⟩ ::= (⟨test⟩ ⟨sequence⟩)

```

Intuitiv wertet `cond` nacheinander alle Tests aus; sobald einer `#t` ergibt, reduziert sich der `cond`-Ausdruck zur entsprechenden rechten Seite. `Else` verhält sich wie ein Test, der nie fehlschlägt.

Formal wird `cond` ebenfalls intern in `if` übersetzt, und zwar nach folgendem induktiven Schema:

```

(cond ((t0 s0) (t1 s2) ...)
⇒
(if t0 s0 (cond ((t1 s2) ...)))

```

```

(cond ((else s))
⇒
s

```

Mit diesen Mitteln könnte die naive Version von `other-peg` so aussehen:

```
(define other-peg
  (lambda (peg-1 peg-2)
    (cond ((= peg-1 1)
          (if (= peg-2 2)
              3
              2))
          ((= peg-1 2)
          (if (= peg-2 1)
              3
              1))
          ((= peg-2 1)
          2)
          (else
          1))))
```

Immer noch nicht toll, aber immerhin besser.

5.3 Symbole und Gleichheit

Manchmal signalisieren Werte Dinge, die sich durch Worte besser beschreiben lassen als durch Zahlen oder Booleans. In einem Nim-Spiel muß ein Wert den Spieler repräsentieren, der an der Reihe an. Zu diesem Zweck (und noch anderen) gibt es in Scheme den Datentyp der *Symbole*, die als externe Repräsentation einen Namen haben. Wie die leere Liste auch sehen Literale und externe Repräsentation bei Symbolen unterschiedlich aus; die Literale fangen mit einem Apostroph an, das bei der externen Repräsentation fehlt:

```
> 'hans-otto
hans-otto
```

Dabei kann jedes Wort ein Symbol sein, was auch eine Variable sein kann und umgekehrt. Entscheidend aber der Unterschied zwischen Symbolen und Variablen:

```
> (define hans-otto 'maria-berta)
> hans-otto
maria-berta
> 'hans-otto
hans-otto
```

Zum Vergleich von Symbolen kann das Prädikat `equal?` verwendet werden:

```
> (equal 'hans-otto 'maria-berta)
#f
> (equal 'hans-otto 'hans-otto)
#t
```

Tatsächlich kann `equal?` für das Testen der Gleichheit *beliebiger* Objekte verwendet werden. (Die Erklärung des Verhaltens von `equal?` ist allerdings dennoch überraschend subtil. Mehr dazu später.)

5.4 Ein Programm für 2-Haufen-Nim

Das Nim-Programm verwendet Paare sowohl für die Repräsentation von Spielständen als auch von Spielzügen:

```

(define play-nim
  (lambda (game-state player)
    (display-game-state game-state)
    (if (over? game-state)
        (announce-winner game-state player)
        (let ((move (next-move game-state player)))
          (announce-move player move)
          (play-nim (apply-move move game-state) (other-player player))))))

(define next-move
  (lambda (game-state player)
    (let ((compute-move
          (cond
            ((equal? player 'human) human-move)
            ((equal? player 'computer) computer-move)
            (else
             (error "player wasn't human or computer:" player))))))
      (compute-move game-state)))

(define other-player
  (lambda (player)
    (cond
      ((equal? player 'human) 'computer)
      ((equal? player 'computer) 'human)
      (else
       (error "player wasn't human or computer:" player))))))

(define computer-move
  (lambda (game-state)
    (if (> (size-of-pile game-state 1) 0)
        (make-move 1 1)
        (make-move 2 1))))

(define prompt
  (lambda (prompt-string)
    (display prompt-string)
    (newline)
    (read)))

(define human-move
  (lambda (game-state)
    (let ((pile (prompt "Which pile will you remove from?")))
      (let ((n-coins (prompt "How many coins do you want to remove?")))
        (make-move pile n-coins)))))

(define over?
  (lambda (game-state)
    (and (= 0 (size-of-pile game-state 1))
          (= 0 (size-of-pile game-state 2)))))

(define announce-winner
  (lambda (game-state player)
    (let ((winner (other-player player)))
      (if (equal? winner 'human)
          (error "human wins")
          (error "computer wins")))))

```

```

        (display "You win. Congratulations.")
        (display "You lose. Better luck next time.))
    (newline)))

(define make-game-state
  (lambda (pile-1 pile-2)
    (cons pile-1 pile-2)))

(define size-of-pile
  (lambda (game-state pile-number)
    (if (= pile-number 1)
        (car game-state)
        (cdr game-state))))

(define display-game-state
  (lambda (game-state)
    (display "  Pile 1: ")
    (display (size-of-pile game-state 1))
    (newline)
    (display "  Pile 2: ")
    (display (size-of-pile game-state 2))
    (newline)))

(define make-move
  (lambda (pile n-coins)
    (cons pile n-coins)))

(define move-pile
  (lambda (move)
    (car move)))

(define move-n-coins
  (lambda (move)
    (cdr move)))

(define announce-move
  (lambda (player move)
    (if (equal? player 'human)
        (display "You")
        (display "I"))
    (display " take ")
    (display (move-n-coins move))
    (display " coins from pile ")
    (display (move-pile move))
    (display ".")
    (newline)))

(define apply-move
  (lambda (move game-state)
    (let ((pile (move-pile move))
          (n-coins (move-n-coins move)))
      (if (= pile 1)
          (make-game-state (- (size-of-pile game-state 1)
                              n-coins)
                            pile)
          (make-game-state (size-of-pile game-state 1)
                            pile))))))

```

```

(size-of-pile game-state 2))
(make-game-state (size-of-pile game-state 1)
  (- (size-of-pile game-state 2)
    n-coins))))))

```

5.5 Repräsentationswechsel

Dieses Programm ist recht fehleranfällig, da Spielstände und -züge u.U. nicht durch bloßes Anschauen unterscheidbar sind. Ist (1 . 5) ein Spielstand, bei dem auf dem ersten Haufen eine und auf dem zweiten Haufen fünf Münzen liegen, oder ist es ein Spielzug, der instruiert, fünf Münzen vom ersten Stapel zu nehmen?

Dies ist insbesondere kritisch bei `apply-move`. Was passiert, wenn beim Aufruf beide Parameter versehentlich vertauscht werden?

```

(define play-nim
  (lambda (game-state player)
    (display-game-state game-state)
    (if (over? game-state)
        (announce-winner game-state player)
        (let ((move (next-move game-state player)))
          (announce-move player move)
          (play-nim (apply-move game-state move) (other-player player))))))

```

Eine Fehlermeldung erscheint da nicht, zumindest nicht sofort:

```

> (play-nim (make-game-state 2 1) 'human)
  Pile 1: 2
  Pile 2: 1
Which pile will you remove from?
1
How many coins do you want to remove?
1
You take 1 coins from pile 1.
  Pile 1: 1
  Pile 2: 0
I take 1 coins from pile 1.
  Pile 1: 1
  Pile 2: 1

```

Möglicherweise sind Paare nicht die richtige Repräsentation für Spielstände und -züge. Wie schwierig ist es, die Repräsentation zu wechseln?

```

(define make-game-state
  (lambda (pile-1 pile-2)
    (lambda (pile-number)
      (cond
        ((= pile-number 1)
         pile-1)
        ((= pile-number 2)
         pile-2))))))

(define size-of-pile
  (lambda (game-state pile-number)
    (game-state pile-number)))

```

Die Spielstände sind nun durch Prozeduren repräsentiert, und damit nicht mehr mit Spielzügen zu verwechseln.

Die Änderung hat sich dabei auf nur zwei Prozeduren beschränkt, der Rest des Programms bleibt unverändert. Das ist kein Zufall: Das Prinzip der abstrakten Datentypen ist es, einen Datentyp über das Verhalten seiner Operationen zu schreiben. Daraus folgen die zwei obersten Prinzipien der Datenabstraktion:

1. Für eine neue Typ Wert, stelle Operationen zur Verfügung, welche es erlauben, die Werte des Typs manipulieren, die aber nicht dessen konkrete Repräsentation verraten.
2. Benutze für die Manipulation der Werte des neuen Typs *ausschließlich* die neuen Operationen, nicht aber mehr die Operationen des Typs, der für die Repräsentation verwendet wurde.

Diese Vorgehensweise erlaubt es, im Nim-Programm durch bloßes Auswechseln zweier Prozeduren die Repräsentation des Spielstands zu verändern. In der Terminologie der abstrakten Datentypen ist `make-game-state` ein Konstruktor, und `size-of-pile` ein Selektor. In der praktischen Programmierung kommt hoffentlich noch ein *Prädikat* hinzu, das Elemente des neuen Datentyps von anderen Werten unterscheidet. In Scheme sind dafür `null?`, `pair?` und `number?` Beispiele.

5.6 Abstraktionen für Datenabstraktion

Auf Dauer ist es allerdings unpraktisch, sich für jeden einzelnen neuen Datentyp, der mit einem anderen verwechselt werden könne, sich eine neue Repräsentation einfällen zu lassen. Vielleicht läßt sich diese Erzeugung neuer Repräsentationen irgendwie automatisieren.

```
(define make-type
  (lambda (name)
    name))

(define make-typed-object
  (lambda (type value)
    (cons type value)))

(define typed-object-value
  (lambda (type object)
    (if (equal? type (car object))
        (cdr object)
        (error "type mismatch"))))

(define typed-object-predicate
  (lambda (type)
    (lambda (value)
      (and (pair? value)
           (equal? type (car value))))))
```

Mit der Hilfe dieses Mechanismus lassen sich Spielstände disjunkt repräsentieren:

```
(define game-state-type (make-type 'game-state))

(define make-game-state
  (lambda (pile-1 pile-2)
```

```

(make-typed-object game-state-type
  (cons pile-1 pile-2)))

(define game-state? (typed-object-predicate game-state-type))

(define size-of-pile
  (lambda (game-state pile-number)
    (let ((pair (typed-object-value game-state-type game-state)))
      (if (= pile-number 1)
          (car pair)
          (cdr pair)))))

(define move-type (make-type 'move))

(define make-move
  (lambda (pile n-coins)
    (make-typed-object move-type (cons pile n-coins))))

(define move? (typed-object-predicate move-type))

(define move-pile
  (lambda (move)
    (car (typed-object-value move-type move))))

(define move-n-coins
  (lambda (move)
    (cdr (typed-object-value move-type move))))

```

Das Nim-Programm mit vertauschten Parametern für `apply-move` ist dadurch zwar immer noch nicht korrekt, aber es verweigert wenigstens die Arbeit — eine große Hilfe für den Programmierer:

```

> (play-nim (make-game-state 2 1) 'human)
  Pile 1: 2
  Pile 2: 1
Which pile will you remove from?
1
How many coins do you want to remove?
1
You take 1 coins from pile 1.
type mismatch

```

Bei `make-typed-object` und Freunden fällt auf, daß sie lediglich verhindern, daß Werte verschiedener Typen durcheinanderkommen. Die Repräsentationsfrage wird von `make-typed-object` gar nicht berührt und muß nach wie vor „von Hand“ durchgeführt werden. Die meisten Programmiersprachen haben einheimische Mittel für die Unterscheidung der Werte verschiedener Typen und für die Repräsentation zusammengesetzter Wert, häufig in einem einzelnen Mechanismus vermischt. Beispiele sind Objekte in Java oder Records in C.