Ein Scheme-Primer

Christine Lemmer-Webber und das Spritely Institute,
Übersetzer Florian Pelz

Inhaltsverzeichnis

Dieser Text bietet einen Einstieg in die Scheme-Familie von Programmiersprachen. Er wurde ursprünglich geschrieben, um Leuten zu helfen, die mit der beim Spritely Institute entwickelten Technologie noch nicht vertraut sind. Trotzdem wurde er so allgemein gestaltet, dass jeder diesen Primer lesen kann, der sich für Scheme interessiert.

1. Einführung

In der ganzen Welt der Computer-Programmierung gibt es nur wenige Sprachen, die so einfach, klar, umfassend, mächtig und erweiterbar sind wie Scheme. Die Einführung am Anfang der R5RS-Ausgabe von Schemes Standardisierung[1] erklärt die Philosophie dahinter gut:

> Programmiersprachen sollten nicht entworfen werden, indem man Funktionalitäten über Funktionalitäten schichtet, sondern indem man Schwächen und Einschränkungen entfernt, die es so erscheinen lassen, als wären weitere Funktionalitäten nötig.

Durch diesen Minimalismus sind die Grundlagen von Scheme leicht zu erlernen. Die R5RS-Einführung geht weiter mit:

1 R5RS steht für „Revised(5) Report on the Algorithmic Language Scheme“. Das ist nicht die einzige Auflage, die es von Scheme gibt, aber es ist besonders klein, klar und minimal. Vielleicht ist es sogar auf eine Art zu klein: Es ist allgemein bekannt, dass Implementierungen von R5RS-Scheme miteinander inkompatibel sind, wenn man sich die damit tatsächlich genutzten Bibliotheken anschaut. Das hat nicht zuletzt den Grund, dass R5RS überhaupt kein System festlegt, wie man Bibliotheken schreibt! Allerdings ist R5RS kurz und leicht zu lesen. Wenn Ihnen der Primer hier gefällt, empfehlen wir, dass Sie auch entweder R5RS lesen (inoffizielle deutsche Übersetzung), oder R7RS-small, was eine nur wenig längere, miteinander kompatible Fassung des Reports zu Scheme darstellt.

> Scheme demonstriert, dass eine sehr kleine Zahl von Regeln, wie Ausdrücke gebildet werden, ohne Einschränkungen, wie man sie zusammenfügt, genügen, um eine praktische und effiziente Programmiersprache zu bilden, die flexibel genug ist, die meisten größeren Programmierparadigmen, die heute genutzt werden, zu bedienen.

Mit nur einer kleinen Anzahl Regeln und unglaublich einfacher Syntax ist es in Scheme möglich, jedes Sprachparadigma aufzugreifen, das Ihnen einfällt. Wegen der minimalistischen Basis und starken Erweiterbarkeit hat Scheme eine Anhängerschaft und viele Nutzer in der akademischen Forschung zu Sprachen gefunden.

Doch mit einem guten Ruf in einem Bereich geht ein schlechter in einem anderen einher. Weil man Scheme als akademische Sprache auffasst, wird gerne behauptet, es sei zu schwer, als dass „normale“ Programmierer es sich aneignen sollten.

In Wirklichkeit ist an Scheme gar nichts schwierig. Jeder Programmierer kann es in sehr kurzer Zeit erlernen. (Tatsächlich gilt das erst recht für Kinder und Nichtprogrammierer.)[2]

Beim Spritely Institute haben wir beschlossen, unsere Kerntechnologie, Spritely Goblins, auf Scheme aufzubauen. Es hat sich für uns herausgestellt, dass es zwar exzellente tiefgehende Literatur zu Scheme gibt und auch ein paar einfache, kurze Scheme-Anleitungen, aber eine Einführung auf mittlerem Niveau fehlt. Der folgende Text stellt daher ein Mittelding zwischen einem kurzen und einem umfassenden Überblick von Scheme dar. Um mit Scheme produktiv loslegen zu können, genügt es, den weiteren Text grob zu lesen, aber enthusiastische Leser bekommen auch die Hintergründe erklärt (besonders in den Fußnoten).

Wir machen keine Annahmen über bisherige Programmierkenntnisse, dennoch werden erfahrene Leser schneller durch die ersten Kapitel durchkommen. Je weiter diese Anleitung voranschreitet, desto fortgeschrittener werden die Themen. Zum Schluss gibt es ein besonderes Spektakel: Wir sehen, wie man einen Scheme-Interpetierer in Scheme schreibt, in nur 30 Code-Zeilen.

2. Einrichten

Sie werden sich entscheiden müssen, mit welcher Scheme-Implementierung Sie arbeiten möchten, und auch mit welchem Editor. Es gibt für beides viele Möglichkeiten, aber wir machen hier nur zwei Vorschläge:

- Guile Scheme + GNU Emacs + Geiser: Damit wurde diese Anleitung verfasst. Diese Wahl steht für mächtige Werkzeuge (und anschließend können Sie damit an Code für Guix

2 Im Allgemeinen kümmern sich die Texteditoren, die Programmierer von Scheme/Lisp verwenden, um die Klammern in deren Code. Meistens lesen sie den Code anhand seiner Einrückung und nicht anhand der Gruppierung durch Klammern. Anders gesagt verbringen Lisp-Programmierer ihre Zeit gar nicht damit, über Klammerung nachzudenken.

Weil aber die Syntax der meisten Programmiersprachen Klammerung *nicht* so allumfassend benutzt, fällt es Programmierern mit Kenntnissen in anderen Sprachen als Lisp teils schwer, mit dem geklammerten Stil in Lisps Syntax klarzukommen. (Dagegen ist es sogar leichter für Schüler ganz ohne Programmiererfahrung, wenn sie diese traditionelle Lisp-Syntax erlernen.)

Als wir Workshops veranstaltet haben, die zeigen, wie man programmiert, haben wir die Erfahrung gemacht, dass die, die zum ersten Mal Programmieren lernen, die Syntax von Lisp nicht für abschreckend halten, während Programmierern mit Vorkenntnissen in den meisten anderen Sprachen die Lisp-Syntax erst einmal fremd vorkommt. Wir haben sogar die Erfahrung gemacht, dass es beim parallelen Unterrichten von Scheme (in Form von Racket) und Python vielen Lernenden ohne jeglichen Programmierhintergrund (die Workshops richteten sich an Studenten mit geisteswissenschaftlichem Hintergrund) wesentlich lieber war, mit geklammerter Lisp-Syntax zu arbeiten, die sie verständlicher fanden und wo ihnen die Fehlersuche leichter fiel, vorausgesetzt der Texteditor unterstützt sie. (In Racket ist diese Unterstützung in deren einsteigerfreundlicher Entwicklungsumgebung, DrRacket, vorhanden.) Sie können mehr über dieses Phänomen im Vortrag Lisp but Beautiful; Lisp for Everyone hören.

arbeiten, einem der interessantesten Scheme-Projekte überhaupt). Andererseits ist die Lernkurve hier auch beachtlich.

- Racket, das eine eingebaute Entwicklungsumgebung („IDE“) namens DrRacket mitbringt. Es ist für den Anfang die leichtere Wahl.

Am Anfang mancher Code-Beispiele schreiben wir `REPL>`. *REPL* steht für „Read Eval Print Loop“ (Lese-Auswerten-Schreiben-Schleife). Damit ist eine interaktive Scheme-Eingabeaufforderung gemeint, auf der man Ausdrücke, englisch *expressions*, probeweise eintippen kann.

3. Hallo Scheme!

Hier sehen Sie das vertraute Hallo-Welt-Programm, geschrieben in Scheme:

```
(display "Hallo Welt!\n")
```

So wird „Hallo Welt!“ auf dem Bildschirm angezeigt. (Das `"\n"` steht für eine neue Zeile, wie wenn Sie die Eingabeteste drücken, nachdem Sie Text in ein Textverarbeitungsprogramm eingetippt haben.)

Wenn Sie sich schon mit anderen Programmiersprachen auskennen, sieht das vielleicht ein bisschen vertraut und doch ein wenig anders aus. In den meisten anderen Programmiersprachen würde es etwa so geschrieben:

```
display("Hallo Welt!\n")
```

In diesem Sinne ist das Aufrufen von Funktionen in Scheme (und anderen ähnlichen Lisps) nicht viel anders als in den anderen Sprachen, bis auf dass der Name innerhalb der Klammern steht.

4. Grunddatentypen, ein paar kurze Funktionen

Nicht so wie in manchen anderen Sprachen sind mathematische Ausdrücke wie `+` und `-` Präfix-Funktionen wie andere Funktionen auch, daher stehen sie vorne:

```
(+ 1 2)         ; => 3
(/ 10 2)        ; => 5
(/ 2 3)         ; => 2/3
```

Die meisten davon können mehrere Argumente nehmen:

```
(+ 1 8 10)   ; äquivalent zu "1 + 8 + 10" in Infix-Notation
```

Prozeduren können auch verschachtelt werden. Mit der „Substitutionsmethode“ können wir ermitteln, wozu sie vereinfacht werden:

```
(* (- 8 (/ 30 5)) 21)   ; Anfangsausdruck
(* (- 8 6) 21)          ; vereinfachen: (/ 30 5) => 6
(* 2 21)                ; vereinfachen: (- 8 6)  => 2
42                      ; vereinfachen: (* 2 21) => 42
```

Eine Vielzahl von Typen wird unterstützt. Hier zum Beispiel sind ein paar mathematische Typen:

```
42          ; Ganze Zahlen, "integer"
98.6        ; Gleitkommazahlen, "floating point"
2/3         ; Brüche bzw. "rationale" Zahlen
-42         ; sie können alle auch negativ sein
```

Weil Scheme einerseits mit „exakten“ Zahlen wie Integern und Brüchen umgehen kann, ohne Einschränkungen bezüglich der Größe der Zahlen, ist es sehr gut geeignet für genaues

wissenschaftliches und mathematisches Rechnen. Die Gleitkommadarstellung wird als „inexakt“ aufgefasst; wir verzichten dort auf Genauigkeit zu Gunsten der Geschwindigkeit.

Hier sind noch ein paar Typen:

```
#t                            ; "boolean" für wahr/true
#f                            ; "boolean" für falsch/false
"Pangalaktischer Donnergurgler"; String (Text, also eine Zeichenkette)
'foo                          ; Symbol
'(1 2 3)                      ; eine Liste (von Zahlen, in diesem Fall)
(lambda (x) (* x 2))          ; Prozedur (dazu später mehr)
'(lambda (x) (* x 2))         ; eine Liste von Listen, Symbolen und Zahlen
```

Symbole sind der vielleicht seltsamste Typ, wenn Sie andere Programmiersprachen als Lisp schon kennen (mit ein paar Ausnahmen). Während Symbole nach Strings aussehen, ist ihre Bedeutung eher programmiertechnischer Natur. (In der Syntax für Methoden in *Goblins* bezeichnen wir Methoden mit Symbolen als Methodenname.) Interessanterweise können wir Ausdrücke in Lisp maskieren beziehungsweise quotieren mit `'`, zum Beispiel gelten oben im quotierten `lambda`-Ausdruck die Symbole im Innern auch automatisch als quotiert.

Wir werden uns für Listen Zeit nehmen, wenn wir in Listen und "cons" angekommen sind. Die Kombination von Listen und Symbolen spielt in vielen Lisps eine große Rolle, so auch in Scheme, denn darin liegt Schemes Erweiterbarkeit begründet: Code kann Code schreiben. Auf diese Stärke und wie man sie benutzt werden wir näher eingehen in Über die Erweiterbarkeit von Scheme (und Lisp generell).

5. Variable und Prozeduren

Variablen können wir Werte zuweisen, indem wir `define` verwenden:

```
REPL> (define name "Jane")
REPL> (string-append "Hallo " name "!")
; => "Hallo Jane!"
```

Wenn wir das, was auf `define` folgt, in Klammern einpacken, interpretiert Scheme das Ganze als eine Prozedurdefinition:

```
(define (greet name)
  (string-append "Hallo " name "!"))
```

Jetzt, wo die Prozedur einen Namen hat, können wir sie aufrufen:

```
REPL> (greet "Samantha")
; => "Hallo Samantha!"
```

Wie Sie merken, sind in *Scheme* Return-Anweisungen *implizit*. Das heißt, weil es der Ausdruck am Ende der Prozedur ist, wird das Ergebnis vom `string-append` automatisch auch das zurückgelieferte Ergebnis an den Aufrufer.

Eigentlich ist diese zweite Syntax für `define` bloß *syntaktischer Zucker*. Diese beiden Definitionen von `greet` sind genau das Gleiche.

```
(define (greet name)
  (string-append "Hallo " name "!"))

(define greet
  (lambda (name)
    (string-append "Hallo " name "!")))
```

`lambda` ist die Bezeichnung für eine „anonyme Prozedur“ (sie hat also keinen Namen). Obwohl wir ihr den Namen `greet` zuweisen, könnte man die Prozedur auch ohne einen Namen benutzen:

```
REPL> ((lambda (name)
         (string-append "Hallo " name "!"))
       "Horatio")
; => "Hallo Horatio!"
```

Es gibt auch andere Möglichkeiten, Dingen einen Namen zu geben, außer mit `define`, nämlich mit `let`. Dafür gibt man eine Reihe von Variablen an, die gebunden werden, und danach gibt man einen Rumpf an, der mit diesen Bindungen ausgewertet wird. `let` hat die Form:

```
(let ((<VARIABLENNAME> <WERT-AUSDRUCK>) ...)
  <RUMPF> ...)
```

(Mit `...` meinen wir in diesem Beispiel, dass der vorherige Ausdruck wiederholt angegeben werden darf.)

Hier sehen wir ein Beispiel, wie `let` benutzt wird:

```
REPL> (let ((name "Horatio"))
        (string-append "Hallo " name "!"))
; => "Hallo Horatio!"
```

Schlauen Lesern mag auffallen, dass das sehr ähnlich aussieht wie im letzten Beispiel, und ja, auch `let` ist *Syntaxzucker* für ein lambda, das wir sofort und mit Argumenten anwenden. Die letzten beiden Code-Beispiele sind völlig äquivalent:

```
REPL> (let ((name "Horatio"))
        (string-append "Hallo " name "!"))
; => "Hallo Horatio!"
REPL> ((lambda (name)
         (string-append "Hallo " name "!"))
       "Horatio")
; => "Hallo Horatio!"
```

`let*` ist wie `let`, allerdings dürfen sich Bindungen auf vorherige Bindungen innerhalb des Ausdrucks beziehen:[3]

```
REPL> (let* ((name "Horatio")
             (greeting
              (string-append "Hallo " name "!\n")))
        (display greeting))   ; greeting auf dem Bildschirm anzeigen
; zeigt an: Hallo Horatio!
```

Es ist möglich, eine Liste von Argumenten manuell auf eine Prozedur anzuwenden, mit `apply`. Zum Beispiel können wir die Summe über eine Liste von Zahlen bilden, indem wir `apply` mit `+` kombinieren.

```
REPL> (apply + '(1 2 5))
; => 8
```

Umgekehrt lässt sich eine Argumentliste variabler Länge in Punkt-Notation beschreiben.[4] Das

3 Außerdem gibt es noch `letrec`, womit sich Bindungen rekursiv aufeinander (oder sich selbst) beziehen können. Sowohl `let*` als auch `letrec` können theoretisch zusätzliche Ressourcen beanspruchen, die nicht unbedingt nötig wären (sogenannten Overhead), aber fortschrittliche Compiler stellen fest, wenn sie zu `let` äquivalent sind, und optimieren entsprechend. Wenn Scheme heute von Grund auf neu entworfen würde, würde man vielleicht nur eine Art `let` haben, die sowohl `let*` als auch `letrec` in sich aufnimmt. Doch Geschichte lässt sich nicht ändern.

4 Diese Notation hat direkt zu tun mit dem Aufbau von `cons`-Zellen, worauf wir in Listen und "cons" näher eingehen.

zeigen wir nun zugleich mit Guiles `format`-Prozedur. Wir können sie mit `#f` als erstem Argument aufrufen, so dass eine formatierte Zeichenkette als Wert zurückgeliefert wird, aber mit `#t` als erstem Argument wird sie stattdessen auf dem Bildschirm angezeigt. Letzteres brauchen wir hier:

```
REPL> (define (geschwätziges-add geschwätziger-name . nums)
        (format #t "<~a> Wenn Sie die addieren, kommt ~a raus!\n"
                geschwätziger-name (apply + nums)))
REPL> (geschwätziges-add "Chester" 2 4 8 6)
; Zeigt an:
;   <Chester> Wenn Sie die addieren, kommt 20 raus!
```

Auch wenn sie nicht zum Standard von Scheme gehören, unterstützen viele Scheme-Implementierungen auch optionale Argumente sowie Schlüsselwort-Argumente. Guile implementiert diese Abstraktion als `define*`:

```
REPL> (define* (ladenbesitzer artikel-den-du-kaufst
                              #:optional (wie-viele 1)
                              (preis 20)
                              #:key (ladenbesitzer "Michael")
                              (laden "Bands Frischeladen"))
        (format #t "Du gehst in ~a, nimmst dir was von den Regalen\n"
                laden)
        (display "und gehst zur Kasse.\n\n")
        (format #t "~a schaut dich an und sagt: "
                ladenbesitzer)
        (format #t "'~a ~a also? Das macht ~a Münzen!'\n"
                wie-viele artikel-den-du-kaufst
                (* preis wie-viele)))
REPL> (ladenbesitzer "Äpfel")
; Zeigt an:
;   Du gehst in Bands Frischeladen, nimmst dir was von den Regalen
;   und gehst zur Kasse.
;
;   Michael schaut dich an und sagt: '1 Äpfel also? Das macht 20 Münzen!'
REPL> (ladenbesitzer "Bananen" 10 28)
; Zeigt an:
;   Du gehst in Bands Frischeladen, nimmst dir was von den Regalen
;   und gehst zur Kasse.
;
;   Michael schaut dich an und sagt: '10 Bananen also? Das macht 280 Münzen!'
REPL> (ladenbesitzer "Schrauben" 3 2
                     #:ladenbesitzer "Horatio"
                     #:laden "Horatios Baumarkt")
; Zeigt an:
;   Du gehst in Horatios Baumarkt, nimmst dir was von den Regalen
;   und gehst zur Kasse.
;
;   Horatio schaut dich an und sagt: '3 Schrauben also? Das macht 6 Münzen!'
```

Zu guter Letzt können wir Schemes Prozeduren noch etwas anderes Interessantes tun lassen, nämlich können sie mehrere Werte zurückliefern mit … `values`! Aus purer Blödelei wählen wir als Beispiel, wie es wäre, zwei Zahlen simultan zu addieren und zu multiplizieren:

```
REPL> (define (add-and-multiply x y)
        (values (+ x y)
                (* x y)))
REPL> (add-and-multiply 2 8)
; => 10
; => 16
REPL> (define-values (added multiplied)
```

```
        (add-and-multiply 3 10))
REPL> added
; => 13
REPL> multiplied
; => 30
```

Wie Sie sehen können, erfassen wir beide Werte mit `define-values` wie oben gezeigt. (Sie können auch `let-values` oder `call-with-values` verwenden, aber in diesem Abschnitt haben wir genug Syntax gesehen!)

6. Bedingungen und Prädikate

Gelegentlich möchten wir prüfen, ob eine Aussage wahr ist oder falsch. Zum Beispiel können wir nachsehen, ob es sich bei einem Objekt um einen String handelt, indem wir `string?` fragen:[5]

```
REPL> (string? "Apfel")
; => #t
REPL> (string? 128)
; => #f
REPL> (string? 'Apfel)
; => #f
```

(Denken Sie daran, dass `#t` für "true" steht und `#f` für "false".)

Wir können das zusammen mit `if` einsetzen. Es hat die Form:

```
(if <TEST>
    <FOLGERUNG>
    [<ALTERNATIVE>])
```

(Die eckigen Klammern um `<ALTERNATIVE>` bedeuten, dass man sie weglassen darf.)[6]

Wir können also eine ulkige Funktion schreiben, die begeistert darüber Auskunft gibt, ob ein an sie übergebenes Objekt eine Zeichenkette ist oder nicht:

```
REPL> (define (string-enthusiast obj)
        (if (string? obj)
            "Große Freude, das ist EIN STRING!!!"
            "Das WAR JA GAR KEIN STRING!! STRINGS BITTE!"))
REPL> (string-enthusiast "Karotte")
```

5 Auf Wahrheit / Falschheit testende Prozeduren heißen in Scheme *Prädikate*. Der Ausdruck verwirrt manche, weil er bei natürlichen Sprachen etwas anderes bedeutet und weil *Prädikate* in der Mathematik etwas über Relationen zueinander aussagen. Technisch gesehen sagt ein Test, der einen booleschen Wert zurückliefert, durchaus etwas aus über eben diesen Test, also ist der Begriff nicht falsch, aber je nach Hintergrund nicht intuitiv.
Schemes Prädikaten gibt man traditionell einen Namen mit `?` als Suffix. Auf Englisch wird das `?` gerne als „huh?“ gesprochen, daher „string-huh?“. Einige andere Lisps hängen auf dieselbe Weise `-p` als Suffix an; in Scheme nimmt man `?`.

6 Folgt man dem Scheme-Standard, ist die `<ALTERNATIVE>` technisch gesehen optional. Doch viele Schemes bieten eine separate Prozedur namens `when` an mit folgender Form:

```
(when <TEST>
  <RUMPF> ...)
```

In *Racket* ist ein `if` ohne `<ALTERNATIVE>` (man sagt „einbeiniges `if=“) verboten und selbst wenn man kein /Racket/ benutzt, wird =when` da vorgezogen. Mit hinein spielt, dass es in *rein funktionalen Programmiersprachen* gar keine Bedingung geben darf, bei der womöglich gar nichts Wertvolles zurückgeliefert wird. Anders gesagt ergibt `when` (oder ein „einbeiniges =if=“) nur Sinn, wenn man sich für einen *Side Effect*, also eine Nebenwirkung, interessiert. Es ist gut, wenn man sich des Unterschieds bewusst ist. Wir kommen später auf `when` zurück und wie man es selbst schreibt, in Über die Erweiterbarkeit von Scheme (und Lisp generell).

```
; => "Große Freude, das ist EIN STRING!!!"
REPL> (string-enthusiast 529)
; => "Das WAR JA GAR KEIN STRING!! STRINGS BITTE!"
```

Wie wir sehen, liefert `if`, anders als in manchen anderen geläufigen Sprachen, auch gleich den Wert als Ergebnis zurück, der sich ergibt, wenn je nach `<TEST>` der eine oder andere Zweig ausgewertet wird.

In Scheme stehen von Anfang an bestimmte mathematische Vergleiche als Test zur Verfügung. `>` und `<` bedeuten jeweils „größer als“ und „kleiner als“, außerdem stehen `>=` und `<=` für „größer als oder gleich“ und „kleiner als oder gleich“, wohingegen man mit `=` auf numerische Gleichheit hin prüft:[7]

```
REPL> (> 8 9)
; => #f
REPL> (< 8 9)
; => #t
REPL> (> 8 8)
; => #f
REPL> (>= 8 8)
; => #t
```

Wenn wir mehrere Möglichkeiten testen wollten, sind verschachtelte `if`-Anweisungen möglich:

```
REPL> (define (goldlöckchen n smallest-ok biggest-ok)
        (if (< n smallest-ok)
            "Zu klein!"
            (if (> n biggest-ok)
                "Zu groß!"
                "Genau richtig!")))
REPL> (goldlöckchen 3 10 20)
; => "Zu klein!"
REPL> (goldlöckchen 33 10 20)
; => "Zu groß!"
REPL> (goldlöckchen 12 10 20)
; => "Genau richtig!"
```

Jedoch sieht es gleich viel hübscher aus, wenn wir stattdessen die Syntax namens `cond` benutzen.

7 Zugegeben, Lisps Präfix-Notation ist hier weniger angemessen als Infix-Notation, denn hinter dem Aussehen der spitzen Klammern steckt die visuelle Notation der Idee, dass eines kleiner und eines größer ist. Manche Schemes unterstützen SRFI-105: Geschweifte Infix-Ausdrücke, was leichter zu lesen ist. Zum Vergleich:

```
REPL> (> 8 9)
; => #f
REPL> (< 8 9)
; => #t
REPL> (> 8 8)
; => #f
REPL> (>= 8 8)
; => #t
```

Gegenüber:

```
REPL> {8 > 9}
; => #f
REPL> {8 < 9}
; => #t
REPL> {8 > 8}
; => #f
REPL> {8 >= 8}
; => #t
```

Sie hat die folgende Form:[8]

```
(cond
 (<TEST>
  <FOLGERUNG-RUMPF> ...) ...
 [(else <ELSE-RUMPF> ...)])
```

Vergleichen Sie, wie wir unsere Prozedur `goldlöckchen` gleich viel lieber mögen mit `cond` anstelle von verschachtelten `if`-Anweisungen:

```
;; Version mit verschachtelten "if"
(define (goldlöckchen n smallest-ok biggest-ok)
  (if (< n smallest-ok)
      "Zu klein!"
      (if (> n biggest-ok)
          "Zu groß!"
          "Genau richtig!")))

;; "cond"-Version
(define (goldlöckchen n smallest-ok biggest-ok)
  (cond
   ((< n smallest-ok)
    "Zu klein!")
   ((> n biggest-ok)
    "Zu groß!")
   (else
    "Genau richtig!")))
```

Außerdem bietet Scheme mehrere verschiedene Vergleiche, ob zwei Objekte gleich sind oder nicht. Am kürzesten und einfachsten (aber nicht vollständig) kann man den Zoo der Gleichheitsprädikate so zusammenfassen: `equal?` vergleicht den Inhalt, während `eq?` die Identität der Objekte vergleicht (in dem Sinn, wie es die Laufzeitumgebung der Sprache festlegt).[9] Zum Beispiel legt `list` eine neue Liste jedes Mal mit neuer Identität an, deshalb sind folgende Objekte `equal?`, aber nicht `eq?`:

```
REPL> (define a-list (list 1 2 3))
REPL> (define b-list (list 1 2 3))
REPL> (equal? a-list a-list)
; => #t
REPL> (eq? a-list a-list)
; => #t
REPL> (equal? a-list b-list)
; => #t
REPL> (eq? a-list b-list)
; => #f
```

Abschließend gilt in Scheme, dass alles, was nicht `#f` ist, als true aufgefasst wird. Das verwendet man manchmal etwa bei `member`. `member` such nach passenden Elementen und liefert die verbleibende Liste zurück, wenn etwas Gleiches gefunden wurde, und sonst `#f`:

```
REPL> (member 'b '(a b c))
; => (b c)
```

8 Wie wir noch sehen werden in Über die Erweiterbarkeit von Scheme (und Lisp generell), erlaubt Scheme es uns, neue Formen von Syntax zu implementieren. Für eine Scheme-Implementierung genügt eine grundlegende Form von Syntax, denn `if` kann man als vereinfachte Version von `cond` implementieren und `cond` als verschachtelte Abfolge von `if`-Anweisungen.

9 Die größte Untertreibung in den Fußnoten der Informatik findet sich in The Art of the Propagator von Gerald Jay Sussman und Alexey Radul, wo es schlicht heißt:
> Gleichheit ist ein schwieriges Thema.

```
REPL> (member 'z '(a b c))
; => #f
REPL> (define (obstschnüffler obst korb)
        (if (member obst korb)
            "Das gesuchte Obst ist im Korb!"
            "Ich kann das Obst nicht finden! Verdammich!"))
REPL> (define obstkorb '(apfel banane zitrone))
REPL> (obstschnüffler 'banane obstkorb)
; => "Das gesuchte Obst ist im Korb!"
REPL> (obstschnüffler 'ananas obstkorb)
; => "Ich kann das Obst nicht finden! Verdammich!"
```

7. Listen und "cons"

```
"My other CAR is a CDR"
  — Autoaufkleber eines Lisp-Anhängers
```

Für strukturierte Daten verwendet man in Scheme Listen, die jeden anderen Typ enthalten können.[10] Wir zeigen nun zwei Schreibweisen für dieselbe Liste:

```
REPL> (list 1 2 "Miezekatze" 33.8 'foo)
; => (1 2 "Miezekatze" 33.8 foo)
REPL> '(1 2 "Miezekatze" 33.8 foo)
; => (1 2 "Miezekatze" 33.8 foo)
```

Es besteht dazwischen ein Unterschied, nämlich musste im quotierten Beispiel das Symbol „foo“ nicht quotiert werden, weil es durch die Quotierung der äußeren Liste implizit mitquotiert wird.

Wir haben eine „besondere“ Liste, bekannt als „die leere Liste“. Es ist eine Liste ohne Elemente und wir schreiben sie einfach als `'()`. (Man kennt sie auch als *nil*. Sie ist das einzige Objekt, für das das Prädikat `null?` laut dem Standard von Scheme `#t` zurückliefert.) Listen in *Scheme* sind tatsächlich „verkettete Listen“. Das sind Kombinationen aus Paaren, die jeweils „cons-Zelle“ genannt werden. Am Ende der Paarabfolge steht die leere Liste:

```
REPL> '()
; => ()
REPL> (cons 'a '())
; => (a)
REPL> (cons 'a (cons 'b (cons 'c '())))
; => (a b c)
```

Äquivalent können wir eines hiervon schreiben:

```
REPL> (list 'a 'b 'c)
; => (a b c)
REPL> '(a b c)
; => (a b c)
```

Aus lang historischen Gründen[11] greift man auf das erste Element einer cons-Zelle mittels `car` zu

10 *Scheme* hat auch eingebaute Unterstützung für *Vektoren*. Die sehen erstmal aus wie Listen, haben aber den Vorteil, dass Zugriff in konstanter Zeit möglich ist, jedoch taugen sie weniger für funktionale Programmierung, denn man kann keine neuen Elemente vorne d’ranhängen. Es gibt in vielen Scheme-Sprachen noch andere interessante Datentypen wie Hashmaps und benutzerdefinierte Verbünde (Records).

11 Der Name `cons` ist sinnvoll; er bedeutet, dass man ein Paar konstruiert/construct. Aber der Grund für die Namen `car` und `cdr` ist ein rein historisches Detail der ersten Lisp-Implementierung. `car` bedeutet „contents of the address register“ und `cdr` bedeutet „contents of the decrement register“. Beeindruckend, wie lange solche Begriffe überdauern, im Guten wie im Schlechten.

Mehr interessante Lisp-Geschichte finden Sie hier:

- History of Lisp von John McCarthy

und auf das zweite Element einer cons-Zelle mit `cdr` (ausgesprochen „kudder“):[12]

```
REPL> (car '(a b c))
; => a
REPL> (cdr '(a b c))
; => (b c)
REPL> (car (cdr '(a b c)))
; => b
```

Das zweite Element in `cons` muss keine andere cons-Zelle oder die leere Liste sein. Wenn wir etwas anderes hineintun, haben wir eine „gepunktete Liste“. Diese schreibt sich in einer für Lisp ungewöhnlichen Infix-Syntax:

```
REPL> (cons 'a 'b)
; => (a . b)
```

Sie sollten den strukturellen Unterschied hierzu erkennen:

```
REPL> (cons 'a (cons 'b '()))
; => (a b)
```

Passt man nicht auf, verbringt man seine Zeit mit dem Auseinanderklamüsern von `cons`-Zellen (wovon es heißt, dass Schemer das viel zu oft täten, aber `cons` ist eben elegant und mächtig).[13]

In gewissem Sinn schweifen wir in diesem Unterabschnitt bloß ab. In unserem Paper vermeiden wir `cons` mit Absicht und erwähnen `car` und `cdr` gleich gar nicht im Haupttext. Warum erwähnen wir Listen dann in einem eigenen Unterabschnitt?

Das liegt daran, dass wir auf ein wichtiges Thema hinarbeiten, nämlich wie leicht sich Scheme erweitern lässt. Scheme ist in seinen Kerndatentypen geschrieben worden und mit ihnen kann man die Sprache auch verändern. Bald sagen wir mehr darüber, aber zunächst bringen wir ein Beispiel, dass man jede Art Ausdruck quotieren kann und so aus Code Daten macht:

```
REPL> (+ 1 2 (- 8 4))
; => 7
REPL> '(+ 1 2 (- 8 4))
; => (+ 1 2 (- 8 4))
REPL> (let ((name "Horatio"))
        (string-append "Hallo " name "!"))
; => "Hallo Horatio!"
REPL> '(let ((name "Horatio"))
         (string-append "Hallo " name "!"))
; => (let ((name "Horatio")) (string-append "Hallo " name "!"))
```

- The Evolution of Lisp von Guy L. Steele und Richard P. Gabriel
- History of LISP von Paul McJones

12 Weil `car` und `cdr` nur „historische Details“ sind, mag man auf die Idee kommen, sie durch bessere Namen zu ersetzen. Für jemanden, der nur Listen verwendet, erscheinen `first` und `rest` wie gute alternative Namen:

```
REPL> (first '(a b c))
; => a
REPL> (rest '(a b c))
; => (b c)
REPL> (first (rest '(a b c)))
; => b
```

Das Problem ist, wenn eine cons-Zelle ein einfaches Paar enthält wie `(cons 'a 'b)`, dann geht der Sinn wieder verloren … `rest` liefert nur ein einzelnes Element und keine Folge. So verbleiben wir bei den Namen `car` und `cdr`.

13 Es lässt sich noch viel über `cons` sagen. Im Buch The Little Schemer wird es als prächtig tituliert, „=cons= the magnificent“, und viel mehr auf `cons` eingegangen und darauf, wie man einen intuitiven Sinn für Rekursion entwickelt.

Das Beispiel eben macht uns neugierig: Endlich haben wir den Zweck von Symbolen erkannt, denn mit ihnen kann man die Namen von Funktionen und von Syntax erfassen, wenn man diese quotiert. Insofern kann man sagen, Lisp (Scheme eingeschlossen) ist in Lisp geschrieben: Es gibt kaum einen Unterschied zwischen der Darstellung, die der Programmierer sieht, und der für den Compiler. Mehr dazu in Über die Erweiterbarkeit von Scheme (und Lisp generell).

Es sei noch gesagt, wenn wir mit einem Apostroph quotieren, ist das eigentlich nur eine Kurzschreibweise für `(quote <AUSDRUCK>)`:

```
;; die beiden sind dasselbe
'foo
(quote foo)

;; auch die beiden sind dasselbe
'(lambda (x) (* x 2))
(quote (lambda (x) (* x 2)))
```

Mit Listen kann man auch eine assoziative Abbildung zwischen Schlüsseln und Werten darstellen. Das nennen wir *Alists* (assoziative Listen). Es gibt mehrere Prozeduren, um den Wert zu einem Schlüssel bequem nachzuschauen, darunter `assoc`, welche das Paar mit ihm liefert, wenn es enthalten ist, und sonst `#f`:

```
REPL> (define tierlaute
        '((katze . miau)
          (hund . wuff)
          (schaf . bäh)))
REPL> (assoc 'katze tierlaute)
; => (katze . miau)
REPL> (assoc 'alien tierlaute)
; => #f
```

Assoziative Listen lassen sich leicht implementieren, schauen ansehnlich genug aus in der gedruckten Darstellung von Scheme und sind geeignet für funktionale Programmierung. (Sie möchten etwas in eine Alist einfügen? cons einfach noch eine cons-Zelle d'rauf!) Also sind sie beliebt bei Schemern. Trotzdem sind sie oftmals nicht effizient. Zwar ist `assoc` bei kurzen assoziativen Listen gut, aber wenn die assoziative Liste eintausend Elemente lang ist, dauert es auch tausend Schritte, ein am Ende vergrabenes Schlüssel-Wert-Paar nachzuschlagen. Scheme-Implementierungen haben für gewöhnlich andere Datenstrukturen wie Hashmaps auf Lager, die im Schnitt in konstanter Zeit nachschlagen lassen. Manchmal sind sie die bessere Wahl.

Neben quote kann man auch quasiquote verwenden, was man mit einem Backtick-Zeichen einleitet und mit einem Komma demaskiert man wieder. Auf diese Weise kann man schnell zwischen den Daten und Code hin- und herschalten. Betrachten wir eine etwas eigenartige Metrik, um Katzenjahre in Menschenjahre umzurechnen:

```
REPL> (define (katzenjahre jahre)
        (cond
         ((<= jahre 1)       ; erstes Jahr macht 15 Menschenjahre
          (* jahre 15))
         ((<= jahre 2)
          (+ 15 (* 9 (- jahre 1))))   ;      zweites Jahr 9
         (else
          (+ 24 (* 4 (- jahre 2)))))) ; spätere Jahre 4
REPL> (define (katze-eintrag name alter)
        `(katze (name ,name)
                (alter  ,alter)
                (katzenjahre-alter ,(katzenjahre alter))))
REPL> (katze-eintrag "Missy Rose" 16)
```

```
; => (katze (name "Missy Rose")
;           (alter 16)
;           (katzenjahre-alter 80))
REPL> (katze-eintrag "Kelsey" 22)
; => (katze (name "Kelsey")
;           (alter 21)
;           (katzenjahre-alter 104))
```

Mensch, sind das alte Katzen!

8. Closures (Abschlüsse)

Erinnern Sie sich zurück, wie wir `goldlöckchen` definiert und benutzt haben:

```
REPL> (define (goldlöckchen n smallest-ok biggest-ok)
        (cond
         ((< n smallest-ok)
          "Zu klein!")
         ((> n biggest-ok)
          "Zu groß!")
         (else
          "Genau richtig!")))
REPL> (goldlöckchen 3 10 20)
; => "Zu klein!"
REPL> (goldlöckchen 33 10 20)
; => "Zu groß!"
REPL> (goldlöckchen 12 10 20)
; => "Genau richtig!"
```

Immer wieder dieselben Werte für `smallest-ok` und `biggest-ok` anzugeben, ist anstrengend. Goldlöckchen hat *nicht* ständig andere Gewohnheiten von Aufruf zu Aufruf. Kann man keine Version von Goldlöckchen mit Gedächtnis entwickeln, damit wir `smallest-ok` und `biggest-ok` nur einmal angeben müssen und trotzdem mehrere Werte von `n` testen? Doch, natürlich kann man. *Closures* (deutsch Abschlüsse) sind die Lösung!

```
(define (konstr-goldlöckchen smallest-ok biggest-ok)
  (define (goldlöckchen n) ; wir haben smallest-ok und
    (cond                  ; biggest-ok in eine äußere
     ((< n smallest-ok)    ; Prozedur eingepackt, damit
      "Zu klein!")         ; nur das n noch als Argument
     ((> n biggest-ok)     ; übergeben werden muss
      "Zu groß!")
     (else
      "Genau richtig!")))
  goldlöckchen)            ; Ergebnis ist goldlöckchen
```

Damit rufen wir `konstr-goldlöckchen` auf und es liefert uns die eingeschlossene `goldlöckchen`-Prozedur.

```
REPL> (konstr-goldlöckchen 10 30)
; => #<procedure goldlöckchen (n)>
```

Jetzt genügt es, das innere `goldlöckchen` wiederholt aufzurufen.

```
REPL> (define goldi
        (konstr-goldlöckchen 10 30))
REPL> (goldi 7)
; => "Zu klein!"
REPL> (goldi 256)
; => "Zu groß!"
```

```
REPL> (goldi 22)
; => "Genau richtig!"
```

Die äußere Prozedur bildet einen Abschluss um die innere Prozedur und gewährt ihr den Zugriff (und Gedächtnis) auf `smallest-ok` und `biggest-ok`.

Wir merken an, dass in *Goblins* nach demselben Muster Konstruktoren für die Objekte geschrieben werden: Die äußere Prozedur ist der Konstruktor, die innere Prozedur das Verhalten des Objekts. (Der Unterschied liegt in erster Linie darin, dass *Goblins*-Objekte, die man mit `spawn` anlegt, eine Capability *bcom* erhalten, mit der sie ihr Verhalten ändern können!)

Es ist schön, wie wir unsere eigene Implementierung von cons-Zellen aus reiner Abstraktion auf dieselbe Weise schreiben können.

```
REPL> (define (abstract-cons car-data cdr-data)
        (lambda (method)
          (cond
           ((eq? method 'car)
            car-data)
           ((eq? method 'cdr)
            cdr-data)
           (else (error "Unbekannte Methode:" method)))))
REPL> (define our-cons (abstract-cons 'foo 'bar))
REPL> (our-cons 'car)
; => foo
REPL> (our-cons 'cdr)
; => bar
```

Das meinen wir, wenn wir Abschlüsse als aus dem Programmfluss selbst bestehende Datenstrukturen beschreiben.

Abgeschlossenheit ergibt sich aus der *lexikalischen Bindung* (Lexical Scope). Wir machen uns diese Eigenschaft in Goblins zunutze: Die Capabilitys, die ein Objekt hat, sind einfach die Capabilitys im Geltungsbereich seines Verhaltens.

9. Iteration und Rekursion

Beim Programmieren geht es oft um Abfolgen von Operationen, besonders im Zusammenhang mit Datenstrukturen, die andere Informationen enthalten. Von besonderem Nutzen in der funktionalen Programmierung ist die Prozedur `map`. Mit ihr wendet man auf jedes Element einer Folge die Prozedur im ersten Argument an.

Ein Beispiel: `string-length` gibt uns die Anzahl von Zeichen, die in einer Zeichenkette enthalten sind:

```
REPL> (string-length "Katze")
; => 5
REPL> (string-length "Gorilla")
; => 7
```

Mit `map` können wir leicht eine Liste konstruieren, die zu jeder Zeichenkette ihre Länge angibt:

```
REPL> (map string-length '("Katze" "Hund" "Gorilla" "Salamander"))
; => (5 4 7 10)
```

Wir können auch eine selbstdefinierte Prozedur angeben:

```
REPL> (define (symbol-length sym)
        (string-length (symbol->string sym)))
REPL> (map symbol-length '(Basilikum Oregano Petersilie Thymian))
```

```
; => (9 7 10 7)
```

Wir müssen der Prozedur eigentlich keinen Namen geben … denn mit `lambda` konstruieren wir ja eine anonyme Prozedur und so eine übergeben wir direkt an `map`:

```
REPL> (map (lambda (str)
             (string-append "Ich liebe "
                            (string-upcase str)
                            " einfach!!!"))
           '("Erdbeeren" "Bananen" "Trauben"))
; => ("Ich liebe ERDBEEREN einfach!!!"
;     "Ich liebe BANANEN einfach!!!"
;     "Ich liebe TRAUBEN einfach!!!")
```

Bei `map` fällt ein bisschen Mehrarbeit an: Mit jedem Mal wird eine Liste von Ergebnissen aufgebaut. Was aber, wenn wir unsere Liebe zu Lebensmitteln einfach nur mit `display` auf dem Bildschirm anzeigen wollten und kein Interesse daran hätten, die Daten weiterzuverarbeiten? Die Antwort lautet `for-each`, das denselben Aufbau hat wie `map`, jedoch kein Ergebnis aufbaut:

```
REPL> (for-each (lambda (str)
                  (display
                   (string-append "Ich liebe "
                                  (string-upcase str)
                                  " einfach!!!\n")))
                '("Eiscreme" "Toffee" "Kekse"))
; zeigt an:
;   Ich liebe EISCREME einfach!!!
;   Ich liebe TOFFEE einfach!!!
;   Ich liebe KEKSE einfach!!!
```

ANMERKUNG! Der folgende Text in diesem Unterabschnitt und eigentlich der gesamten verbleibenden Scheme-Anleitung geht über alle Erfordernisse hinaus, um den Hauptteil unseres Papers „The Heart of Spritely“ zu verstehen! Dennoch trägt er merklich zum Scheme-Verständnis eines neuen Lesers bei.

Eine überraschende Eigenschaft von Scheme ist, dass Iteration in Form von Rekursion definiert ist!

Was wir damit meinen: Wir können unsere eigene Fassung von `for-each` definieren:

```
(define (for-each proc lst)
  (if (eq? lst '())  ; Liste zu Ende?
      'fertig        ; Wir sind fertig und liefern das als Ergebnis
      (let ((item (car lst)))  ; Ansonsten holen wir uns noch ein Item
        (proc item)            ; Rufen die Prozedur auf Item auf
        (for-each proc (cdr lst)))))  ; Und iterieren auf dem Rest
```

Damit rufen wir `proc` nacheinander auf jedem Item auf, bis kein Item mehr übrig ist. Wenn Sie sich schon mit anderen Programmiersprachen auskennen, könnten Sie denken, mit diesem Programmentwurf bekämen wir unter Umständen einen ungewollten Stacküberlauf. Aber Scheme ist eine intelligente Sprache: Es kriegt mit, wenn wir die Version in `for-each` nicht mehr brauchen, nachdem wir in der letzten Zeile angekommen sind. Anders ausgedrückt, wenn `for-each` sich selbst endrekursiv aufruft. Daher braucht Scheme gar nicht erst einen neuen Rahmen auf dem Stack anzulegen, sondern „springt“ wieder zurück an den Anfang von `for-each` mit den neuen Variablen alloziert im Speicher. Das nennt sich *Endrekursionsbeseitigung* und tatsächlich sind in Scheme alle Iterationskonstrukte rekursiv implementiert.

Es ist ebenso möglich, baumrekursive Prozeduren zu entwickeln. Folgende Prozedur (die durchaus etwas für Fortgeschrittene ist; einfach Weiterlesen ist in Ordnung) baut einen Binärbaum:

```
(define (build-tree depth)
  (if (= depth 0)
      '(0)
      (list depth
            (build-tree (- depth 1))
            (build-tree (- depth 1)))))
```

```
REPL> (build-tree 3)
; => (3 (2 (1 (0)
;            (0))
;          (1 (0)
;             (0)))
;       (2 (1 (0)
;             (0))
;          (1 (0)
;             (0))))
```

Oder besser visualisiert:

```
       3
      / \
     /   \
    2     2
   / \   / \
  1  1  1  1
 /\  /\ /\  /\
0 0 0 0 0 0 0 0
```

Anders als `for-each` ruft `build-tree` sich selbst *nicht* nur endständig auf. Es gibt keine Möglichkeit, einfach an den Anfang der Prozedur zu „springen“, ohne sich auf dem Stack zu notieren, was noch zu tun ist, wenn der Code so geschrieben ist: Wir schieben Arbeit auf später auf und `list` muss auf deren Ergebnis warten. `build-tree` ist *nicht* iterativ wie `for-each`, obwohl es rekursiv ist.

Es ist Zeit, über einfachere Schreibweisen nachzudenken. Hier zeigen wir zwei Varianten von `let`, die für sowohl rekursive als auch iterative Prozeduren taugen. Zunächst einmal `letrec`. Damit können sich im `letrec` angegebene Prozeduren selbst oder gegenseitig aufrufen oder aufeinander beziehen, auch wenn die Prozeduren erst weiter unten in der Reihenfolge stehen:

```
REPL> (letrec ((alice
                (lambda (first?)
                  (report-status "Alice" first?)
                  (if first? (bob #f))))
               (bob
                (lambda (first?)
                  (report-status "Bob" first?)
                  (if first? (alice #f))))
               (report-status
                (lambda (name first?)
                  (display
                   (string-append name " ist "
                                  (if first?
                                      "Nummer eins"
                                      "Nummer zwei")
                                  "!\n")))))
        (alice #t)
        (display "-----\n")
        (bob #t))
; zeigt an:
;   Alice ist Nummer eins!
```

```
;   Bob ist Nummer zwei!
;   -----
;   Bob ist Nummer eins!
;   Alice ist Nummer zwei!
```

Die andere hilfreiche Abstraktion ist das *Let mit Namen*, auch eine Variant von `let`, worin als allererstes Argument ein Bezeichner für den Schleifendurchlauf steht:

```
REPL> (let loop ((words '("Karotte" "Kartoffel" "Erbse" "Sellerie"))
                 (num-words 0)
                 (num-chars 0))
        (if (eq? words '())
            (format #f "Das waren ~a Wörter aus ~a Zeichen!"
                    num-words num-chars)
            (loop (cdr words)
                  (+ num-words 1)
                  (+ num-chars (string-length (car words))))))
; => "Das waren 4 Wörter aus 29 Zeichen!"
```

Die Wirkungsweise eines *Let mit Namen* wird über die benannte Prozedur definiert (hier bekommt sie den Namen `loop`) und sie wird dadurch sofort aufgerufen. Anfänglich sind die Variablen wie im `let` gebunden. Später kann die Prozedur im Rumpf des `let` aufgerufen werden, wodurch sich rekursive (vielleicht iterative) Aufrufe leicht schreiben lassen.

10. Veränderungen, Zuweisungen und andere Nebenwirkungen

Dieser Abschnitt steht hier der Vollständigkeit halber. Es sei gesagt, dass Goblins vieles hiervon anders angeht. Den anderen Ansatz diskutieren wir zum Schluss.

Von Haus aus gibt es in Scheme die Möglichkeit, den aktuellen Wert einer Variablen neu zuzuweisen, mit `set!`:

```
REPL> (define truhe 'schwert)
REPL> truhe
; => schwert
REPL> (set! truhe 'gold)
REPL> truhe
; => gold
```

Dies lässt sich kombinieren mit den Techniken aus Closures (Abschlüsse). Zum Beispiel zählt dieses Objekt von einer Startzahl `n` an herunter bis null und ab da wird nur noch null zurückgeliefert.

```
REPL> (define (make-countdown n)
        (lambda ()
          (define last-n n)
          (if (zero? n)
              0
              (begin
                (set! n (- n 1))
                last-n))))
REPL> (define cdown (make-countdown 3))
REPL> (cdown)
; => 3
REPL> (cdown)
; => 2
REPL> (cdown)
; => 1
```

```
REPL> (cdown)
; => 0
REPL> (cdown)
; => 0
```

Mehrere interessante Dinge gibt es in diesem Beispiel:

- Wir haben Zeit und Veränderung in unsere Berechnungen gebracht. Vor der Einführung von Nebenwirkungen, wie Zuweisungen, hatte jeder Aufruf einer Prozedur mit denselben Argumenten immer auch dasselbe Ergebnis gehabt. Aber wie das Beispiel zeigt, ändert sich die Antwort von `cdown` mit der Zeit (sogar ohne dass sie Argumente übergeben bekommt).
- Weil wir beim ersten Aufruf der Prozedur mit der Startzahl loslegen möchten, ist es notwendig, zuerst `last-n` zu behalten, *bevor* wir mit `set!` den Wert von `n` verändern. Wenn wir aus Versehen die Reihenfolge tauschen, hätten wir unbeabsichtigt einen Bug geschaffen und `cdown` hätte im Beispiel ab `2` gezählt statt ab `3`.
- Außerdem kommt eine interessante neue Syntax vor: `begin`. Mit `begin` werden mehrere Ausdrücke in Folge ausgeführt und der Wert des letzten zurückgeliefert.

Letzteres ist wissenswert. Als wir noch keine Wirkungen hatten (wie die Zuweisung dort oben, Anzeigen auf dem Bildschirm, Protokollierung in eine Datei oder Datenbank und so weiter), da gab es nie einen Grund für `begin`. Um das zu verstehen, müssen Sie sich an die *Substitutionsmethode* vom Anfang dieses Tutorials zurückerinnern:

```
(* (- 8 (/ 30 5)) 21)   ; Anfangsausdruck
(* (- 8 6) 21)          ; vereinfachen: (/ 30 5) => 6
(* 2 21)                ; vereinfachen: (- 8 6)  => 2
42                      ; vereinfachen: (* 2 21) => 42
```

Vor Wirkungen hatte jede aufgerufene Prozedur einen neuen Teil des Programms berechnen sollen. Aber weil jeder Zweig des `if` nur einen einzigen Ausdruck auswertet, müssen wir irgendwie die Klausel mit der *Alternative* sequenzieren, dass sie sowohl `set!` durchführt als auch einen Wert zurückliefert.[14] Mit anderen Worten ist der Zweck eines *rein funktionalen Programms* wirklich nur, gegeben eine Reihe von Eingaben stets einen Wert zu berechnen, denselben Wert, jedes Mal. Es ist eine saubere Reihung von Substitutionen von Anfang bis Ende der Auswertung. (Mit anderen Worten bekamen wir vor der Einführung von Zeit gänzlich *deterministische* Programme.)

Mit der Einführung von *Veränderungen* (Mutation) und *Nebenwirkungen* (Side effects) kam ein mächtiges, aber gefährliches, neues Konstrukt in unser Programm: die Zeit. Wir können nicht länger sagen, unsere Programme seien *rein funktional*, denn durch die Zeit sind sie *imperativ* geworden: tu dies, dann tu das. Zeit ist Veränderung und bei Veränderungen spielt eine Rolle, in welche Reihenfolge die Ereignisse sequenziert sind; Substitutionen alleine genügen nicht. Und durch die Zeit werden auch dieselben Programme und Prozeduren, wenn man sie mit denselben Eingaben aufruft, unterschiedliche Ausgaben zur Folge haben können. Wir haben eine zeitlose Welt gegen eine mit Veränderung getauscht.

Trotz aller Mahnung kann Veränderung auch wünschenswert sein. Wir leben in einer Welt mit Zeit und Wandel, meistens genau wie unsere Programme. Die Namensgebung in Scheme berücksichtigt (zumindest in der Regel) Zeit und Veränderlichkeit durch Anhängen von `!` als Suffix, wie wir bei `set!` gesehen haben. Man kann das `!` als eine Art Warnung sehen, wie wenn der Nutzer schreiend auf eine mögliche Veränderlichkeit hinweist. (Allerdings werden durch Schemes Laufzeitumgebung keine Garantien gemacht, dass hinter dem Vorhandensein oder Fehlen dieses Suffixes überhaupt

14 Es sei erwähnt, dass auch in `cond` mehrere Ausdrücke im `<FOLGERUNG-RUMPF>` / `<ELSE-RUMPF>` stehen dürfen, doch können wir das so auslegen, als ob im Innern der Implementierung von `cond` ein `begin` stünde.

etwas von Veränderlichkeit steckt.)

`set!` ist nicht die einzige Form von Veränderung und Mutation im standardisierten Scheme (oder in über den Standard hinausgehenden Schemes).[15] Ein anderes Beispiel bilden veränderliche Vektoren und `vector-set!`:

```
REPL> (define vec (vector 'a 'b 'c))
REPL> vec
; => #(a b c)
REPL> (vector-ref vec 1)
; => b
REPL> (vector-set! vec 1 'boop)
REPL> (vector-ref vec 1)
; => boop
REPL> vec
; => #(a boop c)
```

Die beiden Beispiele machen den Eindruck von Mutation. Aber im Lauf dieser Anleitung haben wir bereits eine andere Art Nebeneffekt kennengelernt, nämlich `display`, das auf den Bildschirm schreibt. Genauer gesagt basiert `display` selbst auf dem Konzept von *Ports*. Ports sind Mechanismen in Scheme, um von Eingabegeräten zu lesen oder auf Ausgabegeräte zu schreiben.

Sie alle stellen uns vor dieselben Schwierigkeiten wie bei `set!`. Einfach ausgedrückt werden, wenn wir die umgebende Zeit in unsere Programme einbringen, unsere Programm zeitabhängiger. Doch es stellt sich als vergeblich heraus, *alle* Zeit oder Veränderung von unseren Rechnern verbannen zu wollen, wie es diese ineinandergelegten Zitate hier darstellen:

> Wie Simon Peyton Jones, ein bekannter funktionaler Programmierer, gerne anmerkt: „Alles, was man noch tun kann, wenn es keine Nebenwirkungen gibt, ist einen Knopf drücken und zuschauen, wie der Kasten eine Zeit lang warm wird.“ (Auch das stimmt nicht ganz, denn auch das Warmwerden des Rechners ist eine Nebenwirkung.)
>
> — Aus *Land of Lisp* von Conrad Barski, M.D.

Worauf Simon und Conrad hinweisen, ist das Dilemma funktionaler Programmierung, dass Nebenwirkungen gefährlich sein können und sich der Nutzer trotzdem eigentlich nur für Nebenwirkunngen interessiert. Irgendwo müssen, damit der Rechner einen Zweck erfüllt, Eingaben vom Nutzer eingelesen und Ausgaben zurückgegeben werden, was von sich aus Nebenwirkungen hat. Und auch wenn man mit der vom geschäftigen Rechner abgestrahlten Hitze sein Haus wärmt, ist das eine Nebenwirkung. Es führt kein Weg daran vorbei, zwischen dem Reich rein mathematischer Utopie und der echten Welt hin- und herzuwechseln.[16]

15 Dass `set!` in vielem quasi eingebaut ist, schafft Probleme in Scheme-Programmen, die sich darin einschränken sollen, was sie anrichten können. Interessierte Leser sollten sich den alternativen Ansatz aus Schemes Variante *W7* in A Security Kernel Based on the Lambda Calculus anschauen.

16 Funktionale Programmierer fangen Nebenwirkungen manchmal in einem cleveren Trick: *Monaden*. Monaden gehören nicht zum Stoff dieser Anleitung, aber man kann sie sich als clevere, explizite Form des Umgangs mit Zeit vorstellen, indem man ein Zustandsbündel durch ein ansonsten zustandsloses Programm fädelt: Zeit existiert, doch sie bleibt deterministisch und der Programmierer wird Herr der Zeit. Der Nachteil ist, dass sich Programmierer damit herumschlagen müssen.

Goblins verfolgt einen anderen Ansatz, den wir in Turns are cheap transactions und Time-travel distributed debugging erörtern. Indem wir die Natur der Zeit in Zügen erfassen, kann der Programmierer durch die Zeit reisen. Durch *Sicherheit auf der Sprachebene*, auf die wir in Application safety, library safety, and beyond eingehen, kann eine gänzlich deterministische und isolierte Ausführung garantiert werden. All das ist möglich, indem wir die Details im Umgang mit Veränderung abstrahieren, so dass der Nutzer nicht darüber nachdenken muss; der Kern von Goblins macht das schon für den Nutzer. Dazu mehr in einem späteren Paper.

11. Über die Erweiterbarkeit von Scheme (und Lisp generell)

Sagen wir, wir hätten gerne neue Syntax. Vielleicht wollen wir, zum Beispiel, falls eine Bedingung erfüllt ist, mehrere Code-Stückchen sequenziert nacheinander ausführen. Wir könnten schreiben:

```
(if (unser-test)
    (begin
      (mach-die-sache-1)
      (mach-die-sache-2)))
```

Doch das sieht nicht schön aus. Wie wäre es, wenn wir neue Syntax just für diesen Zweck erfänden?

```
(when (unser-test)
  (mach-die-sache-1)
  (mach-die-sache-2))
```

`when` lässt sich nicht als eine Funktion realisieren, weil wir `(mach-die-sache-1)` und `(mach-die-sache-2)` nur dann ausführen möchten, wenn `(unser-test)` erfolgreich war. Wir brauchen neue Syntax.

Könnten wir die neue Syntax selbst schreiben? Wenn wir uns zurückerinnern, dass man „Lisp in Lisp schreiben“ kann, scheint es, als laute die Antwort ja:

```
REPL> (define (when test . rumpf)
        `(if ,test
             ,(cons 'begin rumpf)))
REPL> (when '(unser-test)
        '(mach-die-sache-1)
        '(mach-die-sache-2))
; => (if (unser-test)
;        (begin
;          (mach-die-sache-1)
;          (mach-die-sache-2)))
```

Tatsächlich wird die richtige Syntax geschrieben! Und es zeigt, dass man mit Lisp tatsächlich „Code schreiben kann, der Code schreibt“.[17]

Allerdings zeigen sich auch zwei offensichtliche Probleme mit diesem ersten Versuch:

- Wir mussten jedes an `build-when` übergebene Argument quotieren. Das nervt.
- `build-when` führt seinen Code gar nicht aus, sondern liefert nur die *quotierte Struktur* als Ergebnis zurück, zu der der Code entfaltet werden soll.

Aber wenn wir unsere Prozedur nur ein klein wenig anpassen, können wir unsere Prozedur zu einem *Makro* machen: einer besonderen Art von Prozedur, mit der der Compiler Code umschreibt. Alles, was wir tun müssen, ist:

```
(define-macro (when test . rumpf)
  `(if ,test
       ,(cons 'begin rumpf)))
```

Was wir tun mussten, war nur, `define` zu `define-macro` umzubennen! Jetzt weiß Scheme, dass damit Code umgeschrieben werden soll. Damit können wir neue Arten von Syntax-Formen definieren.

17 Weil Lisps *abstrakter Syntaxbaum* aus denselben Datenstrukturen gemacht ist, wie sie auch die Anwender zum Programmieren benutzen, und weil der Syntaxbaum ebenbildlisch so aussieht wie die *Oberflächensyntax* der Sprache, nennen wir sie *homoikonisch*. *Homoikonizität* ist eine Eigenschaft, die Lisp von anderen unterscheidet, und der Grund für einen Großteil von Lisps Erweiterbarkeit.

`define-macro` zeigt sehr klar auf, was Makros in Lisp und Scheme tun: Sie bearbeiten Struktur. Auf diese Weise eine Listenstruktur selbst aufzubauen, ist, wie man in Common Lisp Makros schreibt. Aber in Scheme schreibt man Makros im Allgemeinen auf andere Weise. Scheme-Makros sehen jedoch sehr ähnlich aus:

```
(define-syntax-rule (when test rumpf ...)
  (if test
      (begin rumpf ...)))
```

`define-syntax-rule` benutzt *Pattern Matching* (deutsch Mustervergleich), um Makros zu implementieren. Das erste Argument an `define-syntax-rule` beschreibt das Muster, das vom Nutzer angegeben wird, dann beschreibt das zweite Argument die Schablone, wie es umgeschrieben wird.[18] Wir sehen, dass `rumpf ...` sowohl im Muster als auch in der Schablone steht; die Auslassungspunkte `...` im Muster stehen für mehrere Ausdrücke, die aus der Eingabe des Nutzers genommen werden, und in der Schablone zeigt das `...`, wo sie wiederholt werden sollen. Es zeigt sich, dass es für diesen Mechanismus unnötig ist, selbst zu quotieren; Scheme ist schlau und kümmert sich für uns darum.

Zusammengefasst ist bei der Scheme-Version der Syntax-Definitionen weniger offensichtlich, wie sie intern abläuft, im Vergleich zur Version mit `define-macro`. Andererseits taucht beim Umschreiben von Syntax öfters ein Problem mit etwas auf, dass sich *Hygiene* nennt: Wir erwarten, dass den Rumpf nach dem Umschreiben einer Syntax-Form / Makro nicht temporäre Bezeichner von außen stören. Auf die Debatte gehen wir im Primer nicht ein, aber die Makros von Common Lisp und Scheme haben beide merkliche Nachteile, wobei es in Scheme viel wahrscheinlicher ist, dass sich Makros „hygienisch“ korrekt verhalten. Es ist leicht, simple Syntax-Formen zu schreiben, aber wenn sie kompliziert werden, wird es schwerer, sie zu schreiben und durchzublicken, wie sie intern funktionieren. Das ist der Grund, warum wir, obwohl Sie den `define-macro`-Ansatz in Scheme wahrscheinlich niemals benutzen werden, diesen zeigen, damit Sie die Idee hinter „Code, der Code schreibt“ verstehen.

Nachdem wir so weit gekommen sind, dass wir wissen, wie man neue Syntax auf Scheme-Art erzeugt, wollen wir mal sehen, ob wir uns jetzt das Leben leichter machen können. Schauen wir uns noch einmal an, wie wir `for-each` zuvor benutzt haben:

18 Tatsächlich ist selbst `define-syntax-rule` Zucker. Folgendes ist äquivalent:

```
(define-syntax-rule (when test body ...)
  (if test
      (begin body ...)))

(define-syntax when
  (syntax-rules ()
    ((when test body ...)
     (if test
         (begin body ...)))))
```

Auch lässt sich `define-syntax-rule` mittels `define-syntax` und `syntax-rules` schreiben:

```
(define-syntax define-syntax-rule
  (syntax-rules ()
    ((define-syntax-rule (id pattern ...) template)
     (define-syntax id
       (syntax-rules ()
         ((id pattern ...)
          template))))))
```

Es gibt einen ganzen Zoo anderer Syntax-Formen zum Umschreiben von Syntax in den meisten Scheme-Implementierungen und meist unterscheiden sie sich von Scheme zu Scheme, allerdings gehören `define-syntax` und `syntax-rules` zum Scheme-Standard.

```
REPL> (for-each (lambda (str)
                  (display
                   (string-append "Ich liebe "
                                  (string-upcase str)
                                  " einfach!!!\n")))
                '("Erdbeeren" "Bananen" "Trauben"))
; zeigt an:
;   Ich liebe ERDBEEREN einfach!!!
;   Ich liebe BANANEN einfach!!!
;   Ich liebe TRAUBEN einfach!!!
```

Das klappt, aber es ist umständlicher als nötig. Dass wir `lambda` hinschreiben müssen, ist ein unnötiges Detail! Eine kleine neue Syntaxdefinition macht den Code sauber:

```
(define-syntax-rule (for (item lst) body ...)
  (for-each (lambda (item)
              body ...)
            lst))
```

Schauen wir mal:

```
REPL> (for (str '("Erdbeeren" "Bananen" "Trauben"))
        (display
         (string-append "Ich liebe "
                        (string-upcase str)
                        " einfach!!!\n")))
; zeigt an:
;   Ich liebe ERDBEEREN einfach!!!
;   Ich liebe BANANEN einfach!!!
;   Ich liebe TRAUBEN einfach!!!
```

Es klappt! Das ist viel lesbarer.[19]

Wir müssen hier nicht aufhören. Spritely Goblins enthält eine Technik namens `methods` und wir zeigen sie als ein Beispiel für ein Makro. Diese Version ist ein wenig vereinfacht:

```
(define-syntax-rule (methods ((method-id method-args ...)
                              body ...) ...)
  (lambda (method . args)
    (letrec ((method-id
              (lambda (method-args ...)
                body ...)) ...)
      (cond
       ((eq? method (quote method-id))
        (apply method-id args)) ...
       (else
        (error "No such method:" method))))))
```

19 Eine spaßige Übung für unsere Leser: Versuchen Sie, eine for-Schleife wie in C zu schreiben!
Hier ist C-Code:

```
// C-Version von for:
//   for ( Init; Bedingung; Inkrement ) {
//      Anweisung(en);
//   }
for (i = 0; i < 10; i = i + 2) {
  printf("i ist: %d\n", i);
}
```

Versuchen Sie, Folgendes zum Funktionieren zu bringen:

```
(for ((i 0) (< i 10) (+ i 2))
  (display (string-append "i ist: " (number->string i) "\n")))
```

Wir können daran sowohl sehen, wie viel Beispiele für Mustervergleich im Stil von Scheme ausdrücken können, als auch, wie verwirrend mehrere Schichten mit Auslassungspunkten (also `...`) sein können. Es stellt eine kleine Herausforderung dar, zu erkennen, wie das Code-Umschreibesystem damit umgeht, alles zu entpacken.

Aber denken wir später weiter nach und sehen uns erst ein Beispiel für die Nutzung an:

```
REPL> (define (make-enemy name hp)
        (methods
         ((get-name)
          name)
         ((damage-me weapon hp-lost)
          (cond
           ((dead?)
            (format #t "~a ist leider bereits tot!\n" name))
           (else
            (set! hp (- hp hp-lost))
            (format #t "Du greifst ~a an und machst ~a Schaden!\n"
                    name hp-lost))))
         ((dead?)
          (<= hp 0))))
REPL> (define hobgob
        (make-enemy "Hobgoblin" 25))
REPL> (hobgob 'get-name)
; => "Hobgoblin"
REPL> (hobgob 'dead?)
; => #f
REPL> (hobgob 'damage-me "Keule" 10)
; zeigt an: Du greifst Hobgoblin an und machst 10 Schaden!
REPL> (hobgob 'damage-me "Schwert" 20)
; zeigt an: Du greifst Hobgoblin an und machst 20 Schaden!
REPL> (hobgob 'damage-me "Sellerie" 2)
; zeigt an: Hobgoblin ist leider bereits tot!
REPL> (hobgob 'dead?)
; => #t
```

Da geht noch mehr. Wir können Scheme um Logikprogrammierung erweitern oder um einen eigenen Mustervergleich, etc etc etc. Tatsächlich machen wir gleich im nächsten Unterabschnitt Gebrauch von dem Mustervergleich, der in Guiles Standardbibliothek mit dabei ist.

Weil die Syntax von Lisp/Scheme erweiterbar ist, können wir Funktionalitäten, die Programmiersprachen auszeichnen, nachrüsten, indem wir sie als Bibliothek implementieren, statt dass wir getrennte neue Untersprachen bräuchten. Mehrere Problembereiche können in ein System zusammengefasst werden. Deswegen sagt man, die Sprachen aus der Lisp-Sprachfamilie bieten Unterstützung für *composable domain specific languages*, also domänenspezifische Sprachen, die miteinander verknüpft werden können.

Man gewinnt an Freiheit. In anderen Programmiersprachen sind die Nutzer gezwungen, am Altar der Implementierer ihrer Programmiersprache zu flehen, sie mögen bestimmte Funktionalitäten in die nächste offizielle Veröffentlichung der Sprache aufnehmen, während die Funktionalitäten in den Händen eines Lispers / Schemers mit nur wenigen Zeilen hinzufügbar sind.

Darin steckt wahre Macht. Aber das war nicht alles. Im nächsten Abschnitt werden wir Scheme an sich freilegen, so dass wir selbst die zugrunde liegenden Mechanismen konfigurieren und mit ihnen experimentieren können, wofür man überraschend wenig Code schreiben muss.

12. Scheme in Scheme

Wir präsentieren eine lauffähige Implementierung von Scheme, geschrieben in Scheme:

```
(use-modules (ice-9 match))

(define (env-lookup env name)
  (match (assoc name env)
    ((_key . val)
     val)
    (_
     (error "Variable nicht gebunden:" name))))

(define (extend-env env names vals)
  (if (eq? names '())
      env
      (cons (cons (car names) (car vals))
            (extend-env env (cdr names) (cdr vals)))))

(define (evaluate expr env)
  (match expr
    ;; Unterstützung für die eingebauten Datentypen
    ((or #t #f (? number?))
     expr)
    ;; Quotierung
    (('quote quoted-expr)
     quoted-expr)
    ;; Variable nachschlagen
    ((? symbol? name)
     (env-lookup env name))
    ;; Bedingungen
    (('if test consequent alternate)
     (if (evaluate test env)
         (evaluate consequent env)
         (evaluate alternate env)))
    ;; Lambdas (Prozeduren)
    (('lambda (args ...) body)
     (lambda (. vals)
       (evaluate body (extend-env env args vals))))
    ;; Prozeduraufrufe (-anwendungen)
    ((proc-expr arg-exprs ...)
     (apply (evaluate proc-expr env)
            (map (lambda (arg-expr)
                   (evaluate arg-expr env))
                 arg-exprs)))))
```

Wenn wir Kommentare, leere Zeilen und den Import des Mustervergleichers nicht zählen (der nicht unbedingt notwendig ist, aber einfacher), macht das bloß 30 Code-Zeilen. Obwohl dieser Evaluator nur das Notwendigste enthält, ist er doch vollständig genug, als dass man damit alles Vorstellbare berechnen kann. (Und ja, man könnte auch noch so einen Scheme-Evaluator in diesem Scheme-Evaluator schreiben!)[20]

Unser `evaluator` nimmt zwei Argumente: einen Scheme-Ausdruck `expr` und eine Umgebung („environment“) namens `env`. Weil die Struktur von Scheme so lispig ist, können wir, wie wir gelernt haben, ganze Code-Bereiche einfach quotieren. (Und genau das machen wir auch.) Die `env` im zweiten Argument ist eine assoziative Liste, die Symbole als Namen auf die zugehörige

20 Die Idee, eine Sprache aufbauend auf einer ähnlichen Wirtssprache zu implementieren, nennt sich „metazirkulärer Evaluator“. Forscher auf dem Gebiet der Informatik mögen sie, denn damit lassen sich Varianten im Entwurf von Programmiersprachen erkunden, ohne dass man das gesamte System neu erfinden müsste.

Prozedur abbildet.

Sie sollten es mit eigenen Augen sehen. Machen wir ein bisschen einfache Arithmetik. Dazu legen wir einige Mathe-Prozeduren in der Standardumgebung ab:

```
(define math-env
  `((+ . ,+)
    (- . ,-)
    (* . ,*)
    (/ . ,/)))
```

Wir können beobachten, dass unser erstes Beispiel zur „Substitutionsmethode“ in dieser Umgebung richtig funktioniert.

```
REPL> (evaluate '(* (- 8 (/ 30 5)) 21)
                math-env)
; => 42
```

Das sieht aus wie dieselbe Antwort, die wir damals in unserem Programm hatten!

Auch das Erschaffen eines Lambdas und dessen Anwendung klappt. Hier ein Lambda, das eine Zahl quadriert:

```
REPL> (evaluate '((lambda (x)
                    (* x x))
                  4)
                math-env)
; => 16
```

Wunderbar, es funktioniert.

Beschäftigen wir uns mit etwas Fortgeschrittenerem. Wenn wir nur zwei Operatoren haben, + und =, ist das genug, um die Fibonacci-Folge auszurechnen:

```
REPL> (define fib-program
        '((lambda (prog arg)   ; "boot"
            (prog prog arg))
          (lambda (fib n)      ; "main", das Hauptprogramm
            (if (= n 0)
                0
                (if (= n 1)
                    1
                    (+ (fib fib (+ n -1))
                       (fib fib (+ n -2))))))
          10))                 ; Argument
REPL> (define fib-env
        `((+ . ,+)
          (= . ,=)))
REPL> (evaluate fib-program fib-env)
; => 55
```

Es sieht aus wie Magie. Aber es läuft! Der Evaluator führt die zugrunde liegende Berechnung aus und eine einfache Additionsprozedur (positiver und negativer Zahlen) und eine Prozedur zum Testen auf numerische Gleichheit reichen dazu aus, und diese zwei Prozeduren haben wir verfügbar gemacht.

Es ist notwendig, dass sich das Hauptprogramm selbst aufrufen kann. Aus diesem Grund beginnen wir mit der ersten Prozedur (als `boot` markiert), die ein Programm und ein Argument nimmt und diese Prozedur auf sich selbst und dem Argument aufruft[21] Die zweite Prozedur (markiert als

21 Zu Demonstrationszwecken haben wir ein kompliziertes Beispiel genommen, wo wir auf ein Problem stoßen: Fibonacci, wie wir es implementiert haben, ist selbstrekursiv! Aber selbstrekursive Funktionen kriegen wir sonst nur

Hauptprogramm `main`) nimmt sich selbst als Argument `fib` entgegen (übergeben von unserer Boot-Prozedur) sowie `n` als weiteres Argument (auch übergeben von der Boot-Prozedur) … und dann läuft es! Unser Evaluator berechnet rekursiv die Fibonacci-Folge.

Und unser Evaluator ist gut verständlich. Nehmen wir ihn in seine Einzelabschnitte auseinander.

```
(define (env-lookup env name)
  (match (assoc name env)
    ((_key . val)
     val)
    (_
     (error "Variable nicht gebunden:" name))))
```

Der Anfang ist leicht. Wir definieren Umgebungen als assoziative Listen, deshalb sucht `env-lookup` lediglich nach einem passenden Namen in der Liste. Was neu hinzukommt, wird zuerst gefunden, und so wird eine erneute Definition in einem Geltungsbereich tiefer im Programm die des äußeren Geltungsbereichs *überschatten*. Sehen wir uns das einmal an:

```
REPL> (env-lookup '((foo . neueres-foo)
                    (bar . bar)
                    (foo . älteres-foo))
                  'foo)
; => 'neueres-foo
```

Jetzt folgt ein Hilfsmittel:

```
(define (extend-env env names vals)
  (if (eq? names '())
      env
      (cons (cons (car names) (car vals))
            (extend-env env (cdr names) (cdr vals)))))
```

`extend-env` nimmt eine Umgebung und eine Liste von Namen sowie eine parallel laufende Liste von Werten. Das erleichtert uns später den Code, der Prozeduren definiert. Probieren wir es aus, dann ist alles klar:

```
REPL> (extend-env '((foo . wert-von-foo))
                  '(bar quux)
                  '(wert-von-bar wert-von-quux))
; => ((bar . wert-von-bar)
;     (quux . wert-von-quux)
;     (foo . wert-von-foo))
```

Jetzt geht’s an den Evaluator. Die äußere Schale von `evaluate` ist so aufgebaut:

```
(define (evaluate expr env)
  (match expr
    (<MATCH-PATTERN>
     <MATCH-BODY> ...) ...))
```

`evaluate` nimmt zwei Argumente an:

mit Techniken wie `letrec` hin, das wir nicht bereitgestellt haben. Als Ausweg übergeben wir an das Hauptprogramm (die zweite Prozedur) sie selbst und benutzen dazu die erste Prozedur. Deshalb wird die erste Prozedur im Kommentar auch "boot" genannt.

Es handelt sich um eine Art eingeschränkten Y-Kombinator, einer Bootstrapping-Technik. (Wir meinen nicht das Unternehmen, das Unternehmen gründet, aber da kommt dessen Name her.) Mit dem Y-Kombinator wird dasselbe gemacht, nur allgemeiner. Solche Emulatoren wie unserer sind immer ähnlich wie Y. The Why of Y ist ein hübscher und knapper Artikel zum Y-Kombinator und wie man aus praktischen Gründen darauf kommen könnte, ähnlich zu unserem gedanklichen Pfad. The Little Schemer beendet seine schöne Reise auch mit dem Schreiben eines metazirkulären Evaluators wie bei uns, und dort wird darauf eingegangen, wie man Y herleiten könnte.

- `expr`: welcher Ausdruck ausgewertet werden soll
- `env`: in welcher Umgebung wir den Ausdruck auswerten

Im Rumpf von `evaluate` teilen wir eine andere Verhaltensweise zu, je nachdem, auf welches Muster / Pattern der Ausdruck `expr` passt. Wir benutzen `match` aus Guiles Syntax für Pattern Matching (deren Modul wir oben importiert haben). Kurz gefasst, wenn ein `<MATCH-PATTERN>` passt, hören wir auf, die Muster zu durchsuchen, und es wird nur `<MATCH-BODY>` ausgewertet (mit den Bindungen aus `<MATCH-PATTERN>`, falls vorhanden).

Also sehen wir uns jetzt nach und nach jedes Muster an, das wir in `evaluate` unterstützen. Los geht's mit etwas Einfachem:

```
;; Unterstützung für die eingebauten Datentypen
((or #t #f (? number?))
 expr)
```

Das `or` bedeutet, irgendeines der enthaltenen Muster kann passen. Die ersten zwei sind die Werte wahr und falsch aus Scheme. Die Klammern mit dem Symbol `?` am Anfang zeigen an, dass ein Prädikat passen muss, in diesem Fall `number?`. Wenn eines davon passt, liefern wir genau denselben Ausdruck `expr`, der passen muss, wieder zurück. Wir leihen uns also die Booleschen und Zahlenwerte direkt von der zugrunde liegenden Scheme-Implementierung aus.

Mit anderen Worten, dadurch funktioniert:

```
REPL> (evaluate #t '())
; => #t
REPL> (evaluate #f '())
; => #f
REPL> (evaluate 33 '())
; => 33
REPL> (evaluate -2/3 '())
; => -2/3
```

Das war leicht! Noch ein Leichtes:

```
;; Quotierung
(('quote quoted-expr)
 quoted-expr)
```

Erinnern Sie sich, dass `'foo` nur die Kurzform ist von `(quote foo)`, genau wie `'(1 2 3)` die Kurzform ist von `(quote (1 2 3))`. Mit diesem Muster kümmern wir uns um alle Listen, die losgehen mit dem Symbol `'quote` und einem zweiten Element mit dem zu quotierenden Ausdruck.

Anders gesagt, hierdurch funktioniert:

```
REPL> (evaluate ''foo '())
; => foo
REPL> (evaluate ''(1 2 3) '())
; => (1 2 3)
REPL> (evaluate (quote (quote (1 2 3))) '())
; => (1 2 3)
```

Die letzten beiden sind dasselbe. Beachten Sie, wir quotieren zweimal: Einmal, weil wir das gesamte evaluierte Programm quotieren, und einmal als Teil des quotierten Programms, in dem wir einen Ausdruck quotieren möchten.

So weit, so gut. Das nächste ist wieder eher leicht:

```
;; Variable nachschlagen
((? symbol? name)
 (env-lookup env name))
```

Der Teil mit `(? symbol? name)` bindet `name` an die passende Komponente. (In diesem Fall wird `name` an denselben Wert gebunden wie dem in `expr`, noch bevor wir matchten, aber `name` zu schreiben ist ein wenig leserlicher.)

Der Rumpf, also der `<MATCH-BODY>`, ist recht einfach. Was `env-lookup` tut, haben wir schon gesehen. Mit anderen Worten, wenn wir ein Symbol sehen (das natürlich nicht quotiert ist, sonst hätten wir uns darum gekümmert), dann schlagen wir den entsprechenden Wert in der Umgebung `env` nach.

Das heißt, es funktioniert:

```
REPL> (evaluate 'x '((x . 33)))
; => 33
```

Es leistet aber auch, dass wir bei Lambda-Anwendungen definierte Variable nachschlagen können:

```
REPL> (evaluate '((lambda (x) x) 33) '())
; => 33
```

Auch wenn wir noch gar nicht bei `lambda` angekommen sind! Aber so gut wie.

Als Nächstes, Bedingungen, was sich als recht einfach herausstellt:

```
;; Bedingungen
(('if test consequent alternate)
 (if (evaluate test env)
     (evaluate consequent env)
     (evaluate alternate env)))
```

Anders ausgedrückt passt eine Liste mit dem Symbol `'if` am Anfang und nach dem `'if` drei Unterausdrücken, die während des match-Rumpfs an die Variablen `test`, `consequent`, und `alternate` gebunden sind. Wir benutzen das `if`, das uns das zugrunde liegende Scheme bereitstellt, und werten zunächst `test` in der aktuellen Umgebung `env` aus (das ist eine Rekursion!). Mithilfe des `if` vom Wirts-Scheme wird entschieden und die Folgerung `consequent` oder die Alternative `alternate` in der Umgebung `env` ausgewertet, wozu wieder `evaluate` rekursiv aufgerufen wird.

Nun denn, jetzt zu den Prozeduren. Das ist ein bisschen komplizierter, aber auch nicht zu sehr:

```
;; Lambdas (Prozeduren)
(('lambda (args ...) body)
 (lambda (. vals)
   (evaluate body (extend-env env args vals))))
```

Hier geht es im Muster um eine Liste mit `'lambda` am Anfang, als zweitem Listenelement den Argumenten, und den Rumpf binden wir ans englische Wort für Rumpf, `body`. Dann liefern wir als Ergebnis eine Prozedur zurück, die beim Evaluieren mit der richtigen Anzahl Argumente aufgerufen werden kann.[22] Im Inneren der Prozedur, die wir zurückliefern, wird `evaluate` rekursiv aufgerufen auf den Ausdruck `body` aus dem Lambda, das wir hier mit match vorfinden, aber in einer neuen, erweiterten Umgebung, wo auch die Namen in `args` gebunden sind an die Werte

22 Wenn ein Benutzer die Prozedur mit der falschen Anzahl an Argumenten aufruft, verursacht das eine Fehlermeldung, die aber nicht sagt, was los ist. Denken Sie darüber nach, wie wir eine bessere Meldung bringen könnten. (Hinweis: Wenn sich die `length` von `args` und `vals` unterscheidet, stimmt 'was nicht!)

`vals` des Prozeduraufrufs.[23]

Anders gesagt, damit funktioniert:

```
REPL> ((evaluate '(lambda (x y) x) '())
       'first 'second)
; => first
REPL> ((evaluate '(lambda (x y) y) '())
       'first 'second)
; => second
```

Da bleibt nur noch eines … Prozeduren anzuwenden!

```
;; Prozeduraufrufe (-anwendungen)
((proc-expr arg-exprs ...)
 (apply (evaluate proc-expr env)
        (map (lambda (arg-expr)
               (evaluate arg-expr env))
             arg-exprs)))
```

Dieses Puzzlestück passt auf alle Arten von Prozeduranwendungen! Wenn wir bis hierhin gekommen sind, passt das Muster auf jede Liste mit ein oder mehr Listenargumenten, was auf eine Prozedur hinweist, der Argumente übergeben werden. Wir werten aus, was `proc-expr` ist, denn es ist die Prozedur, die wir auswerten wollen, und dabei gelten die aktuellen Argumente, wozu wir `evaluate` rekursiv mit der aktuellen Umgebung `env` aufrufen. Wir sammeln zudem all die `arg-expr`-Argumentausdrücke ein, die an die Prozedur übergeben werden, indem wir `evaluate` rekursiv auf jedes aufrufen, mit der aktuellen Umgebung, `env`.

Anders gesagt, damit funktioniert:

```
REPL> (evaluate '(* (- 8 (/ 30 5)) 21)
                math-env)
; => 42
```

Und wenn wir alles zusammensetzen, verleiht uns das nicht nur die Macht, die Fibonacci-Folge zu berechnen, sondern auch jedes andere berechenbare Problem, das wir und vorstellen können!

Um fair zu sein, haben wir uns dazu ein gutes Stück von Schemes Macht ausgeliehen, aber auch nicht so viel, wie man denkt … besonders im Vergleich zu dem, was sich viele anhand anderer Sprachen implementierte Sprachen leihen (ganz, ganz besonders viel weniger als sich beispielsweise Clojure von Java ausleiht oder sich so ziemlich jede beliebte Sprache von der C-Standardbibliothek leiht).[24] Auch ist unser Evaluator zu unvollständig, um irgendeinem der Scheme-

23 Hier fällt die Entscheidung zwischen lexikalischer und dynamischer Bindung. Erweitern tun wir nicht die Umgebung von da, wo aufgerufen wird, sondern die Umgebung des letzten Ausdrucks, in der das Lambda definiert worden ist. Das ermöglicht, dass die Techniken aus dem Abschnitt über Closures (Abschlüsse) funktionieren.
Man kann „lexikalische Bindung“ beschreiben als dass die Bindungen „von da kommen, wo der Code geschrieben ist“. Das bietet Sicherheit über Capabilitys, ist explizit und garantiert die Eigenschaft, dass man „nur auf das Zugriff hat, was man auch hat“. Bei „dynamischer Bindung“ gälte hingegen, dass die Bindungen „von da kommen, wo der Aufruf stattfindet“ (und wo der Aufruf davon stattfindet und so weiter, den aktuellen Aufruf-Stack hinauf). Bindungen sind implizit und weithin zugänglich (ambient im Sinne von „Ambient Authority“). Dynamische Bindung hat somit öfter schwerwiegende Sicherheitsnachteile (es ist kein Zufall, dass sie an Zugriffskontrolllisten, ACLs, erinnert).

24 Es ist bemerkenswert, dass wir die Mächtigkeit in `quote` so ausspielen konnten, dass wir einen Interpretierer zustande gebracht haben, ohne überhaupt Syntax in Textform zu parsen: Wir haben lediglich unsere Datenstrukturen quotiert! Tatsächlich geht es in der Mehrzahl der Lehrbücher zum Entwurf von Programmiersprachen viel zu sehr um das Parsen der Textform einer Sprache, und das verleitet zur Illusion, das Wichtige an der Struktur einer Sprache hätte zu tun mit der *Oberflächensyntax*. Das ist eigentlich niemals wahr. Jedes vorstellbare Paradigma einer Programmiersprache ist darstellbar als einfacher symbolischer Ausdruck, so wie in Scheme. In Wirklichkeit sind sich Code und Daten viel ähnlicher als es scheint.

Standards zu genügen. Da hinzukommen, wäre ohne allzu viel Mehraufwand möglich, und zur Demonstration reicht er aus. Zumal es *uns überlassen* ist, wie mächtig wir die Sprache machen wollen. Das hängt alles davon ab, in welcher Anfangsumgebung wir Code evaluieren.

Unsere Sprache kann zudem Sicherheit über Capabilitys bieten! Abgesehen davon, dass man in eine Endlosschleife laufen und übermäßig Ressourcen wie Speicher oder Rechenleistung verbrauchen kann, ist es unmöglich, etwas wirklich Schlimmes mit unserer Sprache anzurichten. Dennoch ist es unsere Entscheidung, wie viel Macht wir ihr geben. Wenn wir das wollen, können wir eine Umgebung mit veränderlichen Zellen oder mit Dateisystemzugriff zur Verfügung stellen. Das ist unsere Sache.

Und mit winzigen Anpassungen können wir unserem Evaluator verschiedenste wunderbare Kunststücke beibringen. Wir können neue Syntax einführen. Wir können neue Syntax einführen, um neue Syntax einzuführen (Makros)! Wir können die Auswertungsreihenfolge ändern, ein statisches Typsystem einführen und noch viel mehr.

Unser Versprechen war, dass Sie am Ende dieser Anleitung Scheme erlernt hätten. Wenn Sie einmal hier angekommen sind, haben Sie weitaus mehr erreicht, denn Sie sind nicht bloß Benutzer von Scheme, sondern ein Gestalter von Scheme. Die Macht gehört Ihnen![25]

Natürlich stimmt es, dass wir für eine nützliche Programmiersprache auch Programme aus Quelldateien auf Datenträgern lesen können wollen. `read` ist ein Teil des Scheme-Standards (und auch in fast jedem anderen Lisp enthalten), also können wir im Allgemeinen einfach davon Gebrauch machen. `read` selber in Scheme zu implementieren, ist aber auch leicht. Dazu ist ungefähr so viel Code nötig, wie wir für `evaluate` gebraucht haben.

25 Sie haben es nicht nur bis zum letzten Abschnitt geschafft, sondern bis zur letzten Fußnote! Wir dürfen also annehmen, dass Sie sich wirklich für solche Dinge interessieren, deshalb nennen wir Ihnen noch weitere Ressourcen:

- Mehr darüber, wie Evaluatoren wie der, den wir oben geschrieben haben, funktionieren, und etwas über die Geschichte dahinter, erfahren Sie in William Byrds großartigem Vortrag: The Most Beautiful Program Ever Written.
- Structure and Interpretation of Computer Programs (kurz SICP oder „Wizard Book“) enthält ausführlichere Fassungen fast aller Themen, die wir auch besprochen haben. Eine deutsche Übersetzung „Struktur und Interpretation von Computerprogrammen“ hat Susanne Daniels-Herold verfasst. In SICP findet man des Weiteren, wie man eine Ereignisschleife aufbaut, Lösungsprogramme für Beschränkungen, Evaluatoren wie unseren, Evaluatoren für Logikprogramme sowie Compiler in effizienteren Maschinencode. Es gibt keine bessere Möglichkeit, etwas über die Natur von Berechnungen zu lernen, als SICP zu studieren. Dazu eignet sich, zwischen dem Lesen des Quellbuchs und dem Schauen der Vorlesungen aus den 1980ern zu wechseln. Die englische Version von SICP ist verfügbar als gedrucktes Buch, als „info“-Handbuch, das man bequem aus Emacs heraus lesen kann, oder als HTML, aber wenn Sie eh schon mit einem Web-Browser lesen, raten wir deutlich zu dieser Fassung.
- The Little Schemer ist ein unterhaltsames Buch, das als Gespräch geschrieben wurde. Darin finden sich viele nützliche Unterrichtseinheiten auf spielerische, puzzleartige Art, garniert mit entzückenden Illustrationen. Es gibt zahlreiche Nachfolgebücher aus dieser Reihe, wo andere Gebiete der Informatik mit Tiefgang erkundet werden.
- Software Design for Flexibility führt viele Ideen aus SICP fort und baut darauf weiter auf. Man kann es als den wahren Nachfolger von SICP auffassen.
- Ganz klar lernt man aber sehr viel über Scheme oder andere Lisps, indem man sie einfach benutzt. Wir schätzen es, wenn Sie zu Spritely oder Guix beitragen, wo Sie Ihre Fähigkeiten einbringen können!

Gute Reise!