

Kapitel 10

Metazirkuläre Interpretation

If you don't understand interpreters, you can still write programs;
you can even be a competent programmer. But you can't be a master.
—Hal Abelson

Das Umgebungsmodell ist ein Hilfsmittel zur Beschreibung des Ablaufes von Programmen. Das Umgebungsmodell bedient sich natürlicher Sprache und Bildern als Beschreibungsmittel. Das sorgt für Anschaulichkeit, aber läßt auch immer Platz für Ungenauigkeiten und Mißverständnisse. Eine präzise Beschreibung des Umgebungsmodells erfordert eine präzisere Sprache — am besten, ein lauffähiges Computerprogramm, welches das Umgebungsmodell simuliert.

Ein solches Programm, das die Auswertung eines anderen Programms simuliert, heißt *Interpreter*. Um einen solchen Interpreter für eine Teilmenge von Scheme geht es in diesem Kapitel. Er hat zwei Funktionen: Einerseits soll er eine präzise Beschreibung der Auswertung eines Scheme-Programms liefern, aber auch neue Erkenntnisse über Programmierung und Programmieretechniken liefern.

Die Tatsache, daß ein Interpreter, welcher die Bedeutung von Programmen festlegt, selbst wieder ein Programm ist, ist eine der wichtigsten Einsichten in der Programmierung überhaupt. Eine weitere nützliche Sichtweise entsteht durch die Erkenntnis, daß eine Datenstruktur selbst auch eine (sehr eingeschränkte) Programmiersprache ist. Diese Einsichten öffnen die Tür zu einem ganzen neuen Arsenal an mächtigen Programmieretechniken.

10.1 Mini-Scheme: eine Untermenge von Scheme

Für einen Interpreter muß zuerst festgelegt werden, welche Sprache er überhaupt behandelt. Der Interpreter dieses Kapitels verarbeitet eine Untermenge von Scheme, genannt *Mini-Scheme*. Mini-Scheme kennt nur die Kernformen von Scheme: es gibt keine direkt abgeleiteten Formen wie `let` oder `cond`. Außerdem darf der Rumpf einer Abstraktion nur einen einzelnen Ausdruck enthalten. Sollen mehrere Ausdrücke hintereinanderstehen, muß `begin` benutzt werden. Die Syntax von Mini-Scheme wird von folgender Grammatik beschrieben:

```
<program> ::= <form>*  
<form> ::= <definition> | <expression>  
<definition> ::= (define <variable> <expression>)  
<expression> ::= <variable>  
                | <literal>  
                | <lambda expression>
```

```

    | <conditional>
    | <procedure call>
    | <assignment>
    | <block>
<literal> ::= <quotation> | <self-quoting>
<quotation> ::= (quote <datum>)
<self-quoting> ::= <boolean> | <number>
<boolean> ::= #f | #t
<lambda expression> ::= (lambda <formals> <body>)
<body> ::= <expression>
<formals> ::= (<variable>*)
<conditional> ::= (if <test> <consequent> <alternate>)
<test> ::= <expression>
<consequent> ::= <expression>
<alternate> ::= <expression>
<procedure call> ::= (<operator> <operand>*)
<operator> ::= <expression>
<operand> ::= <expression>
<assignment> ::= (set! <variable> <expression>)
<block> ::= (begin <sequence>)
<sequence> ::= <expression>+

```

Das Programmieren von Interpretern für Scheme in Scheme wird dadurch erleichtert, daß jede Form von Mini-Scheme durch einen repräsentierbaren Scheme-Wert abbildbar ist. Eine Form läßt sich durch Quotierung in einen Wert verwandeln, der genauso aussieht. Ein Programm wird einfach durch eine Liste solcher Formen abgebildet. Hier ist ein Beispiel für ein Mini-Scheme-Programm:

```

(define mini-scheme-example
  '(define fac
    (lambda (n)
      (if (= n 1)
          1
          (* n (fac (- n 1))))))
    (fac 5)))

```

Die spätere Programmierung wird durch eine Sammlung von Prädikaten und Selektoren erleichtert, welche aus der Repräsentation für Formen die entsprechenden Komponenten extrahieren. Variablen sind durch Symbole repräsentiert:

```

(define variable?
  (lambda (form)
    (symbol? form)))

(define variable-name
  (lambda (form)
    form))

```

Kombinationen werden durch Listen repräsentiert, deren erstes Element den Operator darstellt. Die Erzeugung der entsprechenden Prädikate läßt sich automatisieren:

```

(define make-combination-predicate
  (lambda (name)
    (lambda (form)
      (and (pair? form)
           (eq? name (car form))))))

```

Nun lassen sich Definitionen erkennen:

```
(define define? (make-combination-predicate 'define))
```

```
(define define-variable-name
  (lambda (form)
    (car (cdr form))))
```

```
(define define-expression
  (lambda (form)
    (car (cdr (cdr form)))))
```

Gequotete Literale lassen sich am quote erkennen:

```
(define quote? (make-combination-predicate 'quote))
```

```
(define quote-constant
  (lambda (form)
    (car (cdr form))))
```

Literale sind entweder gequotet oder repräsentieren sich selbst:

```
(define literal?
  (lambda (form)
    (or (quote? form)
        (and (not (pair? form))
              (not (variable? form))))))
```

```
(define literal-constant
  (lambda (form)
    (if (quote? form)
        (quote-constant form)
        form)))
```

Lambda-Ausdrücke bestehen aus Parametern und Rumpf:

```
(define lambda? (make-combination-predicate 'lambda))
```

```
(define lambda-parameters
  (lambda (form)
    (car (cdr form))))
```

```
(define lambda-body
  (lambda (form)
    (car (cdr (cdr form)))))
```

Conditionals bestehen aus Test, Konsequente und Alternative:

```
(define if? (make-combination-predicate 'if))
```

```
(define if-test
  (lambda (form)
    (car (cdr form))))
```

```
(define if-consequent
  (lambda (form)
    (car (cdr (cdr form)))))
```

```
(define if-alternative
  (lambda (form)
    (car (cdr (cdr (cdr form))))))
```

Nun sind Zuweisungen an der Reihe:

```
(define set!? (make-combination-predicate 'set!))
```

```
(define set!-variable-name
  (lambda (form)
    (car (cdr form))))
```

```
(define set!-expression
  (lambda (form)
    (car (cdr (cdr form)))))
```

Blöcke mit `begin` werden noch gebraucht:

```
(define begin? (make-combination-predicate 'begin))
```

```
(define begin-expressions
  (lambda (form)
    (cdr form)))
```

Alle Kombinationen, die in keine der obigen Kategorien gehören, müssen Prozeduranwendungen sein:

```
(define application-operator
  (lambda (form)
    (car form)))
```

```
(define application-operands
  (lambda (form)
    (cdr form)))
```

10.2 Repräsentation von Werten

Die Werte, die ein Mini-Scheme-Programm manipuliert, müssen natürlich auch vom Interpreter repräsentiert werden. Dabei ist es notwendig, zwischen zwei Sorten von Werten zu unterscheiden: Werte von Ausdrücken im Mini-Scheme-Programm und eingebaute, „primitive“ Werte, die Wert keines Mini-Scheme-Ausdrucks sind. Primitive Werte sollen ausschließlich Prozeduren sein und werden ausschließlich auf „gewöhnliche“ Werte angewendet. (Diese Beschränkung zu umgehen ist nicht schwierig, erfordert aber einige strukturelle Änderungen am Interpreter — eine nährwertige Fingerübung.)

Dementsprechend werden primitive Werte als Werte vom Typ `primitive-value` repräsentiert:

```
(define primitive-value-type (make-type 'primitive-value))
```

```
(define make-primitive-value
  (lambda (value)
    (make-typed-object primitive-value-type value)))
```

```
(define value-primitive?
```

```
(typed-object-predicate primitive-value-type))
```

```
(define primitive-value
  (lambda (value)
    (typed-object-value primitive-value-type value)))
```

Sie lassen sich unterscheiden von gewöhnlichen Werten, die auf die gleiche Art und Weise, nur mit anderem Typ erzeugt werden:

```
(define ordinary-value-type (make-type 'ordinary-value))
```

```
(define make-ordinary-value
  (lambda (value)
    (make-typed-object ordinary-value-type value)))
```

```
(define value-ordinary?
  (typed-object-predicate ordinary-value-type))
```

```
(define ordinary-value
  (lambda (value)
    (typed-object-value ordinary-value-type value)))
```

Die meisten Werte, die durch Mini-Scheme-Programme zirkulieren, lassen sich durch die direkten Gegenstücke in Scheme repräsentieren: Zahlen durch Zahlen, Booleans durch Booleans etc. Lediglich bei Prozeduren schreibt das Umgebungsmodell die Verwendung von Closures vor. (In der Tat ist auch die Verwendung von Prozeduren für die Repräsentation von Prozeduren möglich, aber bei der Erläuterung des Umgebungsmodells nicht sehr erhellend.) Closures sind Tripel aus Parametern und Rumpf der Prozedur sowie einer Umgebung:

```
(define closure-type (make-type 'closure))
```

```
(define make-closure
  (lambda (parameters body environment)
    (make-typed-object
     closure-type
     (cons (cons parameters body)
           environment))))
```

```
(define closure-parameters
  (lambda (closure)
    (car (car (typed-object-value closure-type closure)))))
```

```
(define closure-body
  (lambda (closure)
    (cdr (car (typed-object-value closure-type closure)))))
```

```
(define closure-environment
  (lambda (closure)
    (cdr (typed-object-value closure-type closure)))))
```

10.3 Umgebungen und Frames repräsentieren

Umgebungen sind Folgen von Frames, die in einer globalen Umgebung enden. Eine Umgebung ist somit als Paar aus einem Frame und einer umschließenden Umgebung repräsentiert. Hier der Typ, der Konstruktor und die Selektoren:

```
(define environment-type (make-type 'environment))

(define make-environment
  (lambda (frame enclosing-environment)
    (make-typed-object
     environment-type
     (cons frame enclosing-environment))))

(define environment-frame
  (lambda (environment)
    (car (typed-object-value environment-type environment))))

(define environment-enclosing-environment
  (lambda (environment)
    (cdr (typed-object-value environment-type environment))))
```

Frames wiederum bestehen aus Bindungen, wobei jede Bindung ein Paar aus Variable und Wert ist. Da sich zu einem Frame (genauer gesagt: dem Frame der globalen Umgebung) auch Bindungen hinzufügen lassen, werden die Bindungen als car eines Pairs abgelegt, daß später mutiert werden kann:

```
(define frame-type (make-type 'frame))

(define make-frame
  (lambda (bindings)
    (make-typed-object frame-type
                       (cons bindings #f))))

(define frame-bindings
  (lambda (frame)
    (car (typed-object-value frame-type frame))))
```

Die Prozedur `extent-frame!` fügt einem Frame eine Bindung hinzu:

```
(define extend-frame!
  (lambda (frame name value)
    (let ((frame-cell (typed-object-value frame-type frame)))
      (set-car! frame-cell
                (cons (cons name value) (car frame-cell))))))
```

Die Bindungen eines Frame bilden eine Assoziationsliste. In dieser lassen sich mit Hilfe von `assoc` die Bindungen „nachschiagen“:

```
(define frame-lookup-binding
  (lambda (frame name)
    (assoc name (frame-bindings frame))))
```

Die Bindung einer Variablen in bezug auf eine Umgebung läßt sich durch Verfolgung der Frames entlang der umschließenden Umgebungen finden. Das Umgebungsmodell aus Kapitel 8 schreibt vor:

Der *Wert einer Variable* in bezug auf eine Umgebung ist der Wert der ersten Bindung in der Frame-Folge, die von der Umgebung ausgeht.

```
(define environment-lookup-binding
  (lambda (environment name)
    (let ((try-here (frame-lookup-binding (environment-frame environment) name))))
```

```
(if try-here
    try-here
    (let ((enclosing-environment
          (environment-enclosing-environment environment)))
        (if enclosing-environment
            (environment-lookup-binding enclosing-environment name)
            #f))))))
```

Der Wert einer solchen Bindung wird ggf. durch `environment-lookup` zurückgeliefert:

```
(define environment-lookup
  (lambda (environment name)
    (let ((stuff (environment-lookup-binding environment name)))
      (if stuff
          (cdr stuff)
          #f))))
```

Für die Behandlung von `set!` wird der Interpreter Bindungen nachträglich verändern müssen. Die Prozedur `environment-mutate-binding!` erledigt diese Aufgabe:

```
(define environment-mutate-binding!
  (lambda (environment name new-value)
    (let ((binding (environment-lookup-binding environment name)))
      (set-cdr! binding new-value))))
```

Eine mögliche initiale globale Umgebung enthält Definitionen für einige unverzichtbare Primitiva, welche durch ihre Scheme-Gegenstücke repräsentiert werden:

```
(define make-global-environment
  (lambda ()
    (make-environment
     (make-frame
      (list
       (cons '+ (make-primitive-value +))
       (cons '* (make-primitive-value *))
       (cons '= (make-primitive-value =))
       (cons '- (make-primitive-value -))))
     #f)))
```

10.4 Auswertung und Anwendung

Das Herzstück des Interpreters steckt in den Prozeduren `evaluate` and `apply-procedure` (in der Folklore meist einfach nur `eval` und `apply` genannt). `Evaluate` wertet einen Ausdruck in bezug auf eine Umgebung aus. Der Rumpf von `evaluate` ist im wesentlichen eine große Fallunterscheidung nach der Sorte Ausdruck, die ausgewertet wird. Der Code folgt dabei den Regeln des Umgebungsmodells. Einige Besonderheiten sind wichtig:

- `Evaluate` muß sorgfältig alle Werte mit `make-ordinary-value` verpacken, und diese Werte bei Berechnungen auch wieder auspacken.
- Das Einpacken ist besonders wichtig bei der Behandlung von `set!`: `Set!` gibt zwar „irgendeinen“ unspezifizierten Wert zurück, es muß sich aber wie sonst auch um einen `ordinary-value` handeln. Der entsprechende Wert ist `unspecific-value` mit folgender Definition:

```
(define unspecific-value
  (make-ordinary-value 'unspecific))
```

Nun ist `evaluate` an der Reihe. Der Rumpf dieser Prozedur ist eine große Fallunterscheidung nach dem Typ des Ausdrucks `expression`. Für Literale ist der Code trivial:

```
(define evaluate
  (lambda (expression environment)
    (cond
      ((literal? expression)
       (make-ordinary-value (literal-constant expression))))
```

Bei einer Variablen muß in der Umgebung nachgeschlagen werden:

```
((variable? expression)
 (environment-lookup environment
  (variable-name expression)))
```

Zur Implementation von `if` in Mini-Scheme wird einfach das `if` in Scheme, der Metasprache, verwendet:

```
((if? expression)
 (if (ordinary-value (evaluate (if-test expression) environment))
     (evaluate (if-consequent expression) environment)
     (evaluate (if-alternative expression) environment)))
```

Die Behandlung von Zuweisungen folgt der Vorschrift des Umgebungsmodells:

Eine Zuweisung führt dazu, daß die Bindung der angegebenen Variable verändert wird, so daß sie die Variable an den Wert des Ausdrucks der Zuweisung bindet.

Die Realisierung kann dazu `environment-mutate-binding!` benutzen:

```
((set!? expression)
 (environment-mutate-binding! environment
  (set!-variable-name expression)
  (evaluate (set!-expression expression)
            environment))
  unspecific-value)
```

Die Ausdrücke eines Blocks werden in einer Schleife nacheinander ausgewertet. Die Schleife merkt sich den Wert des jeweils letzten Ausdrucks in `last-value`:

```
((begin? expression)
 (letrec ((loop (lambda (expressions last-value)
                  (if (null? expressions)
                      last-value
                      (loop (cdr expressions)
                           (evaluate (car expressions) environment))))))
  (loop (begin-expressions expression) #f)))
```

Die Regel für Abstraktionen im Umgebungsmodell lautet folgendermaßen:

Der Wert einer `lambda`-Abstraktion ist ein Tripel aus den Parametern, dem Rumpf der Abstraktion und der momentanen Umgebung.

Die Realisierung folgt dem genau:

```

(lambda? expression)
  (make-ordinary-value
   (make-closure (lambda-parameters expression)
                 (lambda-body expression)
                 environment)))

```

Der erste Teil der Auswertungsregel für Prozeduranwendungen lautet:

Bei der Auswertung eines Prozeduraufrufs werden zunächst Prozedur und Operanden in bezug auf die momentane Umgebung ausgewertet.

Dieser Teil wird noch in `evaluate` erledigt. Für den Rest ist `apply-procedure` zuständig:

```

(else
 (let ((procedure (evaluate (application-operator expression)
                           environment))
       (parameter-values
        (map (lambda (operand)
              (evaluate operand environment))
             (application-operands expression))))
      (apply-procedure procedure parameter-values))))

```

`Apply-procedure` schließlich ist die andere Hälfte des Interpreters. Sie kümmert sich zunächst in einem Spezialfall um die direkte Anwendung von primitiven Prozeduren:

```

(define apply-procedure
  (lambda (procedure parameter-values)
    (if (value-primitive? procedure)
        (make-ordinary-value
         (apply (primitive-value procedure)
                (map ordinary-value parameter-values)))
        (apply-procedure procedure parameter-values))))

```

Bei gewöhnlichen, im Mini-Scheme-Programm entstandenen Prozeduren gilt die Regel aus dem Umgebungsmodell:

Der Operator muß eine Prozedur mit ebensovielen Parametern sein, wie der Prozeduraufruf Operanden hat. Nun wird ein neues Frame erzeugt, dessen umschließende Umgebung die Umgebung in der Closure ist, und in der Bindungen der Parameter an die Werte der Operanden angelegt werden. Der Wert des Prozeduraufrufs ist dann der Wert des Prozedur-rumpfes in bezug auf die Umgebung, die vom neuen Frame ausgeht.

`Apply-procedure` packt sie die Closure aus und ruft `evaluate` rekursiv auf dem Rumpf der Closure auf:

```

(let* ((closure (ordinary-value procedure))
      (parameters (closure-parameters closure))
      (new-frame
       (make-frame (zip parameters parameter-values)))
      (environment
       (make-environment new-frame
                        (closure-environment closure))))
      (evaluate (closure-body closure) environment))))

```

Die Hilfsprozedur `zip` macht dabei aus zwei Listen eine Liste von Paaren:

```
(define zip
  (lambda (list-1 list-2)
    (letrec
      ((loop
        (lambda (list-1 list-2 reverse-result)
          (if (null? list-1)
              (reverse reverse-result)
              (loop (cdr list-1) (cdr list-2)
                    (cons (cons (car list-1) (car list-2))
                          reverse-result))))))
      (loop list-1 list-2 '()))))
```

10.5 Programme ausführen

Ein Programm besteht nun nicht aus einem einzelnen Ausdruck, sondern einer Liste von Formen, unter denen auch Definitionen vorkommen können. Die Prozedur `evaluate-program` nimmt für Definitionen die entsprechenden Bindungen vor und wertet die anderen Ausdrücke aus. Die Werte der Ausdrücke werden dabei in einer Liste aufgesammelt:

```
(define evaluate-program
  (lambda (forms)
    (let ((global-environment (make-global-environment)))
      (letrec
        ((loop
          (lambda (forms reverse-values)
            (cond
              ((null? forms) (reverse reverse-values))
              ((define? (car forms))
               (process-definition (define-variable-name (car forms))
                                   (define-expression (car forms))
                                   global-environment)
               (loop (cdr forms) reverse-values))
              (else
               (let ((value (evaluate (car forms) global-environment)))
                 (loop (cdr forms) (cons value reverse-values))))))
          (map ordinary-value
               (loop forms '()))))))))
```

Eine Definition wird bearbeitet, indem das Frame der globalen Umgebung erweitert wird:

```
(define process-definition
  (lambda (name expression environment)
    (let ((value (evaluate expression environment)))
      (extend-frame!
       (environment-frame environment)
       name
       value))))
```

Fertig ist der Interpreter!