

## 17 Interpretation

- Scheme-Programme als Datenstruktur
- Interpretation von Ausdrücken
- Interpretation von Lambda
- Lambda als Datenstruktur
- Toplevel Definitionen
- set!

## 17.1 Programme als Datenstruktur

### 17.1.1 Mini-Scheme

$\langle program \rangle ::= \langle form \rangle^*$   
 $\langle form \rangle ::= \langle definition \rangle$   
 $\quad \quad \quad | \langle expression \rangle$   
 $\langle definition \rangle ::= (\text{define } \langle variable \rangle \langle expression \rangle)$   
 $\langle expression \rangle ::= \langle literal \rangle$   
 $\quad \quad \quad | \langle variable \rangle$   
 $\quad \quad \quad | (\text{if } \langle expression \rangle \langle expression \rangle \langle expression \rangle)$   
 $\quad \quad \quad | (\text{lambda } (\langle variable \rangle^*) \langle expression \rangle)$   
 $\quad \quad \quad | (\langle expression \rangle \langle expression \rangle^*)$   
 $\quad \quad \quad | (\text{begin } \langle expression \rangle^*)$   
 $\quad \quad \quad | (\text{set! } \langle variable \rangle \langle expression \rangle)$

- BNF Definition, kontextfreie Grammatik

## 17.1.2 Bausteine von Mini-Scheme Programmen

- Bausteine von Mini-Scheme-Programmen (nach Grammatik):
  - *Syntaktische Variable, Nichtterminalsymbole*  
 $\langle program \rangle$ ,  $\langle form \rangle$ ,  $\langle definition \rangle$ ,  $\langle expression \rangle$ ,  $\langle literal \rangle$ ,  $\langle variable \rangle$
  - Eigentlicher Programmtext, *Terminalsymbole*:  
if, lambda, begin, set!, define, (, )
- Jeder Baustein kann durch einen Term (Baum) dargestellt werden
- Operationssymbol  $\hat{=}$  rechte Regelseite einer BNF Regel
- Stelligkeit: Anzahl der Nichtterminalsymbole in rechter Regelseite
- Bei Wiederholungen (angedeutet durch \*) verwende Liste

### 17.1.3 Symbole und Quote

- Ziel: Scheme-Repräsentation für Namen und Programme
- Sprechlevel: DMdA — fortgeschritten
- Anderes Druckformat in der REPL (Standard-Scheme Druckformat)
  - > (list 1 2 3 4)
  - (1 2 3 4)
  - > empty
  - ()
- So entworfen, dass das Format auch als Eingabeformat verwendet werden kann

## Quote

- Problem: Scheme interpretiert (1 2 3 4) als Operator 1 angewendet auf Argumente 2 3 4
- Abhilfe: Spezieller Operator quote, der das verhindert
- Beispiele

> ()

?: Zusammengesetzte Form ohne Operator

> (1 2 3 4)

?: Operator darf kein Literal sein

> (quote ())

()

> (quote (1 2 3 4))

(1 2 3 4)

## Quote für String, Zahlen und Wahrheitswerte

- quote wirkt auch auf Strings, Zahlen und Wahrheitswerte

```
> (quote "Lang lebe Carla")
```

```
"Lang lebe Carla"
```

```
> (quote 4711)
```

```
4711
```

```
> (quote #t)
```

```
#t
```

- ... ist aber nicht erforderlich:

Diese Literale sind *selbstquotierend*

```
> "Lang lebe Carla"
```

```
"Lang lebe Carla"
```

```
> 4711
```

```
4711
```

```
> #t
```

```
#t
```

## Abkürzung für Quote

- Statt (quote *⟨literal⟩*)  
darf auch  
*'⟨literal⟩*  
geschrieben werden

- Beispiele

> '()

()

> '(1 2 3 4)

(1 2 3 4)

> '"LLC"

"LLC"

> '1945

1945

> '#f

#f

## Namen und Symbole

- Experiment

```
> (lambda (x) x)
```

```
#<procedure>
```

```
> '(lambda (x) x)
```

```
(lambda (x) x)
```

```
> (eisbären lars flocke)
```

```
eisbären: Ungebundene Variable
```

```
> '(eisbären lars flocke)
```

```
(eisbären lars flocke)
```

- Frage: Was sind lambda, x, eisbären, lars, flocke?

⇒ Neuer Datentyp: Symbol; einzige Operation: equal?

- Dient der Repräsentation von Namen in Programmen
- Allgemein für symbolische Daten



## Beispiele mit Symbolen

```
> (define le '(lambda (x) x))
```

```
> (length le)
```

```
3
```

```
> (first le)
```

```
lambda
```

```
> 'lambda
```

```
lambda
```

```
> 'eisbären
```

```
eisbären
```

```
> (symbol? (first le))
```

```
#t
```

```
> (equal? (first le) 'lambda)
```

```
#t
```

```
> (equal? 'lars 'flocke)
```

```
#f
```

## Syntax mit Symbolen und Quote

```
> '(define id (lambda (x) x))  
(define id (lambda (x) x))  
> '(+ 1 2)  
(+ 1 2)  
> (define exp '(+ 1 2))  
> (symbol? (first exp))  
#t  
> (number? (first (rest exp)))  
#t
```

## Quote von Quote

Was, wenn das Programmstück Quote enthält?

```
> '()'
```

```
'()
```

```
> (pair? '())
```

```
#t
```

```
> (first '())
```

```
quote
```

```
> (rest '())
```

```
(())
```

```
> (equal? (quote ()) '())
```

```
#t
```

```
> (equal? (quote (quote ())) '())
```

```
#t
```

## Werte, die durch Quote erzeugt werden können

- quote erlaubt die Konstruktion von *repräsentierbaren Werten*
- **Definition** (repräsentierbarer Wert)
  - Zahlen, Wahrheitswerte, Strings und Symbole sind repräsentierbar.
  - Eine Liste aus repräsentierbaren Werten ist repräsentierbar.
  - Nichts sonst ist repräsentierbar.

## 17.2 Eingebaute Repräsentation von Programmen

- Scheme-Programme sind repräsentierbar

- Beispiel: Die Form

```
(define id (lambda (x) x))
```

wird repräsentiert durch die Liste

```
(list 'define 'id (list 'lambda (list 'x) 'x))
```

- Scheme-Programme verwenden *nicht* die Standardrepräsentation für Terme

⇒ `define-record-procedures` kommt nicht zur Anwendung

⇒ (Konstruktoren,) Tests und Selektoren für Scheme-Programme müssen selbst programmiert werden

## Repräsentation von Variablen

- Eine Variable  $\langle variable \rangle$  wird durch ein Symbol repräsentiert.

```
(define expression-variable?  
  symbol?)
```

```
(define variable-name  
  (lambda (x) x))
```

## Repräsentation von Literalen

- Ein Literal  $\langle literal \rangle$  wird durch sich selbst repräsentiert.
- Ausnahme: Symbole (benötigt quote)

```
(define expression-literal?  
  (lambda (x)  
    (or (number? x) (string? x) (boolean? x))))
```

```
(define literal-value  
  (lambda (x) x))
```

## Standardtest für zusammengesetzte Ausdrücke

- jeder zusammengesetzte Ausdruck ist eine Liste
- die Art des Ausdrucks wird durch das erste Element bestimmt (ein Symbol)
- Ausnahme: Kombination

```
(define test-form
  (lambda (sym)
    (lambda (x)
      (and (pair? x) (equal? (first x) sym)))))
```



## Repräsentation von quote

- `(quote <rep.value>)`
- wird repräsentiert durch eine Liste  
`(list 'quote \meta{rep.value})`

```
(define expression-quote?
```

```
  (test-form 'quote))
```

```
(define quote-value
```

```
  (lambda (exp)
```

```
    (first (rest exp))))
```

## Repräsentation von if

- (if *<expression>* *<expression>* *<expression>*) wird repräsentiert durch eine Liste  
(list 'if exp1 exp2 exp3)  
wobei exp1, exp2, exp3 Ausdrücke sind.

```
(define expression-if?  
  (test-form 'if))  
(define if-condition  
  (lambda (x) (first (rest x))))  
(define if-consequent  
  (lambda (x) (first (rest (rest x)))))  
(define if-alternative  
  (lambda (x) (first (rest (rest (rest x))))))
```

## Repräsentation von Anwendungen

- ( $\langle operator \rangle \langle operand \rangle^*$ )
- auch: Applikation

```
(define application-rator  
  first)
```

```
(define application-rands  
  (lambda (x) (rest x)))
```

## 17.3 Auswertung von Mini-Scheme-Ausdrücken

```
(define eval-exp
  (lambda (exp env)
    (letrec ((eval (lambda (exp)
                     (cond
                      ((expression-variable? exp)
                       (variable-value env (variable-name exp)))
                      ((expression-literal? exp)
                       (literal-value exp))
                      ((expression-if? exp)
                       (if (eval (if-condition exp))
                           (eval (if-consequent exp))
                           (eval (if-alternative exp))))
                      (else
                       ;must be application
                       (let ((rator (eval (application-rator exp)))
                           (rands (map eval (application-rands exp))))
                         (apply-procedure rator rands)))))))
      (eval exp))))
```

## Hilfsfunktionen

- Wert einer Variable

```
(define variable-value  
  (lambda (env var)  
    (lookup env var)))
```

- Anwendung einer Funktion

```
(define apply-procedure  
  (lambda (fun vals)  
    (apply fun vals)))
```

## Repräsentation der Umgebung

- Ein *Eintrag* ist ein Wert

`(make-entry x v),`

wobei `x` ein Symbol ist und `v` ein Wert

```
(define-record-procedures-2 entry
```

```
  make-entry entry?
```

```
  (entry-var entry-value))
```

- Ein *Frame* ist ein Wert

`(make-frame enclosing entries),`

wobei `enclosing` das Frame des umschließenden Gültigkeitsbereichs ist und

`entries` die Liste der Einträge für den aktuellen Gültigkeitsbereich ist

```
(define-record-procedures-2 frame
```

```
  make-frame frame?
```

```
  (frame-enclosing (frame-entries set-frame-entries!)))
```

## Vordefinierte Funktionen in der Umgebung

```
(define initial-env
  (make-frame #f
    (list (make-entry '+ +)
          (make-entry '- -)
          (make-entry '* *)
          (make-entry '/ /)
          (make-entry '= =)
          (make-entry 'odd? odd?)
          (make-entry 'not not)
          (make-entry 'zero? zero?))))
```

## Beispiele

```
(eval-exp '42 initial-env)
```

```
=> 42
```

```
(eval-exp '(- 42) initial-env)
```

```
=> -42
```

```
(eval-exp '(/ 42) initial-env)
```

```
=> 1/42
```

```
(eval-exp '(if (odd? 5) 0 1) initial-env)
```

```
=> 0
```

```
(eval-exp '(if (odd? 6) 0 1) initial-env)
```

```
=> 1
```



## 17.4 lambda-Ausdrücke und Closures

### 17.4.1 Syntaktische Repräsentation

- Ein lambda Ausdruck ist eine Liste mit
  - erstem Element 'lambda
  - zweitem Element: eine Liste von Symbolen
  - drittem Element: ein Ausdruck *<expression>*

- Typprädikat und Selektoren:

```
(define expression-lambda?
```

```
  (test-form 'lambda))
```

```
(define lambda-vars
```

```
  (lambda (x) (first (rest x))))
```

```
(define lambda-body
```

```
  (lambda (x) (first (rest (rest x)))))
```

## 17.4.2 Direkte Interpretation von lambda-Ausdrücken

- Interpretation: Erweitere Fallunterscheidung in eval-exp um

```
((expression-lambda? exp)
 (make-procedure (lambda-body exp) (lambda-vars exp) env))
```

- Einfachste Möglichkeit:  
Repräsentiere lambda durch lambda

```
(define make-procedure
 (lambda (exp vars env)
 (lambda vals
 (let ((extended-env
 (make-frame env (map make-entry vars vals))))
 (eval-exp exp extended-env))))))
```

### 17.4.3 Datenstruktur für Funktionen

- Repräsentation von `lambda` durch `lambda` ist unbefriedigend.
- ⇒ kein Einblick in die wirkliche Implementierung von Funktionen
- Entwerfe **Datenstruktur** (ohne Verwendung von Funktionen) zur Repräsentation von Funktionen

## Closures

Betrachte die vorige Implementierung:

```
(define make-procedure
  (lambda (exp vars env)
    (lambda vals
      (let ((extended-env
             (make-frame env (map make-entry vars vals))))
        (eval-exp exp extended-env))))))
```

- Was ist notwendig zur Auswertung einer Funktion nach der Einsetzungsregel?
  - Der Rumpf `exp`, in den eingesetzt wird.
  - Die Variablen `vars`, für die eingesetzt wird.
  - Die Umgebung `env`, d.h., die Werte der in `exp` auftretenden Variablen.
- Also: Fasse diese drei Werte zu einer Datenstruktur **Closure** zusammen!

```
(define-record-procedures-2 closure
  make-closure closure?
  (closure-exp closure-vars closure-env))
```

## Implementierung von Funktionen mit Closures

```
(define make-procedure
  make-closure)
(define apply-procedure
  (lambda (fun vals)
    (if (closure? fun)
        (let* ((exp (closure-exp fun))
              (vars (closure-vars fun))
              (env (closure-env fun))
              (extended-env
                 (make-frame env (map make-entry vars vals))))
          (eval-exp exp extended-env))
        (apply fun vals))))
```

- Verschiebt den Aufruf von `eval-exp` aus der Interpretation von `lambda` in die Interpretation von Funktionsaufrufen.
- Einsetzung der Werte in den Rumpf der Funktion geschieht über die Umgebung

## Beispiel

```
> (evaluate-expression '((lambda (five) (lambda (x) five)) 5) initial-env)
#<record:closure
  five
  (x)
  #<record:frame
    #<record:frame
      f
      #<record:entry + #<primitive:+>>
      #<record:entry - #<primitive:->>
      #<record:entry * #<primitive:*>>
      #<record:entry / #<primitive:/>>
      #<record:entry = #<primitive:=>>
      #<record:entry odd? #<primitive:odd?>>
      #<record:entry not #<procedure:DMdA-not>>
      #<record:entry zero? #<primitive:zero?>>>>
    (#<record:entry five 5>>>
```

## 17.5 Toplevel Definitionen

- Spezialbehandlung:

Jede Toplevel Definition fügt dem Toplevel-Frame einen neuen Eintrag hinzu

; Auswerten einer Definition

; evaluate-definition : definition env -> unspecified

; Effekt: erweitert env um neuen Eintrag

```
(define evaluate-definition
```

```
  (lambda (d env)
```

```
    (let ((x (definition-variable d))
```

```
          (e (definition-expression d))))
```

```
      (let ((v (evaluate-expression e env))))
```

```
        (env-extend env x v))))))
```

## Erweitern der Toplevel-Umgebung

```
; extend the environment with a new entry (destructively)
; env-extend : env var value -> unspecified
(define env-extend
  (lambda (env x v)
    (set-frame-entries!
     env (make-pair (make-entry x v)
                    (frame-entries env)))))
```



## Programmauswertung

```
; Wertet ein Programm aus
; program-run : p env -> list(value)
(define program-run
  (lambda (f* env)
    (if (empty? f*)
        empty
        (let ((f (first f*))
              (f* (rest f*)))
          (if (definition? f)
              (begin (evaluate-definition f env)
                     (program-run f* env))
              (let* ((v (evaluate-expression f env))
                    (v* (program-run f* env)))
                (make-pair v v*))))))))))
```

## 17.6 Veränderliche Variable und set!

*Any problem in computer science can be solved with another level of indirection. [But that usually will create another problem.]*

David John Wheeler, 1927–2004

- Erfordert Änderungen am Interpreter
  - Hinzufügen der Ausdrücke `begin` und `set!`
  - Veränderung an den Umgebungseinträgen: Einfügen von Locations
- Eine *Location* ist ein Wert  
(`make-location v`),  
wobei `v` ein beliebiger Wert ist, der geändert werden kann.  

```
(define-record-procedures-2 location
  make-location location?
  ((location-content set-location-content!)))
```
- Revision: Ein *Eintrag* ist ein Wert  
(`make-entry x l`),  
wobei `x` ein Symbol ist und `l` eine Location.

## Zugriff auf Variable

- Jede Variable wird zunächst an eine Location gebunden.
- ⇒ Bei Auswertung der Variable muss die Location dereferenziert werden

- Neudefinition:

```
(define variable-value
  (lambda (env var)
    (location-content (lookup env var))))
```

## Anwenden von Prozeduren

- Bindungen werden nur in apply-procedure erzeugt
- Neue Version davon:

```
(define apply-procedure-loc
  (lambda (proc args)
    (cond
      ((closure? proc)
       (let ((body (closure-exp proc))
             (vars (closure-vars proc))
             (env (closure-env proc)))
         (let ((new-env
                (make-frame
                 env
                 (map make-binding vars args))))
           (eval-exp body new-env))))
      (else
       (apply proc args))))))
```

## Erzeugen von Bindungen

- Jeder gebundene Wert muss in eine eigene Location eingepackt werden.

```
; create a binding from a variable to a location
; filled with an initial value
; make-binding : variable value -> entry
(define make-binding
  (lambda (var arg)
    (make-entry var (make-location arg))))
```

## Implementierung von set!

- ... ändert nur den Inhalt der Location

```
((expression-set? exp)
 (let ((x (set-variable exp))
       (e (set-expression exp)))
   (let* ((val (eval e))
          (loc (env-lookup env x)))
     (set-location-content! loc val))))
```

## Veränderte initiale Umgebung

```
; initiale Umgebung mit locations
(define initial-env-loc
  (make-frame #f
    (list (make-binding '+ +)
          (make-binding '- -)
          (make-binding '* *)
          (make-binding '/ /)
          (make-binding '= =)
          (make-binding 'odd? odd?)
          (make-binding 'not not)
          (make-binding 'zero? zero?))))))
```

## Beispiel

```
> (eval-exp '(+ 17 4) initial-env-loc)
```

```
21
```

```
> (eval-exp '((lambda (to) 0)
              (+ 17 4))
      initial-env-loc)
```

```
0
```

```
> (eval-exp '((lambda (to)
                (begin
                  (set! to (+ to 1))
                  to))
              (+ 17 4))
      initial-env-loc)
```

```
22
```



## 17.7 Zusammenfassung

- Symbole und Quote
- Repräsentation von Scheme-Programmen
- Auswertung von Scheme-Ausdrücken
- Implementierung von Funktionen durch Funktionen
- Implementierung von Funktionen durch Closures
- Für `set!`: Bindungen erfolgen an Adressen, deren Inhalt geändert wird