16 Interpretation

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

16.1 Programme als Datenstruktur

16.1.1 Mini-Scheme

$$
\begin{array}{lcl}
\langle \textit{program} \rangle & ::= & \langle \textit{form} \rangle^* \\
\langle \textit{form} \rangle & ::= & \langle \textit{definition} \rangle \\
& | & \langle \textit{expression} \rangle \\
\langle \textit{definition} \rangle & ::= & \texttt{(define } \langle \textit{variable} \rangle \ \langle \textit{expression} \rangle \texttt{)} \\
\langle \textit{expression} \rangle & ::= & \langle \textit{literal} \rangle \\
& | & \langle \textit{variable} \rangle \\
& | & \texttt{(if } \langle \textit{expression} \rangle \ \langle \textit{expression} \rangle \ \langle \textit{expression} \rangle \texttt{)} \\
& | & \texttt{(lambda (} \langle \textit{variable} \rangle^* \texttt{)} \ \langle \textit{expression} \rangle \texttt{)} \\
& | & \texttt{(} \langle \textit{expression} \rangle \ \langle \textit{expression} \rangle^* \texttt{)} \\
& | & \texttt{(begin } \langle \textit{expression} \rangle^* \texttt{)}
\end{array}
$$

- BNF Definition, kontextfreie Grammatik

16.1.2 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`, `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

16.1.3 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: Eingabeformat für allgemeine Liste
- 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 "Elvis lebt")
"Elvis lebt"
> (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`?

$\Rightarrow$ Neuer Datentyp: **Symbol**, Vertrag `symbol`

  - Operationen darauf: `equal?`, `symbol->string`, `string->symbol`
  - Repräsentation von Namen in Programmen
  - 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.

16.2 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

$\Rightarrow$ `define-record-procedures` kommt nicht zur Anwendung

$\Rightarrow$ (Konstruktoren,) Tests und Selektoren für Scheme-Programme müssen selbst programmiert werden

Repräsentation von Variablen

- Eine Variable ⟨*variable*⟩ wird durch ein Symbol repräsentiert.

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

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

Repräsentation von Literalen

- Ein Literal $\langle \textit{literal} \rangle$ 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` ⟨*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 Funktionsanwendungen (-applikationen)

- (⟨*operator*⟩ ⟨*operand*⟩*)

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

16.3 Auswertung von Mini-Scheme-Ausdrücken

```
(define eval-exp
  (lambda (exp)
    (letrec ((eval (lambda (exp)
      (cond
       ((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))))
```

Hilfsfunktion: Anwendung einer Prozedur

```
(: apply-procedure  ((%v ... -> %v) (list %v) -> %v))
(define apply-procedure
  (lambda (fun vals)
    (apply fun vals)))
```

Variable und Umgebungen

- Der Wert einer Variablen hängt vom Kontext (bzw. der Interpretation der Variablen) ab

$\Rightarrow$ kann nicht im Interpreter generiert werden

$\Rightarrow$ übergebe die Interpretation als zusätzlichen **Umgebungsparameter** `env`

- Konzeptuell: Umgebung = Abbildung von Namen auf Werte

16.4 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

```
(: variable-value (frame symbol -> %v))
(define variable-value
  (lambda (env var)
    (lookup env var)))
```

Repräsentation der Umgebung

- Ein *Eintrag* repräsentiert eine Bindung. Es 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* repräsentiert einen Gültigkeitsbereich. Es 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

```
(: initial-env frame)
(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
```

16.5 `lambda`-Ausdrücke und Closures

16.5.1 Syntaktische Repräsentation

- Ein `lambda` Ausdruck ist eine Liste mit
  - erstem Element `'lambda`
  - zweitem Element: eine Liste von Symbolen (den formalen Parametern)
  - 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)))))
```

16.5.2 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)))))
```

16.5.3 Datenstruktur für Funktionen

- Repräsentation von `lambda` durch `lambda` ist unbefriedigend.

$\Rightarrow$ 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

```
> (eval-exp '((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>)>>
```

16.6 Toplevel Definitionen

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

```
; Auswerten einer Definition
(: evaluate-definition ((predicate definition?) frame -> %unspecified))
; Effekt: erweitert env um neuen Eintrag
(define evaluate-definition
  (lambda (d env)
    (let ((x (definition-variable d))
          (e (definition-expression d)))
      (let ((v (eval-exp e env)))
        (env-extend env x v)))))
```

Erweitern der Toplevel-Umgebung

```
; extend the environment with a new entry (destructively)
(: env-extend  (frame symbol %v -> %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 ((list %form) frame -> (list %v)))
(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*)))))))
```

16.7 Referenzen

- Erinnerung:

```
(define-record-procedures-parametric-2 ref ref-of
  make-ref ref?
  ((get-ref set-ref!)))
```

- Hinzufügen zum Interpreter durch Erweiterung der initialen Umgebung

```
(make-entry 'make-ref make-ref)
(make-entry get-ref get-ref)
(make-entry set-ref! set-ref!)
```

- Hinzufügen von `(begin ...)`

`begin` **Ausdrücke**

```
(define expression-begin?
  (test-form 'begin))
(define begin-exprs
  (lambda (x) (rest x)))
```

- Auswertung "der Reihe nach"
- Ergebnis: Wert des letzten Ausdrucks

16.8 Zusammenfassung

- Symbole und Quote
- Repräsentation von Scheme-Programmen
- Auswertung von Scheme-Ausdrücken
- Implementierung von Funktionen durch Funktionen
- Implementierung von Funktionen durch Closures