

SYNTAKTISCHE ABSTRAKTION: MACROS

Am Anfang stand: **Funktionale Abstraktion** als entscheidender Schritt zum Aufbau komplexer Programme.

Benennung und Parametrisierung semantisch abgeschlossener Programmteile: *Wiederverwendbarkeit*.

Nun: **Abstraktion auf syntaktischer Ebene**

Zunächst eingeführt in Assemblersprachen.

Verallgemeinert in funktionalen Programmiersprachen wie Scheme:

Definition von *Spezialformen* mit eigenen Bindungs- und Auswertungsregeln.

Programmtransformation

Die Menge von Regeln bzw. die Prozedur, die spezifiziert, wie ein Macro-Aufruf in einen Programmtext umgesetzt wird, heißt **Transformer**.

Da der eingesetzte Programmtext i.a. länger als der Macro-Aufruf selbst ist, spricht man auch von Macro-Expansion.

Macro-Expansion ist ein Fall von *Programmtransformation*: syntaktische Abbildung von Programmtexten in Programmtexte.

Beispiel:

```
(let ((a1 v1) ...) <body>)
```

```
⇒ ((lambda (a1 ...) <body>) v1 ...)
```

kann nicht als Prozedur definiert werden wegen Auswertung in Applikationsordnung (call-by-value)

— benötigt wird: Normalordnung der Auswertung.

ZENTRALE ÜBERLEGUNG:

1. Ein gegebenes Stück Programmtext wird — analog zur funktionalen Abstraktion — mit einem Namen, dem *syntaktischen Schlüsselwort*, versehen und durch Variablen parametrisiert.
2. Abstraktionsoperator ist der *Macro-Operator*.
3. Wird bei der *Syntaxanalyse* (vor der Auswertung!) der *Aufruf* (*Verwendung*, “*use*”) erreicht, so wird dieser durch den Programmtext *ersetzt* und die Aktualparameter werden unausgewertet für die Formalparameter eingesetzt.
4. Zum Zeitpunkt der Programmausführung ist von der textuellen Abstraktion nichts mehr sichtbar: es liegt der “expandierte” Programmtext vor.

Spezialformen

Durch das Macrokonzept werden zwei Ziele erreicht:

- Erklärung vorhandener Spezialformen
- Einführung neuer Spezialformen

In beiden Fällen wird das Programm durch iterierte Transformationen auf eine macro-freie Form zurückgeführt (“Kernsprache”).

Welche Spezialformen sollte ein Scheme-System vorsehen?

- Bindungskonstrukte: `lambda`, `let`, `do`, ...
- Kontrollkonstrukte: `cond`, `if`, `and`, `apply`, ...
- Spezialformen zur Definition *neuer* Spezialformen, z.B. `define-syntax`

Spezialformen: Realisierung

Entkopplung der Mechanismen der syntaktischen Erweiterung und der Auswertung — Macro-Expansion durch einen während der Syntaxanalyse aktivierten Präprozessor.

Fragen:

Wie werden die Transformationen beschrieben?

Wie und wann werden sie ausgeführt?

Macro-Expansion

Die Stellen im Programm, an denen Transformationen vorzunehmen sind, sind Terme der Form (Macro-Aufrufe, Verwendungen):

(<keyword> <parameter-1> ... <parameter-n>)

Macro-Aufrufe werden in drei Schritten ausgeführt:

1. Suche den Transformer zum <keyword>
2. Instantiierung des Transformers mit den Parametern
3. Einsetzung des instantiierten (expandierten) Ausdrucks an die Stelle des Macro-Aufrufs.

Ein Transformer kann grundsätzlich eine *Funktion* ("individueller Ansatz") oder eine *Datenstruktur* ("zentralistischer Ansatz") sein

Der klassische individuelle Ansatz

`define-macro` à la Common LISP

Der Transformer ist ein Funktionsobjekt

- Zu jedem <keyword> existiert ein Funktionsobjekt
- Kommt in einem Term des Programms ein <keyword> an Funktorposition vor, wird das zugehörige Funktionsobjekt mit den geg. Aktualparametern aufgerufen.
- Der Wert, den diese Funktion liefert, wird im Programmtext an der Stelle des Macro-Aufrufs eingesetzt.

Da solche Funktionsobjekte üblicherweise komplexe verschachtelte Terme zu erzeugen haben, werden sie durch die entsprechende große Anzahl von Konstruktionsoperationen schnell unübersichtlich.

Stattdessen wäre es eine große Erleichterung, wenn man die zu erzeugenden Strukturen in der gewünschten Form (externe Repräsentation) direkt hinschreiben könnte und aus dieser Notation dann die entsprechenden Terme zu ihrer Erzeugung *generiert* würden.

Zu erzeugende Strukturen haben i.d.R. *konstante* und *variable* Teile. Wegen letzterer kann die Struktur nicht einfach quotiert werden.

Benötigt: "Quotierung bis auf" variable Teile.

Quasi-Quotierung

Das Zeichen ' ("Backquote") wird von der Einlese-Routine besonders interpretiert (sog. "Read-Macro"): Wird ein Ausdruck von Backquote angeführt, wirkt es wie ein Quote-Zeichen mit folgender Ausnahme:

- Kommt im Ausdruck als zweites ausgezeichnetes Zeichen das Komma vor, wird für den folgenden Teilausdruck die Quotierung aufgehoben, dieser ausgewertet, und der resultierende Wert an dessen Stelle eingesetzt.
- Steht hinter einem Komma als weiteres ausgezeichnetes Zeichen @ , wird die folgende Listenstruktur nicht als ganze, sondern als *Folge* ihrer Werte eingesetzt.

Quasi-Quotierung (3)

Mit Quasi-Quotierung vereinfacht:

```
'(DEFINE (,name ,arg1 ,arg2)
  (COND ((NULL? ,arg1) ,result)
        (ELSE
         (,fun ,expr
          (,name (CDR ,arg1) ,arg2)))) )
```

Das Macro-Zeichen ' ist mit einem Transformer assoziiert, das es in die Spezialform `quasiquote` expandiert, analog die eingebetteten Macro-Zeichen , mit `unquote` , und ,@ mit `splice-unquote` .

Quasi-Quotierung (2)

Beispiel: Es soll ein Schema der folgenden Form generiert werden (konstante Teile in Groß-, variable Teile in Kleinschreibung):

```
(DEFINE (name arg1 arg2)
  (COND ((NULL? arg1) result)
        (ELSE (fun expr (name (CDR arg1) arg2))))))
```

wäre ohne Quasi-Quotierung zu schreiben:

```
(list 'DEFINE
  (list name arg1 arg2)
  (list 'COND
    (list (list 'NULL? arg1) result)
    (list 'ELSE
      (list fun expr
        (list name (list 'CDR arg1) arg2)))) )
```

DEFINE-MACRO

Der individuelle Ansatz ist in STk auf einfache Weise in Anlehnung an Common LISP durch die Spezialform `define-macro` realisiert:

```
(define-macro (<keyword> <varlist>) <expander>)
```

Es gilt:

- `<keyword>` ist das syntaktische Schlüsselwort der Spezialform und wird Name des `<expander>`.
- `<expander>` ist ein Funktionsrumpf und bildet zusammen mit der `<varlist>` ein Funktionsobjekt (ohne `lambda!`).

Vor seiner Aktivierung werden die Aktualparameter (unausgewertet) an die Formalparameter gebunden.

DEFINE-MACRO-Beispiel (1)

```
(define-macro (delay-memo para) ; Macro: Verzögerte Auswertung
  '(let ((done #f)
        (res '()))
    (lambda ()
      (cond (done res)
            (else (set! res ,para)
                  (set! done #t)
                  res))))))

(define (force-memo obj) (obj))

(define a 10) ==> unspecified

(define b (delay-memo a)) ==> unspecified
```

Der zentralistische Ansatz

Der *Transformer* ist eine Datenstruktur; Transformation wird durch Schema- bzw. Mustervergleich ("Pattern Matching") beschrieben:

- Zu jedem <keyword> existiert (mindestens) ein Paar von Schema (Muster für den Macro-Aufruf) und Transformer.
- Kommt in einem Term des Programms ein <keyword> an Funktorposition vor, wird zwischen diesem Term (Macro-Aufruf) und dem dem <keyword> zugeordneten Schema ein Mustervergleich durchgeführt.
- Verläuft der Vergleich erfolgreich, resultiert eine Variablensubstitution für die Variablen im Schema, die dann zur Instantiierung des Transformers verwendet wird.

DEFINE-MACRO-Beispiel (2)

```
(set! a 9) ==> unspecified

(force-memo b) ==> 9

(set! a 8) ==> unspecified

(force-memo b) ==> 9
```

delay-memo liefert "verzögertes" Objekt: es repräsentiert suspendierte Auswertung von para.

Erste Anwendung von force-memo darauf: para wird in seiner Definitionsumgebung ausgewertet.

Wiederholte Anwendung von force-memo auf das verzögerte Objekt liefert denselben Wert, es wird also "memorisiert".

Bindungskonstrukte

für syntaktische Schlüsselwörter:

define-syntax, let-syntax, letrec-syntax

binden syntaktische Schlüsselwörter an Macro-Transformer (analog zu den Bindungskonstrukten, die Variablen an Lokationen binden).

define-syntax (\Rightarrow R⁵RS)

```
(define-syntax <keyword> <transformerspec>)
```

Die <transformerspec> ist ein Ausdruck der Mustersprache:
(syntax-rules < literals > (< pattern > < template >) ...)
wobei Ellipsen durch die spezielle Mustervariable "... " ausgedrückt werden können.

Beispiel:

```
(define-syntax let*  
  (syntax-rules ()  
    ((let* () body1 body2 ...)  
     (let () body1 body2 ...))  
    ((let* ((name1 val1) (name2 val2) ...) body1 body2 ...)  
     (let ((name1 val1)  
           (let* ((name2 val2) ...) body1 body2 ...))))))
```

Hygienische Macro-Expansion

Die Erzeugung ungewollter Variablenbindungen bei der Macro-Expansion zu verhindern und diese damit sicher zu machen, sollte systemseitig erfolgen.

Verfahren: *Hygienische Macro-Expansion* (Kohlbecker)

folgt dem Prinzip, dass generierte Identifikatoren nur Variablen binden dürfen, die im selben Transformationsschritt erzeugt werden:

Variablen werden mit einem Zeitstempel markiert; diese werden nach der Expansion unter geeigneter Variablenumbenennung wieder entfernt.

Das Capturing-Problem

Mit einem naiven Macro-Expansions-Algorithmus können unbeabsichtigte Variablenbindungen auftreten, z.B.

```
(define-syntax new-or  
  (syntax-rules  
    ()  
    ((new-or e1 e2)  
     (let ((v e1)) (if v v e2))))))
```

(new-or #f v) würde erzeugen: (let ((v #f)) (if v v v)) !!!

Wie kann Abhilfe geschaffen werden?

Verwendung exotischer Variablennamen wie %%%v? Nein — nicht sicher!

Verwendung von (gentemp), das einen neuen eindeutigen Identifikator erzeugt? Nein — nicht Standard; ebenfalls nicht sicher!

Syntactic Closures

Alternative Lösung des Capturing-Problems (Bawden und Rees) durch Unterscheidung von zwei Umgebungstypen:

Die *syntaktische Umgebung* bildet Identifikatoren auf Variablen ab und enthält die Interpretation der syntaktischen Schlüsselwörter.

Die *Wert-Umgebung* bildet Variablen auf Lokationen ab, die deren Werte enthalten.

Syntaktische Closures werden zur Auflösung von Bereichsproblemen zur Macro- Expansionszeit eingeführt:

Eine syntaktische Closure besteht aus einer Umgebung (Liste von Namen) und einem Ausdruck.

Alle im Ausdruck vorkommenden Namen außer den in dieser Umgebung enthaltenen werden relativ zur aktuellen Umgebung interpretiert. Die Bedeutungen der Namen in der Liste (Umgebung) bleiben uninterpretiert und werden erst später bestimmt (\Rightarrow Parametrisierung des Ausdrucks): "call-by-context".

Vor- und Nachteile beider Vorgehensweisen:

- Hygienische Macro-Expansion ist nicht besonders effizient (quadratischer Zeitaufwand),
- Syntaktische Closures sind auf einer relativ systemnahen Ebene angesiedelt, daher nicht einfach zu benutzen und erlauben nicht, eine automatische mustergesteuerte Macro-Expansion zu implementieren.

Standardisierung in R⁵RS

Vereinigung der Vorteile beider: "Macros that work" (Clinger und Rees).

Syntaktische Schlüsselwörter müssen nicht mehr reservierte Namen sein (die anders als "normale" Identifikatoren zu behandeln sind). Bindungskonstrukte `define-syntax` etc. binden syntaktische Schlüsselwörter an Transformer. Die Macros erfüllen die "Hygiene"-Bedingung und sind *referentiell transparent*: Namenskonflikte werden durch Umbenennung vermieden, für freie Variable gelten ihre Bindungen zum Zeitpunkt der Spezifikation des Transformers. Weiterhin gibt es eine kompatible Möglichkeit, auf niedrigerer Abstraktionsstufe Transformer zu definieren, die nicht in der Mustersprache ausgedrückt werden können. Damit wird auch die Mustersprache selbst implementiert.

Aktivierungsumgebung des Transformers

wird durch syntaktische Closures eindeutig festgelegt.

Interessant, wenn der Transformer freie Variablen enthält: Die statische Umgebung ist die, in der das Macro definiert wurde. Wie aber steht es um die dynamische Umgebung, d.h. soll die Macro-Expansion Bestandteil des Programmablaufs sein?

Man könnte einen Macro-Aufruf bereits expandieren, wenn die ihn enthaltende Funktion definiert wird ("destruktive Macro-Expansion") oder erst dann, wenn diese Funktion aufgerufen wird (dann muss die Expansion bei jedem Funktionsaufruf erneut durchgeführt werden). In manchen LISP-Systemen wird die Entscheidung dem Programmierer überlassen.

Achtung: Compilation; Debugging!

Objekt-orientierte Spracherweiterungen

Mit Macros ist es relativ leicht möglich, Spracherweiterungen zu definieren. Hier als Beispiel: **Objekt-orientierte Spracherweiterungen**.

Zur Erinnerung: **Closures** sind Schlüsselkonzept zur Implementierung objekt-orientierter Programmierung.

Beispiel:

```
(define (MAKE-POINT the-x the-y)
  (define (get-x) the-x)           ;; a "method"
  (define (get-y) the-y)

  (define (set-x! new-x)
    (set! the-x new-x)
    the-x)
```

Hinzufügen der Vererbung

“Vererbung” bedeutet, dass ein Objekt spezialisiert werden kann durch Erweiterung oder Verschattung des “Verhaltens” eines anderen Objekts.

Hier einfach: Delegation!

```
(define (MAKE-POINT-3D a b the-z)
  (let ((point (make-point a b)))

    (define (get-z) the-z)

    (define (set-z! new-value)
      (set! the-z new-value)
      the-z)
```

```
(define (set-y! new-y)
  (set! the-y new-y)
  the-y)

(define (self message)
  (case message
    ((x)      get-x) ;; return local function
    ((y)      get-y)
    ((set-x!) set-x!)
    ((set-y!) set-y!)
    (else
     (error "POINT: Unknown message ->" message))))

self ;; return value of make-point is the dispatch function
)
```

```
; interface functions

(define (x obj) ((obj 'x))

(define (set-x! obj new-val)
  ((obj 'set-x!) new-val))

(define p1 (make-point 132 75))
(define p2 (make-point 132 57))
(set-x! p1 12) => 12
(x p1)        => 12
(x p2)        => 132 ; p1 and p2 share code but
                  ; have different local data
```

```
(define (self message)
  (case message
    ((z)      get-z)
    ((set-z!) set-z!)
    (else (point message))))

self)

(define p3 (make-point-3d 12 34 217))

(x p3)      => 12
(z p3)      => 217
(set-x! p3 12) => 12
(set-x! p2 12) => 12
(set-z! p3 14) => 14
```

Wo liegt das Problem?

Das gezeigte Vorgehen reicht im Prinzip aus, aber . . .

Wie können wir sagen, welche Funktionen Punkte repräsentieren und welche nicht? Wir können ein POINT?-Prädikat, definieren, aber nicht alle Scheme-Objekte werden eine 'point? Nachricht annehmen: Die meisten werden eine Fehlermeldung produzieren, aber einige werden einfach Unsinn machen.

```
(define (POINT? obj)
  (and (procedure? obj) (obj 'point?)))

(point? list) ==> (point?)
; eine Liste mit dem Symbol 'point?
```

Ziel: Ein System, in das alle Objekte integriert sind und in dem Programmierstile gemischt werden können. Die Konstruktion von Dispatch-Funktionen kann automatisiert werden (. . . Macros).

Weiterhin ist *multiple Vererbung* wünschenswert.

Die Implementierung des Objektsystems soll der Maxime gehorchen, die Schnittstelle von der Implementation zu trennen.

Eine Lösung: YASOS (Yet Another Scheme Object System)

Basisinventar:

- Keine expliziten Klassendefinitionen!
- Einstelliges Prädikat INSTANCE?, um festzustellen, ob ein Objekt eine Instanz ist.
- (DEFINE-PREDICATE <opname?>), ein Generator für Prädikate
- (DEFINE-OPERATION (opname self arg ...) default-body) zur Definition von Operationen, die auf beliebige Datenobjekte anwendbar sind. Das default-Verhalten dient für Objekte, die die Operation *nicht* ausführen können; falls nicht angegeben, entspricht es der Erzeugung eines Fehlers.

- (OBJECT operation) zur Lieferung von Werten, die Instanzen des Objektsystems sind, wobei eine Operation die Form ((opname self arg ...) body) hat.

Es gibt auch eine LET-ähnliche Form für multiple Vererbung:

```
(OBJECT-WITH-ANCESTORS
  ( (ancestor1 init1) ...)
  operation ...) ; bei Ambiguität: Tiefensuche
```

- "Send to super":
(OPERATE-AS component operation composite arg ...) für Fälle, wo man wie ein übergeordnetes Objekt operieren, aber die "self"-Identität behalten möchte (sicheres "Code-Sharing" bei gleichzeitiger Lokalität der Operationen in der Objekthierarchie).

Bemerkung: YASOS ist ein *sehr* einfaches Objektsystem, aber es erfüllt die genannten Forderungen. Der Code ist *zwei* Seiten lang! (⇒ SLIB)

Beispiel: POINTs

```
;; POINT general operations
(define-operation (PRINT obj port)
  (format port ;; if an instance does not have a PRINT op.
    (if (instance? obj) "#<INSTANCE>" "~s")
    obj))

(define-operation (SIZE obj) ;; default behavior
  (cond
    ((vector? obj) (vector-length obj))
    ((list?  obj) (length obj))
    ((pair?  obj) 2)
    ((string? obj) (string-length obj))
    ((char?  obj) 1)
    (else (error "operation not supported: size" obj)) ))
```

```
;; POINT interface
(define-predicate POINT?) ;; #f by default
(define-operation (X obj))
(define-operation (Y obj))
(define-operation (SET-X! obj new-x))
(define-operation (SET-Y! obj new-y))

;; POINT implementation
(define (MAKE-POINT the-x the-y)
  (object
    ((POINT? self) #t) ;; yes, this is a point object
    ((X self) the-x)
    ((Y self) the-y)
    ((SET-X! self val) (set! the-x val) the-x)
    ((SET-Y! self val) (set! the-y val) the-y)))
```

```
((SIZE self) 2)
((PRINT self port)
  (format port "#<point: ~a ~a>" (x self) (y self))) ))

;; 3D POINT interface additions
(define-operation (Z obj))
(define-operation (SET-Z! obj new-z))

;; 3D POINT implementation
(define (MAKE-POINT-3D the-x the-y the-z)
  (object-with-ancestors
    ((a-point (make-point the-x the-y)))
    ((Z self) the-z)
    ((SET-Z! self val) (set! the-z val) the-z)
    ;; override inherited SIZE and PRINT ops.
    ((SIZE self) 3)))
```

```
((PRINT self port)
  (format port "#<3D-point: ~a ~a ~a>"
    (x self) (y self) (z self))) ))

(define P2 (make-point 123 32131))
(define P3 (make-point-3d 32 121 3232))
(size "a string")      => 8
(size p2)              => 2
(point? p2)            => #t
(point? p3)            => #t
(point? "a string")    => #f
(x p2)                => 123
(x "a string")         ERROR: Operation not handled: x "a string"
(print p2 #t)          #<point: 123 32131>
(print p3 #t)          #<3D-point: 32 121 3232>
(print "a string" #t)  "a string"
```

Weitergehende objekt-orientierte Spracherweiterungen

YASOS ist eine kleine effiziente Spracherweiterung, das die wichtigsten Elemente zur objekt-orientierten Programmierung enthält.

Professionelle Objektsysteme bieten eine deutlich größere Ausdruckskraft bei den genannten Sprachmitteln sowie zusätzliche Eigenschaften, die ihre Leistungsfähigkeit signifikant erweitern.

Mit dem Vorbild CLOS (CommonLISP Object System) für Scheme steht in STk eine Variante von "TinyCLOS" als STklos zur Verfügung mit:

- Metaklassen
- multipler Vererbung
- generischen Funktionen
- Multi-Methoden

Implementation beruht auf einem echten Meta-Objektprotokoll.