Programming Languages from Hell: Scheme

Marek Kubica
Technische Universität München

4.6.2010

Kurzer und verständlicher Programmcode ist ein wichtiger Aspekt der Programmierung, da solcher Code vom Programmierer einfacher zu überblicken und verstehen ist. Seit jeher werden daher in der Programmierung zusätzliche Abstraktionen eingeführt, um die zunehmende Komplexität der Systeme überschaubar zu kapseln. Eine mächtige Abstrationsmethode ist die Metaprogrammierung, die Programmierung von Programmen. Scheme ermöglicht Metaprogrammierung über sogenannte Makros, die es erlauben eigene Syntaxkonstrukte einzuführen. Somit können neue Abstraktionen nahtlos in die Sprache integriert werden.

1 Über die Sprache Scheme

Scheme ist ein Dialekt der Lisp-Familie, die sich hauptsächlich dadurch auszeichnet, dass sie auf Listenoperationen ausgelegt sind sowie dass ihre Syntax ebenfalls aus ineinander verschachtelten Listen besteht. Diese Listen werden S-Expressions genannt und haben die Form `(name parameter ...)`. Dadurch besteht in Lisp-Dialekten eine enge Verzahnung zwischen *Code* und *Daten*. Lisp-Dialekte haben üblicherweise ein starkes, dynamisches Typsystem bei dem die Typen nicht implizit ineinander konvertiert werden, jedoch findet keine statische Typprüfung statt. Typannotationen sind in Lisp-Dialekten ebenfalls unüblich.

Scheme ist ein nicht rein-funktionaler Dialekt von Lisp der sowohl Konzepte der funktionalen Programmierung als auch Konzepte der imperativen Programmierung beinhaltet. Über die im folgenden diskutierten Makros lassen sich eine Vielzahl anderer Paradigmen in die Sprache integrieren.

2 Makros

Makros werden verwendet, um eigene Syntaxerweiterungen zu ermöglichen. Gerade im Zuge von Domain Specific Languages ist es wünschenswert, dem Programmierer die Freiheit zu geben, neben domänenspezifischen Datenstrukturen und Funktionen auch domänenspezifische Syntax bereitzustellen.

Ein Beispiel für diese Art von DSLs ist objektorientierte Programmierung. Wie C bringt Scheme keine syntaktische Unterstützung für Objektorientierung mit, ermöglicht aber die Implementierung verschiedenartiger Objektsysteme, die durch die Verwendung von Makros auch syntaktische Integration in den Sprachkern bieten.

Ein Beispiel für Makros in anderen Sprachen ist die Programmiersprache C. C-Makros werden von Präprozessor ausgewertet, der den Quellcode nach bestimmten Sequenzen durchsucht und sie substituiert. Dadurch ist es kompliziert fehlerfreie Makros zu schreiben, die Abstraktion bewegt sich auf niedrigem Abstraktionslevel und fehlerhafte Makros können zu ungültigem Code expandieren.

Makros in Scheme operieren hingegen auf der Listenstruktur des Programmes. Der Programmcode wird als Daten aufgefasst und kann daher von Scheme-Code verarbeitet werden ohne dass eine zusätzliche Template-Sprache, wie der C-Präprozessor, notwendig wird. Diese Verarbeitung geschieht über spezielle Funktionen, die Scheme-Code als Listen entgegennehmen, verarbeiten und wiederum Scheme-Code aus Listen ausgeben.

```
1 (define-syntax postfixed
2   (syntax-rules ()
3       ((_ (operands ... operator)) (operator (postfixed operands) ...))
4       ((_ atom) atom)))
```

Listing 1: Das `postfixed`-Makro

Ein Beispiel der Makro-Syntax in Scheme ist das Makro in Listing 1. Dieses Makro ermöglicht die Verwendung von Postfix-Notation in Scheme, indem es Postfix-Notation in Prefix-Notation transformiert.

In Zeile 1 wird ausgesagt, dass ein Makro mit dem Namen `postfixed` definiert werden soll. Die zweite Zeile weist dem Makro-Namen einen Syntax-Transformer zu – dieser wird von `syntax-rules` zurückgegeben. Dies ist die eigentliche Funktion, die Scheme-Code transformiert, sie wird in diesem Fall von `syntax-rules` aus den Regeln generiert. `syntax-rules` nimmt als ersten Parameter eine Liste von Keywords (im Fall von `postscheme` werden keine Keywords genutzt). Die folgenden Listen bestehen jeweils aus Paaren von `((muster) (expansion))`. Auf den Mustern wird Pattern-Matching ausgeführt, ähnlich zu Sprachen aus der ML-Familie. Das vorliegende Makro definiert in Zeile 3 ein Muster das aus dem Makronamen besteht (dieser wird als `_` abgekürzt) und darauffolgend einer Liste von beliebig vielen Argumenten. Die `...` bedeuten, dass alle

Elemente dieser Liste in eine neue, an `operands` gebundene Liste gesetzt werden. Das letzte Element der ursprünglichen Liste erhält eine Sonderbehandlung, es wird an den Namen `operator` gebunden. Danach ist der Muster-Teil der ersten Regel zu Ende, es folgt die Expansion. Diese konstruiert eine neue Liste mit `operator` an der ersten Stelle. Darauf folgen rekursive Aufrufe des Makros auf die einzelnen Elemente der `operands`-Liste die durch die `...` ausgelöst werden. `...` wechselt die Bedeutung, je nachdem ob es im Muster-Teil oder im Expansions-Teil der Regel ist. Zeile 4 fährt mit der Trivialregel fort, die ein Element zu sich selbst expandiert, wenn es nicht Teil einer Liste ist.

Listing 2 zeigt, wie die Schritte einer möglichen Expansion aussehen könnten, die exakte Expansionsreihenfolge der Makros ist der Implementation überlassen. In der letzten Zeile ist kein Makro-Aufruf mehr vorhanden – dieser Code wird letztendlich ausgeführt.

```
(+ (postfixed (1 (7 5 +) +)) (postfixed (4 6 +)))
(+ (+ (postfixed 1) (postfixed (7 5 +))) (postfixed (4 6 +))))
(+ (+ 1 (postfixed (7 5 +))) (postfixed (4 6 +))))
(+ (+ 1 (+ (postfixed 7) (postfixed 5))) (postfixed (4 6 +))))
(+ (+ 1 (+ 7 (postfixed 5))) (postfixed (4 6 +))))
(+ (+ 1 (+ 7 5)) (postfixed (4 6 +))))
(+ (+ 1 (+ 7 5)) (+ (postfixed 4) (postfixed 6))))
(+ (+ 1 (+ 7 5)) (+ 4 (postfixed 6)))
(+ (+ 1 (+ 7 5)) (+ 4 6))
```

Listing 2: Eine mögliche Expansion des `postfixed`-Makros

2.1 Hygienische Makros

Angenommen man möchte ein `swap!`-Makro implementieren, dass zwei Werte vertauscht. Dieses Makro muss einen dritten, temporären Zwischenspeicher einführen, wie Listing 3 demonstriert.

```
(swap! x y)
;; expandiert zu
(let ((VALUE x))
  (set! x y)
  (set! y VALUE))
```

Listing 3: Ansatz, ein `swap!`-Makro zu implementieren

Ein solches Makro funktioniert jedoch nur solange nicht versucht wird `(swap! VALUE y)` oder `(swap! x VALUE)` zu evaluieren – in diesem Fall würden sich die Werte überschreiben. Dieses Verhalten ist in der Lisp-Welt als *capture* bekannt. Eine Möglichkeit ist es, eine spezielle Funktion namens `gensym` zu verwenden, die sicherstellt, dass ein

neuer Identifier generiert wird, von dem die Implementation garantiert, dass er nicht im Programm benutzt wird[1]. Listing 4 zeigt, wie `gensym` verwendet werden könnte, um das Makro gegen den Fall abzusichern, dass sich Variablen überdecken.

```
(let ((value-name (gensym)))
  (let ((value-name x))
    (set! x y)
    (set! y value-name)))
```

Listing 4: Verbesserter Ansatz eines `swap!`-Makros

Jedoch ist auch dieses Makro noch nicht fehlerfrei; für den Fall dass jemand `set!` im umgebenden Namensraum umdefiniert schlägt es fehl, wie Listing 5 demonstriert. Statt die Werte zu vertauschen, zeigt das `swap!`-Makro die Werte nur an.

```
(let ((set! display))
  (swap! x y))
;; expandiert zu
(let ((set! display))
  (let ((value-name (gensym)))
    (let ((value-name x))
      (set! x y)
      (set! y value-name))))
;; set! substituieren
(let ((value-name (gensym)))
  (let ((value-name x))
    (display x y)
    (display y value-name))))
```

Listing 5: Der verbesserte `swap!`-Ansatz ist immer noch fehlerhaft

Diese Art von Makrosystem ist als „unhygienisches“ Makrosystem bekannt und ist in einigen Lisp-Dialekten wie Common Lisp oder Clojure in Verwendung. Scheme führt hingegen *hygienische Makros* ein, bei denen die genutzten Identifier sich auf den Namensraum des Makros zur *Definitionszeit* und nicht zur *Laufzeit* beziehen. Somit ist das in Listing 6 dargestellte Makro sicher gegen die Eingabe von `VALUE` sowie dem Neubinden von `set!`.

[1] R[6]RS-Scheme standardisiert `gensym` nicht, jedoch bieten viele Implementationen diese Funktionalität an

```
(define-syntax swap!
  (syntax-rules ()
    ((_ x y) (let ((VALUE x))
               (set! x y)
               (set! y VALUE)))))
```

Listing 6: Ein hygienisches `swap!`-Makro

Um hygienische Makros zu implementieren, wurde in Scheme das Konzept der Syntax-Objekte eingeführt. Diese Objekte sind der Listenstruktur von Lisp-Programmen sehr ähnlich, enthalten darüber hinaus aber Metainformationen über den Code, wie etwa die Position an der der Code in der Datei steht, sowie der lexikalische Scope der Identifier. Der genaue Aufbau von Syntax-Objekten ist implementationsspezifisch. Jedoch ist es möglich Scheme-Code mittels `datum->syntax` in Syntax-Objekte zu konvertieren. Die Gegenrichtung von Syntax-Objekt zu Scheme-Code ist über `syntax->datum` ebenfalls möglich, wobei die Metainformationen verworfen werden.

Makros sind somit eigentlich nur Syntax-Transformer. Die Implementation konvertiert den zu transformierenden Code implizit in ein Syntax-Objekt mit entsprechenden Metainformationen über den lexikalischen Scope. Dieses Objekt wird dem Makro als Parameter übergeben und nach den Regeln, die im Makro definiert worden sind, modifiziert. Heraus kommt wiederum ein Syntax-Objekt, das entweder wieder in Scheme-Code überführt und ausgeführt wird, oder weitere Makroexpansionen durchläuft.

```
(define-syntax capture-swap!
  (lambda (stx)
    (syntax-case stx ()
      ((capture-swap! x y)
       #`(let-syntax ((set! (syntax-rules ()
                              ((_ name val)
                               (#,(datum->syntax #'capture-swap! 'set!)
                                name val)))))
           (let ((value x))
             (set! x y)
             (set! y value)))))))
```

Listing 7: Eine „unhygienische“ Variante des `swap!`-Makros aus Listing 6

Man kann dieses Vorgehen gut am `capture-swap!`-Makro in Listing 7 nachvollziehen: Es wird mittels `define-syntax` ein neues Makro definiert, das von einer anonymen Funktion `lambda` mit einem Argument `stx` abgearbeitet wird. Die Funktion gibt einen von `syntax-case` erzeugten Syntax-Transformer zurück, der die Syntax-Objekte nach den Regeln des Makros verarbeitet und zurückgibt.

Das `capture-swap!`-Makro entspricht von der Funktionsweise dem in Listing 6 vorgestellten `swap!`-Makro. Es ist jedoch nicht hygienisch und übernimmt die Definition von `set!` aus dem lokalen Scope, so dass ein lokal definiertes `set!` sich auch auf das Makro auswirkt. Es fällt auf, dass statt `syntax-rules` in diesem Makro `syntax-case` genutzt wird. Beide Ausdrücke generieren aus den ihnen übergebenen Satz von Mustern und Regeln Syntax-Transformer, die wiederum auf der implementationsspezifischen Repräsentation der Syntax-Objekte operieren. Dadurch muss der Makro-Autor sich nicht mit den Interna der jeweiligen Implementation beschäftigen. Der Unterschied zwischen `syntax-rules` und `syntax-case` besteht darin, dass `syntax-rules` keinen Zugriff auf die Syntax-Objekte bietet, wohingegen jede Regel in `syntax-case` ein Syntax-Objekt zurückgeben muss.

In Listing 7 wird mit `#`` eine Spezialsyntax genutzt, die statt Listenliteralen direkt Syntax-Objekte konstruiert. In dieser Syntax wird mit `#,` eine Art Escapesequenz eingesetzt, mit der man den Aufruf von `datum->syntax` einbinden kann. Dieser Aufruf ermöglicht das Einbinden von `set!` aus der *run*-Phase. Dieses „externe“ `set!` wird im in Zeile 5 neu definierten, lokalen `set!`-Makro verwendet um dem restlichen Makro ab Zeile 9 ein nicht-hygienisches `set!` bereitzustellen.

Der Aufruf des Makros bezeugt, dass das Makro tatsächlich unhygienisch ist. Der erste Makro-Aufruf in Listing 8 führt wie erwartet zum Vertauschen der Werte von `a` und `b` da `set!` sich auf das von Scheme bereitgestellte `set!` bezieht. Der zweite Aufruf von `capture-swap!` findet in einer Umgebung statt, in der `set!` durch `+` überschrieben ist. In diesem Fall werden die Werte nicht vertauscht, stattdessen bleiben sie gleich und das Makro gibt die Summe von `a` und `b`, also 65, zurück.

```
(define a 42)
(define b 23)
(capture-swap! a b)
(let ((set! +))
  (display (capture-swap! a b)))
```

Listing 8: Aufruf des `capture-swap!`-Makros

Zusammenfassend kann man sagen, dass das hygienische Makrosystem von Scheme durchdacht ist. Es sichert den Makro-Autor gegen unabsichtliches *capturing*, ermöglicht ihm aber dennoch auf verlangen Werte aus dem lokalen Scope zu lesen. Die Verwendung der Template-Syntax in den Makros ist zwar anfangs ungewohnt, erweist sich aber für viele Makros als kurz und elegant.

2.2 Phasenseparation

Um Scheme-Code auszuführen, müssen erst die Makros ausgewertet werden. Dabei gibt es verschiedene Ansätze, *wann* man Makros expandiert und *wie* man den Makros die

benutzten Identifier bekannt macht. Der R^{6}RS-Standard spezifiziert mehrere Phasen mit jeweils eigenen, getrennten Namensräumen, die der Scheme-Code durchläuft, bietet jedoch genug Implementationsfreiraum, so dass mehrere Ansätze entstanden sind, wie die Identifier an Phasen gebunden werden. Der folgende Abschnitt führt zunächst in die Phasenseparation ein und beleuchtet danach die konkurrierenden Ansätze die sicherstellen, dass jede Phase Zugriff auf die benötigten Identifier hat.

2.2.1 Expand- und Runtime

Es ist in Scheme generell so, dass Makros und Funktionen zu verschiedenen Zeiten definiert und evaluiert werden. Daher kann es sein, dass ein Makro auf eine Funktion zuzugreifen versucht, die aber dem System noch nicht bekannt ist, da Makrodefinitionen vor Funktionsdefinitionen kommen können. Die Phase in der Makros expandiert werden, wird *expand* genannt. Der Code wird hingegen in einer Phase namens *run* ausgeführt. Der einfachste Ausführungsmodus ist es, erst in die *expand*-Phase zu gehen, alle Makros des Programmes zu expandieren, danach in die *run*-Phase zu wechseln und in dieser den Code auszuführen.

Komplexer wird es wenn das Scheme-System eine REPL (Read-Eval-Print-Loop) bereitstellen soll. Scheme-Implementationen haben oftmals einen interaktiven Modus. Jeder vom Programmierer neu eingegebene Code muss auch die *expand*-Phase durchlaufen, um eventuelle Makros zu expandieren, bevor er ausgeführt werden kann.

1. *Batch-Compiler*: Der Code wird komplett durchlaufen, wobei alle Makro- und Funktionsdefinitionen gelesen werden. Die Makros werden in einer *expand*-Phase expandiert. Die Funktionen werden zur Laufzeit nur noch aufgerufen.
2. *Inkrementeller Compiler*: Der Code wird wie bei dem Batch-Compiler gelesen und die *expand*-Phase durchlaufen, bevor es zur *run*-Phase kommt. Jedoch ist es immer wieder möglich die *expand*-Phase zu durchlaufen, etwa wenn neue Funktionen in der REPL definiert werden – diese müssen erst eine *expand*-Phase durchlaufen, um eventuell vorkommende Makros zu definieren oder sie im eingegebenen Code zu evaluieren.
3. *Interpreter*: Es gibt keine Unterscheidung zwischen den Phasen, Makros werden zur Laufzeit expandiert, wenn die Programmausführung an den Aufruf eines Makros ankommt.

Ein wichtiger Unterschied zwischen den einzelnen Phasen ist, welche Identifier in der jeweiligen Phase definiert sind. Man kann aus der *expand*-Phase nicht auf Identifier zugreifen, die erst in der *run*-Phase gültig werden. Trotzdem muss in bestimmen Fällen, wie in Listing 10 gezeigt, eine Funktion in der *expand*-Phase verfügbar sein, diese lässt sich mittels speziellen Syntaxkonstrukten einbinden.

Ein Problem ist an dieser Stelle, dass der R^{6}RS-Standard es offen lässt, ob der Autor angeben muss, in welcher Phase er den Identifier nutzen will oder ob er diese Deklaration

weglassen kann. Der erstere Ansatz ist als *explicit phasing* bekannt, der letztere wird *implicit phasing* bezeichnet. Bei *explicit phasing* wird der Identifier in die vom Autor angegebene Phase importiert, wodurch er nun für allen Code, der in dieser Phase abläuft, sichtbar wird. Falls keine Phase beim Import angegeben ist, wird die *run*-Phase angenommen. Der Ansatz über *implicit phasing* fordert von der Scheme-Implementation, dass sie die notwendigen Phasendefinitionen aus der Nutzung der zu importierenden Identifier ableitet – wenn ein Identifier nur in der *expand*-Phase verwendet wird, dann wird dies von der Implementation erkannt.

Daraus folgt, dass Programme die explizites Phasing verwenden, in Implementationen mit *implicit phasing* funktionieren, nicht aber umgekehrt. Ein Beispiel dafür ist die `distinct?`-Funktion aus der `distinct`-Bibliothek in Listing 9. Diese Funktion überprüft, ob die Elemente einer Liste paarweise unterschiedlich sind.

```
(library (distinct)
         (export distinct?)
         (import (rnrs))

(define (distinct? eq? items)
  (if (null? items) #t
      (let* ((first (car items))
             (rest (cdr items)))
        (cond
          ((or (exists (lambda (element) (eq? first element)) rest)
               (not (distinct? eq? rest))) #f)
          (else #t))))))
```

Listing 9: Die `distinct`-Bibliothek

Mit `distinct?` lässt sich ein neues Makro definieren, `assert-distinct`, welches zusichert, dass alle übergebenen Identifier paarweise verschieden sind. Ein Aufruf von `(assert-distinct a b)` ist gültig, wohingegen `(assert-distinct a a)` nicht gültig ist, da beide Identifier gleich sind. Für die Makrodefinition in Listing 10 wird die `distinct`-Bibliothek eingebunden. Mit dieser wird in der Makrodefinition geprüft, ob die Identifier unterschiedlich sind, andernfalls wird ein Syntaxfehler signalisiert. Dieses Makro bietet den Vorteil, dass der Nutzer des Makros schon zur Compilezeit mitgeteilt bekommt, dass es ein Syntaxfehler ist, denselben Identifier mehrmals zu verwenden.

implicit phasing	*explicit phasing*
Ikarus	PLT
Ypsilon	Larceny
Mosh	
Chez	

Tabelle 1: Übersicht der R^{6}RS-Implementationen nach unterstützter Phasing-Variante

```
(import (rnrs)
        (distinct))

(define-syntax assert-distinct
  (lambda (stx)
    (syntax-case stx ()
      ((_ args ...) (distinct? bound-identifier=? #'(args ...))
                    #'(args ...))
      ((_ args ...) (syntax-violation 'assert-distinct "Duplicate name"
                                      #'(args ...))))))

(let ((a display) (b 42))
  (assert-distinct a b)
  (assert-distinct a a))
```

Listing 10: Das `assert-distinct`-Makro

Dieses Makro funktioniert in Implementationen mit *implicit phasing* (Tabelle 1 listet entsprechende Implementationen auf). Eine R^{6}RS-Implementation mit *explicit phasing* signalisiert den Fehler, dass `distinct?` nicht bekannt ist. Dies liegt daran, dass der Import von `distinct` nur in der *run*-Phase gültig ist, da diese Phase implizit angenommen wird. Damit `distinct?` gefunden werden kann, muss man der Implementation mitteilen, dass der Import von `distinct` für die *expand*-Phase gilt. Listing 11 zeigt eine Variante, die in allen R^{6}RS-Implementationen lauffähig ist.

```
(import (rnrs)
        (for (distinct) expand)) ; neuer Code, gibt die Phase explizit an
        ;;(distinct)) ; alter Code, funktioniert nur mit implicit phasing
```

Listing 11: Eine portablere Variante der Import-Instruktion aus Listing 10

2.2.2 Metalevel

Es gibt Fälle, in denen man möchte, dass ein Makro wiederum zu einem Makro expandiert. Ein Beispiel hierfür ist das `static-map`-Makro aus Listing 12. Das Makro expandiert zu einem weiteren, nun anonymen Makro, welches die dem `static-map`-Makro übergebenen Schlüssel-Werte-Paare zurückgibt. Damit kann ein Dictionary (Hashmap, assoziatives Array) zur Compilezeit definiert werden.

```
1  (library (static-map)
2           (export static-map)
3           (import (rnrs)
4                   (for (rnrs) (meta -1)))
5
6  (define-syntax static-map
7    (syntax-rules ()
8      ((_ (name value) ...)
9       (syntax-rules (<names> name ...)
10        ((_ <names>) '(name ...))
11        ((_ name) value) ...)))))
```

Listing 12: Makro, dass zu einem Makro expandiert

Wenn man versucht dieses Makro naiv zu implementieren, melden Implementationen mit *explicit phasing* einen Fehler, der besagt, dass `quote` nicht bekannt sei. In Scheme wird `quote` verwendet, um einen Wert zu escapen – in diesem Fall `'(name ...)`. Um dem Programm `quote` bekannt zu machen muss es in die entsprechende Phase importiert werden. Ein Versuch es in die *expand*-Phase zu importieren scheitert, da sich das innere, anonyme Makro nicht in der *expand*-Phase befindet, sondern in einer Phase, die Meta-Level -1 bezeichnet wird. Tatsächlich ist es so, dass *expand* und *run* nur Aliase für die Meta-Level 1 und 0 sind, daher muss bei verschachtelten Makros darauf geachtet werden, die Identifier in die richtigen Meta-Level zu importieren.

Meta-Level sind die Bezeichner für die zusätzlichen Phasen, die durchlaufen werden müssen, um Scheme-Programme mit verschachtelten Makros auszuführen. Wie die bereits bekannten *run* und *expand*-Phasen verfügen sie über einen eigenen Namensraum sowie einen definierten Zeitpunkt, wann sie durchlaufen werden.

Die Verwendung des in Listing 12 definierten Makros ist trotz der Meta-Level nicht kompliziert, solange bedacht wird dass `static-map` ein anonymes Makro zurückgibt. Im Listing 13 wird ein Makro definiert, dass den Namen einer Farbe auf den Anfangsbuchstaben abbildet.

```
(import (rnrs)
        (for (static-map) expand))

(define-syntax color
  (static-map (red #\R) (green #\G) (yellow #\Y)))

(display "Available colors: ")
(display (color <names>))
(display (list (color red) (color green) (color yellow)))
(newline)
```

Listing 13: Das anonyme Makro aus `static-map` wird `color` genannt

Wenn ein Makro zu neuen Makros expandiert wird der Meta-Level dekrementiert. Analog dazu gibt es auch Fälle, bei denen der Meta-Level inkrementiert wird: wenn in der Definition des Makros wiederum ein Makro definiert wird. Listing 14 definiert ein Makro `m`, in dessen *Definition* (nicht in dessen *Expansion*, wie in Listing 12) ein weiteres Makro `m2` definiert wird. Die Phase von `m` ist *expand*, die Phase von *m2* ist hingegen die Phase von `m` + 1, also `(meta 2)`. In `m2` wird auf `begin`, `lambda` und `display` zugegriffen, also müssen diese Identifier auch in `(meta 2)` existieren – daher wird in der Import-Deklaration explizit angegeben, dass diese drei Identifier in Meta-Level 2 importiert werden sollen.

```
(import (rnrs)
        (for (only (rnrs) begin lambda display) (meta 2)))

(define-syntax m
  (let ()
    (define-syntax m2
      (begin
        (display "at metalevel 2\n")
        (lambda (x) "expanded-m\n")))
    (define _ (display "at metalevel 1\n"))
    (lambda (x) (m2))))

(display (m))
```

Listing 14: Makros mit Meta-Level ≥ 1

Im Gegensatz dazu benötigen Implementationen mit *implicit phasing* (siehe Tabelle 1) keinerlei zusätzliche `(meta n)`-Definitionen, sondern inferieren den benötigten Meta-Level.

3 Zusammenfassung

Das Makrosystem, welches in R^{6}RS-Scheme standardisiert wurde, bietet mit `syntax-rules` sowohl eine Möglichkeit einfache, sowie mit `syntax-case` komplexere Makros zu bauen. Beide Systeme nehmen dem Makro-Autor im Vergleich zu anderen Sprachen viel Arbeit ab, insbesondere bei Verwendung einer Implementation mit *implicit phasing*. Es bleibt zu hoffen, dass sich dieses System in Zukunft durchsetzt und zukünftige Sprachstandards dieses Verfahren favorisieren werden.

Ob so mächtige Makros in einer Programmiersprache sinnvoll sind, kann nicht abschließend geklärt werden. Makros in Scheme bieten die Möglichkeit, elegant Abstraktionen etwa für die parallele Programmierung wie Aktoren oder STM in die Sprache zu integrieren ohne an der Scheme-Implementation selbst editieren zu müssen. Jedoch kann übermäßiger Einsatz von Makros zu schwer zu verständlichem Code und einer Zersplitterung der Sprache in verschiedene Dialekte führen. Dies beweist die Existenz von mindestens 14 inkompatiblen, nicht-standardisierten Objektsystemen[2], deren Syntax und Semantik der Programmierer jeweils immer neu lernen muss.

Literatur

[1] Community Scheme Wiki. hygiene-versus-gensym. `http://community.schemewiki.org/?p=hygiene-versus-gensym&c=hv&t=1270440933`.

[2] Community Scheme Wiki. object-systems. `http://community.schemewiki.org/?p=object-systems&c=hv&t=1270869643`.

[3] R. Kent Dybvig. *The Scheme Programming Language, 4th Edition*. MIT Press, 2009.

[4] Matthew Flatt. Composable and compilable macros: You want it when? In *Proceedings of the ICFP*. International Conference on Functional Programming, October 2002.

[5] Abdulaziz Ghuloum and R. Kent Dybvig. Implicit phasing for r6rs libraries. In *Proceedings of the ICFP*, pages 303–313. International Conference on Functional Programming, October 2007.

[6] Michele Simionato. The Adventures of a Pythonista in Schemeland. `http://www.phyast.pitt.edu/~micheles/scheme/TheAdventuresofaPythonistainSchemeland.pdf`.

[7] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten, Robby Findler, and Jacob Matthews. Revised^6 Report on the Algorithmic Language Scheme. *J. Funct. Program.*, 19(S1):1–301, 2009.

[2] TinyCLOS, Swindle, Gauche, GOOPS, STklos, MIT-Scheme-SOS, Meroon, YASOS, Protobj, Prometheus, TinyTalk, OakLisp, ClosureTalk, LispMeObjects