Macro-by-Example:
Deriving Syntactic Transformations from their Specifications

By

Eugene E. Kohlbecker
Indiana University
Bloomington, IN 47405

and

Mitchell Wand
Northeastern University
Boston, Massachusetts 02115

TECHNICAL REPORT NO. 206

Macro-by-Example:
Deriving Syntactic Transformations from their Specifications

by

Eugene E. Kohlbecker, Indiana University
and Mitchell Wand, Northeastern University

November, 1986

This material is based on work supported in part an IBM Fellowship Award to Eugene E. Kohlbecker, and by the National Science Foundation grant numbers MCS83-03325, MSC83-04567, DCR85-01277, and DCR86-05218.

Macro-by-Example:
Deriving Syntactic Transformations from their Specifications

Eugene E. Kohlbecker, Indiana University
Mitchell Wand, Northeastern University

Abstract

This paper presents two new developments. First, it describes a "macro-by-example" specification language for syntactic abstractions in Lisp and related languages. This specification language allows a more declarative specification of macros than conventional macro facilities do by giving a better treatment of iteration and mapping constructs. Second, it gives a formal semantics for the language and a derivation of a compiler from the semantics. This derivation is a practical application of semantics-directed compiler development methodology.

1. Introduction

Modern programming languages offer powerful facilities for procedural and data abstraction which encapsulate commonly-used patterns of procedure invocation and of data usage, respectively. Often, however, one encounters typical patterns of linguistic usage that do not fall neatly into either category. An example is the `let` expression, which encapsulates an often-used pattern of function creation and call:

```
(let ((i e) ...) b ...)
  ==> ((lambda (i ...) b ...) e ...)
```

This is not a procedural abstraction; rather, it is a *syntactic* abstraction: an abstraction of a typical pattern of syntax.

Most modern languages do not provide tools for the creation of syntactic abstractions comparable to those available for procedural or data abstractions. Even in languages such as Lisp that allow syntactic abstractions, the process of defining them is notoriously difficult and error-prone. To define `let` as a macro, we must write a procedure that transforms any expression which matches the left-hand pattern into a corresponding instance of the right-hand pattern. The code for this looks like:

```
(lambda (s)
  (cons
   (cons 'lambda
         (cons (map car (cadr s))
               (cddr s)))
   (map cadr (cadr s))))
```

This code can hardly be considered transparent. It bears no obvious relation to the transformation it engenders. It is difficult to see the shape of the structure it is building or the shape of the structure it is trying to take apart. Furthermore, it does no error checking on its input.

Modern Lisps supply some tools to help the macro-writer, most notably backquote and `defmacro` (*e.g.*, [Foderaro, Sklower, & Layer 83]). Backquote makes the code for building the output look more like the output itself, and `defmacro` includes a pattern-matching facility. Using these tools, the `let` macro might be defined as:

```
(defmacro let (decls . body)
  '((lambda ,(map car decls) . ,body)
    . ,(map cadr decls)))
```

This is considerably better, but the mapping functions are still mysterious. Furthermore, the backquote mechanism is itself error-prone: one *always* leaves out at least one comma on the first try!

This Material is based on work supported by the National Science Foundation under grant numbers MCS 8303325, MCS 8304567, and DCR 8605218. Eugene Kohlbecker was supported by an IBM Graduate Fellowship while part of this work was being developed.

Authors' addresses: Eugene E. Kohlbecker, Computer Science Department, Indiana University, Lindley Hall 101, Bloomington, IN 47405. Mitchell Wand, College of Computer Science, Northeastern University, 360 Huntington Avenue #161CN, Boston, MA 02115.

To appear in the 1987 ACM Symposium on Principles of Programming Languages.

In our facility, one defines `let` as follows:

```
(declare-syntax let
  [(let ((i e) ...) b ...)
   ((lambda (i ...) b ...) e ...)])
```

This is close to the language used for specifying syntactic extensions in the revised Scheme report [Steele and Sussman 78] and the 1986 Scheme report [Rees, Clinger, *et al.* 86], except that it is executable. The specification language has the following features:

1. Pattern-matching, including error-checking, is done on the input.
2. The output specification matches the form of the output. No commas or other symbols are needed.
3. Repetitive elements are specified naturally on both input and output.

This specification mechanism has been used in various versions of Scheme since 1982, and has proved to be a robust and highly useful feature. Only recently, however, did the need for formal documentation of the mechanism lead us to develop the formal semantics and the semantically-derived compiler presented here.

We call this mechanism macro-by-example, or MBE. It has allowed us to embed a number of interesting languages in Scheme, such as:

- a type-checked Scheme variant called SPS [Wand 84]
- a coroutine mechanism [Friedman, Haynes, & Wand 86]
- an import/export mechanism for modules [Felleisen and Friedman 86]
- two quite different semantically-derived subsets of Prolog [Felleisen 85, Wand 85]

The presence of this convenient syntactic extension tool provides an important design dimension: given any proposed language extension, we get to decide how much of it should be treated as procedural abstraction, how much as data abstraction, and how much as syntactic abstraction. The presence of a good syntactic abstraction mechanism means that a language extension which might be highly cumbersome expressed in a purely procedural way can be made far more usable by a propitious choice of syntax. (Indeed, the `declare-syntax` facility itself may be viewed as such a propitious choice of syntax). Conversely, given a complex syntactic abstraction, we can try to convert some of it to a procedural or a data abstraction.

In Section 2, we give a brief description of the specification language itself and additional examples. In Section 3, we present a formal semantics of the specification language. This semantics could, if desired, be used to create an interpreter for this language of transformations. In Sections 4 and 5, we show how the semantics can be transformed into a compiler which takes a transformation specification and produces Scheme code for performing the transformation (including error checking and mapping functions).

2. The Specification Language

Consider the following example of MBE:

```
(declare-syntax and
  [(and) true]
  [(and e) e]
  [(and e1 e2 ...) (if e1 (and e2 ...) false)])
```

It illustrates some of the important features of MBE:

1. An MBE specification consists of a series of input-output patterns, which are tried in order. If a call matches none of the input patterns, then an error is signalled.
2. Backquotes and commas in the output patterns are avoided by a simple convention: identifiers which appear in the input pattern are treated as pattern variables; all other identifiers are constants. No special treatment is necessary for `lambda` or `quote` in the transcription proper, but the result of the transcription can be processed using the α-converting expander of [Kohlbecker, *et al.* 86] to avoid capture of bound variables in macros such as:

   ```
   (declare-syntax or2
     [(or2 x y) (let ((v x)) (if v v y))])
   ```

   which would otherwise capture any occurrences of `v` in its second argument. (The production version of MBE also allows the user to specify the capture of particular variables).
3. Whenever the ellipsis (indicated by the atom "`...`") is used, it must be preceded by a pattern, and it must be the last element of the pattern list or sublist. (Again, the production version is somewhat more generous; we do not illustrate this here). Ellipses may be nested, so pattern variables may denote S-expressions, lists of S-expressions, lists of lists of S-expressions, *etc.*

As an illustration of MBE as a language-embedding tool, consider the following macros, which are taken from [Wand 85]. These macros define two new special forms whose translation produces (through details that are of no concern here) an interface to an underlying Prolog semantics.

```
(declare-syntax clauses
  ((clauses (var1 ...) cl1 ...)
   (lambda (z)
     (let ((var1 (genvar)) ...)
        (call/kf
          (lambda (kf)
             (ff-set-ref! **cut-point** kf)
             (choose (clause cl1) ...)))))))
```

```
(declare-syntax clause
   ((clause (pat act1 ...))
    (begin
       (unify (pattern pat) z)
       (action act1) ...)))

(declare-syntax action
   ((action (fn e1 ...))
    (fn (list (pattern e1) ...))))
```

Without the ability to adapt the syntax, the interface would have been impossible to use (indeed, an earlier version of our Prolog semantics foundered on precisely this point). With these macros in place, we can write Prolog predicates as Scheme procedures in a moderately convenient syntax, for example:

```
(define Append
   (clauses (y a d u v)
      ((nil y y))
      (((a . d) u (a . v))
       (Append d u v))))
```

The production version [Kohlbecker 86] includes a number of additional bells and whistles. First, arbitrary tests may be specified by an optional argument called a *fender*. For example, the macro for `let` should include a test to make sure the bound variables are atoms:

```
(declare-syntax let
  [(let ((i e) ...) b ...)
   (mapand atom? '(i ...))
   ((lambda (i ...) b ...) e ...)])
```

Second, arbitrary processing of the macro call may be performed by a `with`-specification, which binds its variable to the result of a Scheme calculation.

```
(declare-syntax new-lang
  [(new-lang e ...)
   (with ([code (translate-to-scheme '(e ...))])
       (top-level code))])
```

And third, the macro-writer may also specify *keywords* to serve as pattern constants in the input patterns (*e.g.,* `else`) and as terminators for ellipses.

3. Semantics

In this section, we sketch the formal semantics of the MBE mechanism. We restrict ourselves to a single input-output pair, emphasizing the correct treatment of ellipses.

In our description, we employ relatively standard notation. We use three closely-spaced dots (...) for the ellipsis symbol and center dots ($\cdots$) for ellipsis in the meta-language. We use $(x)_1$, $(x)_2$ for projection, $(\alpha \Rightarrow \beta, \gamma)$ for conditional, and $A \mathrel{-\!\!\circ\!\!\rightarrow} B$ for the set of partial functions from A to B.

We define the following domains:

$$
\begin{aligned}
\textit{S-exp} &::= \textit{Ident} \mid (\textit{S-exp} \cdots \textit{S-exp}) \\
S^\sharp &::= \textit{S-exp} \mid (S^\sharp \cdots S^\sharp) \\
\textit{Pat} &::= () \mid \textit{Ident} \mid (\textit{Pat} \;.\; \textit{Pat}) \mid (\textit{Pat} \;...) \\
\textit{Env} &= \textit{Ident} \mathrel{-\!\!\circ\!\!\rightarrow} (\textit{Int} \times S^\sharp)
\end{aligned}
$$

The function *fv*, defined over *Pat* returns a list of identifiers contained in a pattern. The meta-variable a ranges over *Ident*, p over *Pat*, s over *S-exp* and $S^\sharp$, and ρ over *Env*.

The semantics is comprised of three main functions:

$$
\begin{aligned}
\mathcal{B}&: \textit{Pat} \rightarrow \textit{S-exp} \rightarrow \textit{Bool} \\
\mathcal{D}&: \textit{Pat} \rightarrow \textit{S-exp} \mathrel{-\!\!\circ\!\!\rightarrow} \textit{Env} \\
\mathcal{T}&: \textit{Pat} \rightarrow \textit{Env} \mathrel{-\!\!\circ\!\!\rightarrow} \textit{S-exp}
\end{aligned}
$$

$\mathcal{B}$ takes an input pattern and an S-expression; it returns a boolean indicating whether the S-expression matches the pattern. $\mathcal{D}$ takes an input pattern and an S-expression and (if it matches the pattern) returns an environment associating pattern variables with their resulting bindings. Each binding is a pair consisting of a non-negative integer and an element of $S^\sharp$. The integer indicates the variable's level, the number of ellipses enclosing it. The element of $S^\sharp$ is of the corresponding type: if the level is 0, it is an S-expression; if the level is 1, it is a list of S-expressions; if the level is 2, a list of lists of S-expressions, *etc.*

$\mathcal{T}$ takes an output pattern and an environment and expands the pattern in that environment. Thus a transform function with type *S-exp* $\mathrel{-\!\!\circ\!\!\rightarrow}$ *S-exp* for one input-output pattern pair (*lhs rhs*) may be described as:

$$
\mathcal{E}\llbracket(\textit{lhs}\ \ \textit{rhs})\rrbracket = \lambda s.(\mathcal{B}\llbracket \textit{lhs}\rrbracket s \Rightarrow \mathcal{T}\llbracket \textit{rhs}\rrbracket(\mathcal{D}\llbracket \textit{lhs}\rrbracket s), \\ \textit{error})
$$

The variable s represents a macro invocation.

We begin with the clauses for $\mathcal{B}$. They are as follows:

$$
\begin{aligned}
\mathcal{B}\llbracket()\rrbracket &= \lambda s.\textit{null?}\, s \\
\mathcal{B}\llbracket a\rrbracket &= \lambda s.\textit{true} \\
\mathcal{B}\llbracket(p_1\ .\ p_2)\rrbracket &= \lambda s.\textit{pair?}\, s \wedge \mathcal{B}\llbracket p_1\rrbracket(\textit{hd}\, s) \wedge \mathcal{B}\llbracket p_2\rrbracket(\textit{tl}\, s) \\
\mathcal{B}\llbracket(p\ ...)\rrbracket &= \lambda s.\textit{list?}\, s \wedge \textit{mapand}\,(\mathcal{B}\llbracket p\rrbracket)\, s
\end{aligned}
$$

The pattern () matches only the empty list. The pattern a is a variable and matches anything. The pattern $(p_1\ .\ p_2)$ matches s if and only if s is a pair (that is, not an atom and not empty), and the corresponding subpatterns match. The pattern $(p\ ...)$ matches s if and only if s is a list and p matches each element of s.

The equations for $\mathcal{D}$ are guaranteed to be sensible only on the condition that $\mathcal{B}\llbracket p\rrbracket s$ is true:

$$
\begin{aligned}
\mathcal{D}\llbracket()\rrbracket &= \lambda s.\emptyset \\
\mathcal{D}\llbracket a\rrbracket &= \lambda s.\{(a \mapsto (1, s))\} \\
\mathcal{D}\llbracket(p_1\ .\ p_2)\rrbracket &= \lambda s.\mathcal{D}\llbracket p_1\rrbracket(\textit{hd}\, s) \cup \mathcal{D}\llbracket p_2\rrbracket(\textit{tl}\, s)
\end{aligned}
$$

Here we assume that there are no variables in common between p_1 and p_2.