Visualizing Evaluation in Applicative Languages

David S. Touretzky and Peter Lee

November 7, 1989
CMU-CS-89-198 2

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

In this article we present a technique for visualizing evaluation in applicative languages that helps to graphically explain a number of basic concepts, including lexical *vs.* dynamic scoping, closures, local and special variables, and macro expansion. Called "evaltrace notation," it appears in a recent textbook by the first author and has been employed in several courses at Carnegie Mellon. Although our discussion focuses primarily on the notation itself, we also provide some insights into the implementation of Lisp and Scheme interpreters and the differences between the lexical and dynamic scoping disciplines. It is our hope that evaltrace notation will be widely adopted by Lisp educators. In support of this, we have made available a set of LaTeX macros to allow others to produce evaltrace diagrams similar to the ones that appear here.

This research was supported in part by the Office of Naval Research under contract N00014-84-K-0415 and in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 5404, monitored by the Office of Naval Research under the same contract.

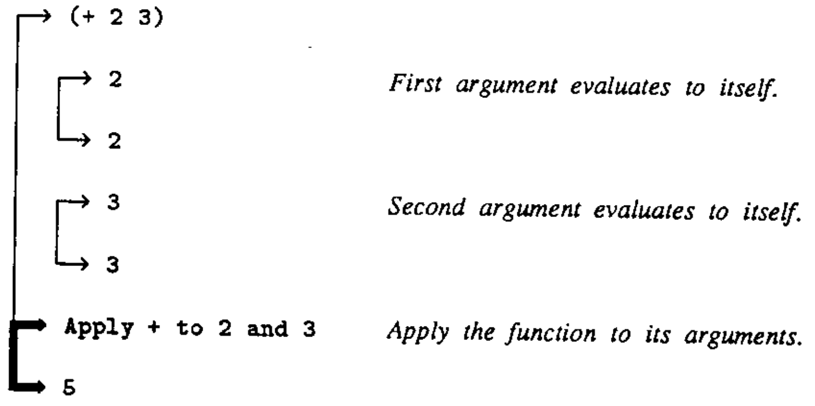
1 Introduction

The EVAL and APPLY operations form the core of a Lisp interpreter and are fundamental to implementations of applicative languages in general. One of the more difficult tasks in teaching Lisp to beginners is getting them to understand the true nature of the EVAL/APPLY duality. With the advent of Scheme [2] and Common Lisp [3], even experienced Lisp programmers may find that they don't fully understand evaluation, especially as it relates to lexical *vs.* dynamic scoping, closures, local and special variables, and macro expansion.

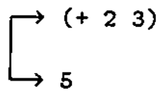
In this article we present a technique for visualizing evaluation in applicative languages that helps to graphically explain each of the above concepts. Called "evaltrace notation," it appears in a recent textbook by the first author [4] and has been employed in several courses at Carnegie Mellon. Although our discussion in this paper focuses primarily on the notation itself, we also provide some insights into the implementation of Lisp and Scheme interpreters and the differences between the lexical and dynamic scoping disciplines. Extensions to other features such as tail-recursion elimination and first-class continuations are quite easy to make, though we do not include them here. It is our hope that evaltrace notation will be widely adopted by Lisp educators. In support of this, we are making available a set of LaTeX macros to allow others to produce evaltrace diagrams similar to the ones that appear here.

2 The Basic Evaltrace Notation

Evaluation in Lisp proceeds according to a set of evaluation rules built into the EVAL and APPLY functions. Evaluation of the simple expression `(+ 2 3)` employs two of these rules. First, numbers evaluate to themselves. Second, to evaluate a list describing a function call, one first evaluates the arguments to the function, and then *applies* the function to the evaluated arguments. This leads to the evaltrace of `(+ 2 3)` shown below. Thin lines represent calls to EVAL, and thick lines calls to APPLY.


(+ 2 3)
2
First argument evaluates to itself.
2
3
Second argument evaluates to itself.
3
Apply + to 2 and 3
Apply the function to its arguments.
5

Often we will want to suppress some details, such as the evaluation of trivial arguments to a function, or the application of a primitive like +. In that case, the above evaluation is depicted this way:



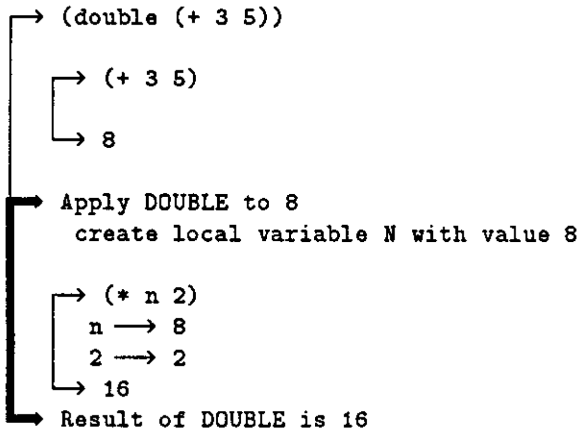
When an evaltrace diagram is elided like this to a single application of EVAL, it is equivalent to the simple evaluation arrow used in [3] and most other books on Lisp:

```
(+ 2 3) ⟶ 5
```

The real power of evaltrace notation becomes apparent when we consider the application of non-primitive functions, since these may create local variables and call other functions from within their bodies. The function DOUBLE below creates a local variable N to hold its argument, and calls the primitive function * to compute its result. (All of the examples in this paper are written in Common Lisp.)

```
(defun double (n)
  (* n 2))
```

An evaltrace of the expression `(double (+ 3 5))` shows how a local variable is created inside the body of DOUBLE to hold the evaluated argument.



```
→ (double (+ 3 5))

   → (+ 3 5)

   → 8

→ Apply DOUBLE to 8
   create local variable N with value 8

   → (* n 2)
     n ⟶ 8
     2 ⟶ 2
   → 16
→ Result of DOUBLE is 16
```

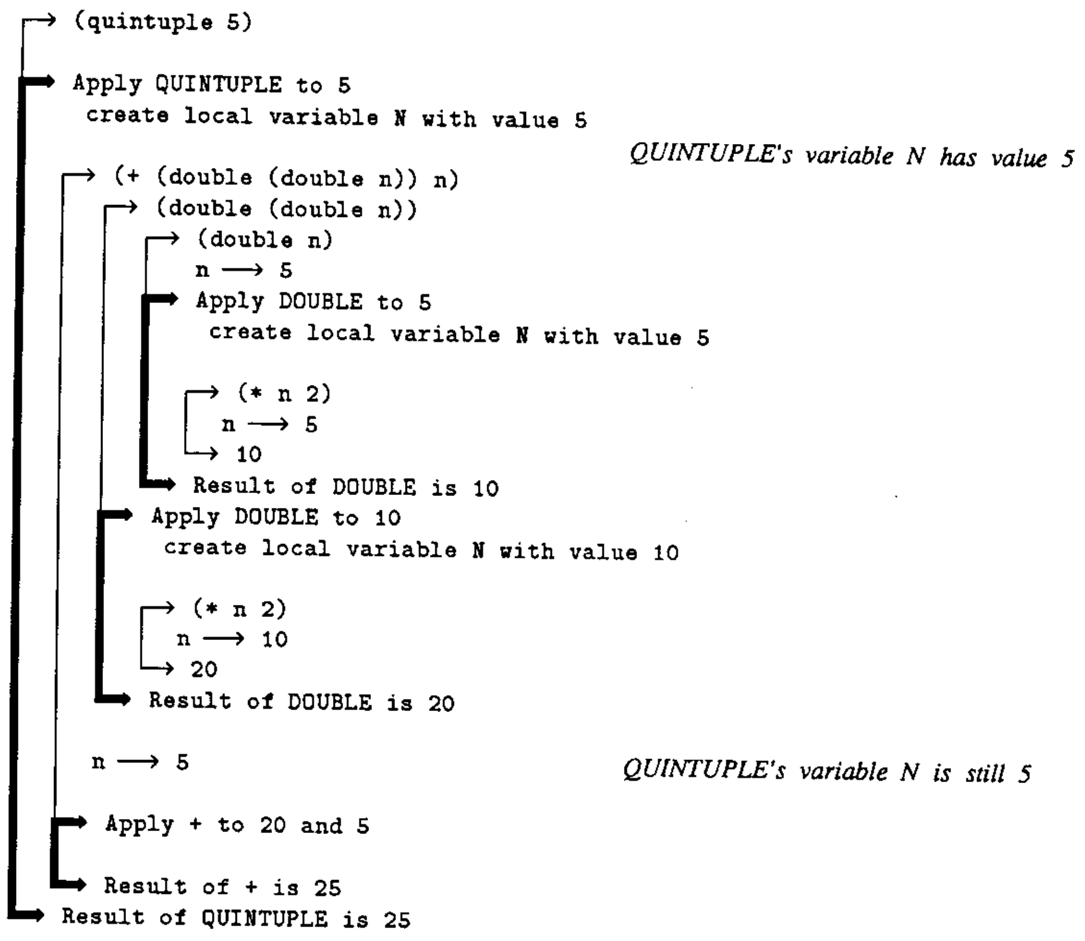
This example involves the evaluation of the symbol N. The evaluation rule for symbols is that they evaluate to the value of the variable which they name. In earlier Lisp dialects people would speak of symbols themselves being variables or having values, but this is not appropriate in Scheme or Common Lisp since several variables with the same name may exist simultaneously in different lexical contexts. The symbol N appearing in the body of DOUBLE refers to the local variable N that is created when DOUBLE is applied. Outside the body of DOUBLE, the symbol N does not refer to this variable. The fact that this is a *textual* property of the program is important, as it means that illegal references to N can easily be caught by a compiler, before calls to DOUBLE are actually evaluated.

When DOUBLE is applied, a ***lexical contour*** is created in which the variable named N resides. In the evaltrace diagram, the thick line marking the application of DOUBLE shows the boundaries of this lexical contour. Only thick lines denote lexical contours. Thin lines, which represent calls to EVAL, do not denote contours. This distinction is an important part of the scoping of Lisp programs, which will be discussed later.

It is easy to forget that lexical contours are created only when a function is applied, not when it is defined. If DOUBLE were to be applied five times, five distinct lexical contours, each with its own variable named N, would be created. Consider the following function:

```
(defun quintuple (n)
  (+ (double (double n)) n))
```

Both DOUBLE and QUINTUPLE, when applied, create local variables named N. Because of lexical scoping, QUINTUPLE's N is distinct from DOUBLE's N. Furthermore, each of the two applications of DOUBLE creates a *new* variable referred to by the name N. This can be seen in the following evaltrace diagram:

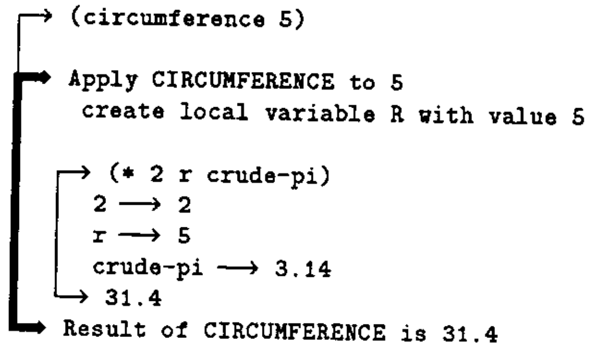


```
→ (quintuple 5)
  Apply QUINTUPLE to 5
   create local variable N with value 5
                                                QUINTUPLE's variable N has value 5
    → (+ (double (double n)) n)
      → (double (double n))
        → (double n)
          n ⟶ 5
          Apply DOUBLE to 5
           create local variable N with value 5

            → (* n 2)
              n ⟶ 5
            → 10
          Result of DOUBLE is 10
        Apply DOUBLE to 10
         create local variable N with value 10

          → (* n 2)
            n ⟶ 10
          → 20
        Result of DOUBLE is 20

      n ⟶ 5                                     QUINTUPLE's variable N is still 5

    Apply + to 20 and 5

    Result of + is 25
  Result of QUINTUPLE is 25
```

Three distinct lexical contours are shown (not counting the one for +), and each contains a variable named N. These variables are independent. Thus, the value of N in QUINTUPLE's contour remains 5, even after the second call to DOUBLE assigns the value 10 to a (distinct) variable which is also named N.

Every lexical contour has a *parent contour*, except that global variables exist in a top-level contour (also called the *global contour*) that has no parent. Consider the function CIRCUMFERENCE that references a global variable CRUDE-PI: (SETF is the general assignment operator in Common Lisp.)

```
(setf crude-pi 3.14)

(defun circumference (r)
  (* 2 r crude-pi))
```



```
→ (circumference 5)
  Apply CIRCUMFERENCE to 5
   create local variable R with value 5

    → (* 2 r crude-pi)
      2 ⟶ 2
      r ⟶ 5
      crude-pi ⟶ 3.14
    → 31.4
  Result of CIRCUMFERENCE is 31.4
```

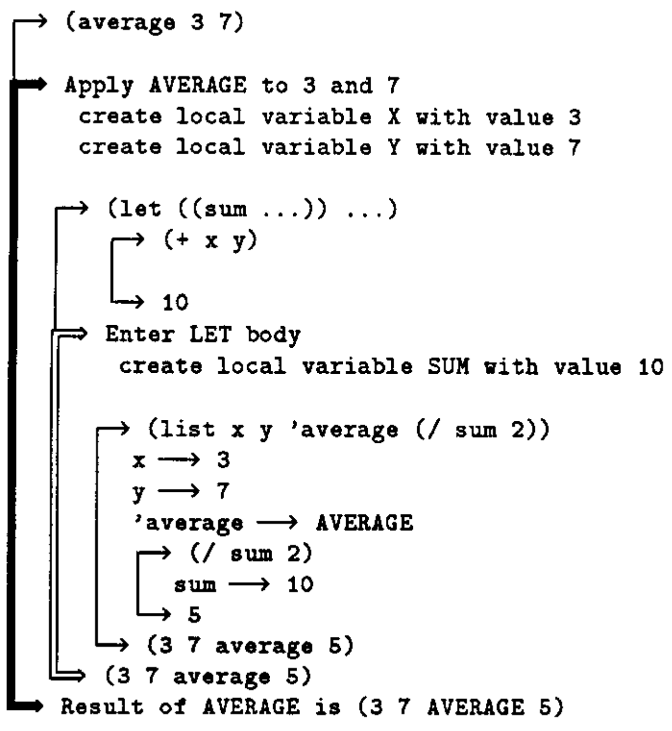
The rules for mapping symbols to the variables which they name are called *scope rules*. When evaluating the symbol R in the body of CIRCUMFERENCE, the first scope rule tells us to look in the current lexical contour for a variable with that name. Since there is a local variable named R created by CIRCUMFERENCE, the symbol R is treated as a reference to that variable. But in the case of CRUDE-PI, there is no local variable by that name in the current contour. Consequently, the rules for lexical scoping tell us to look in the parent contour, which happens to be the global contour (which is also the only other contour in this example). A variable named CRUDE-PI does exist in the global contour, so the symbol CRUDE-PI is taken to refer to that variable.

The difference between lexical and dynamic scoping disciplines is in the way parent contours are determined. We will return to this topic shortly.

3 Nested Lexical Contours

Within the body of a function, additional local variables may be created using the LET special function. Consider the following:

```
(defun average (x y)
  (let ((sum (+ x y)))
    (list x y 'average (/ sum 2))))
```



```
→ (average 3 7)

➡ Apply AVERAGE to 3 and 7
   create local variable X with value 3
   create local variable Y with value 7

   → (let ((sum ...)) ...)
      → (+ x y)

      → 10
   ⇒ Enter LET body
      create local variable SUM with value 10

      → (list x y 'average (/ sum 2))
        x ⟶ 3
        y ⟶ 7
        'average ⟶ AVERAGE
         → (/ sum 2)
           sum ⟶ 10
         → 5
      → (3 7 average 5)
   ⇒ (3 7 average 5)
➡ Result of AVERAGE is (3 7 AVERAGE 5)
```

Notice that the lexical contour associated with the LET body is drawn as a hollow arrow rather than a solid arrow. A solid arrow indicates that the parent contour is the global contour. The hollow arrow in this diagram indicates that the parent contour is the immediately enclosing contour. Thus the parent contour of the LET body is the contour generated by the body of AVERAGE. AVERAGE's parent contour is the global contour. When evaluating the symbol SUM inside the LET body, we find that there is a variable by that name in the current contour. But when evaluating X and Y in the LET body, we find we must go look in the parent contour. Since there are local variables named X and Y in the contour of AVERAGE, the symbols X and Y are taken to refer to those variables. (This example also illustrates another of Lisp's evaluation rules: quoted objects evaluate to themselves, without the quote.)

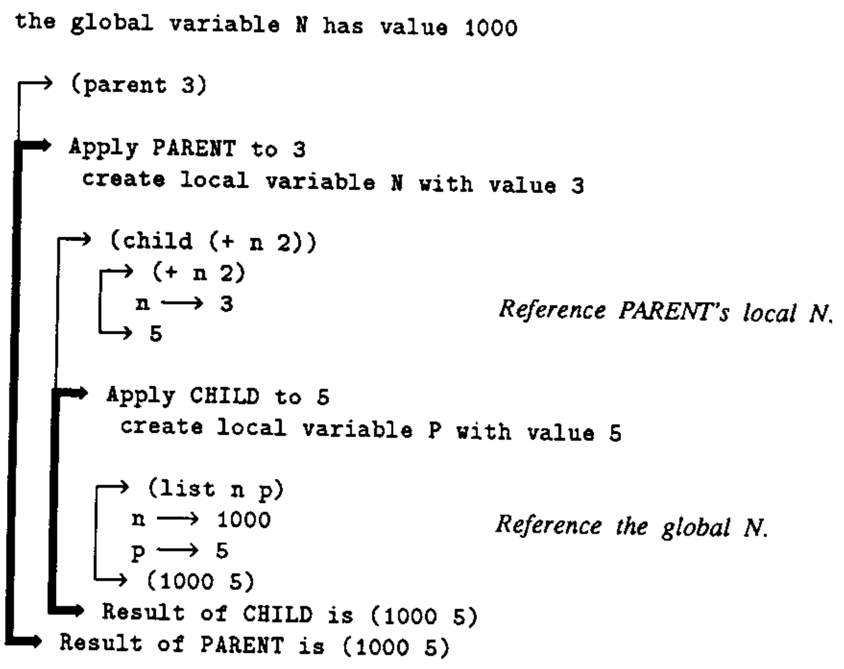
Now we can present an example that highlights the distinction between lexical and dynamic scoping. Notice below that there is both a global variable N with value 1000, and a local variable N in the argument list of PARENT. PARENT calls CHILD, which contains the symbol N in its body. To which variable does CHILD's N refer?

```
(setf n 1000)

(defun parent (n)
  (child (+ n 2)))

(defun child (p)
  (list n p))
```

Under lexical scoping, every function has associated with it, as part of its definition, a parent contour. Functions defined at the top level with DEFUN always have the global contour as their parent. (We'll have more to say later about how contours are associated with functions that are not defined at the top level.) A function's *environment* is the set of objects visible within its contour, in other words, objects that reside in its own contour, or in its parent contour, or in its parent contour's parent contour, and so on. Functions can only access those variables that are visible within their environment. Variables that have the same name "shadow" each other. For instance, a variable N in the current contour would shadow a variable N in the parent contour, so the latter would not be visible.

In evaltrace diagrams, environments are depicted graphically via the chain of nested contours that define them. Most contours drawn with a hollow arrow (such as LET contours) have the immediately surrounding contour as their parent, while contours for functions defined at top level are drawn with a solid arrow, indicating that their parent contour is the global contour.

Getting back to our present example, the lexical scope rules dictate that the symbol N inside the body of CHILD must refer to the global variable N, not to PARENT's local N. This is illustrated by the following evaltrace diagram:

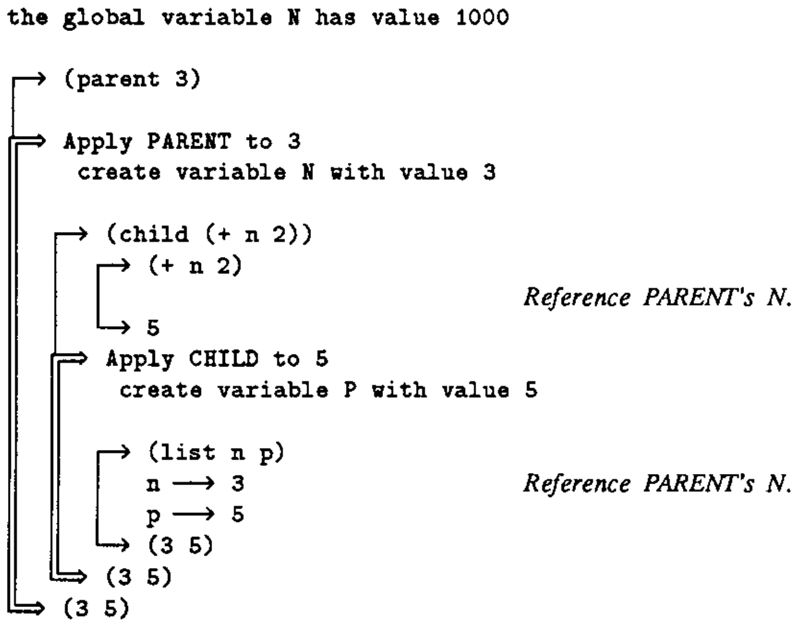


```
the global variable N has value 1000

(parent 3)

Apply PARENT to 3
 create local variable N with value 3

  (child (+ n 2))
    (+ n 2)
    n → 3                          Reference PARENT's local N.
    5

  Apply CHILD to 5
   create local variable P with value 5

    (list n p)
    n → 1000                       Reference the global N.
    p → 5
    (1000 5)
  Result of CHILD is (1000 5)
Result of PARENT is (1000 5)
```

CHILD's environment consists of its own contour (in which the variable P resides), and its parent contour, which is the global contour. Since CHILD doesn't have a local variable named N, the symbol N in its body must refer to the global N, whose value is 1000. PARENT's local variable N is completely invisible to CHILD, since PARENT's contour is not part of CHILD's environment.

This scheme is called *lexical* scoping because environments mirror the nesting structure of the code. Since CHILD isn't defined inside of PARENT, none of PARENT's local variables are visible to it. We'll say more about the relationship between textual nesting and environments in the next section.

Early Lisp dialects, from Lisp 1.5 up through MacLisp and InterLisp—and APL and SNOBOL as well—use dynamic rather than lexical scoping [1]. In dynamic scoping functions do not have parent contours permanently associated with them; instead each contour uses the most recently created contour as its parent. Thus, environments are determined dynamically rather than statically. If the preceding example were tried in a dynamically scoped Lisp, the parent contour of CHILD would be the one created by PARENT, and the result of the evaluation would be different, as shown below:



```
the global variable N has value 1000

(parent 3)

Apply PARENT to 3
 create variable N with value 3

  (child (+ n 2))
    (+ n 2)
                                   Reference PARENT's N.
    5
  Apply CHILD to 5
   create variable P with value 5

    (list n p)
    n ⟶ 3                         Reference PARENT's N.
    p ⟶ 5
    (3 5)
  (3 5)
(3 5)
```

Dynamic scoping was used in early Lisp dialects because in an interpreter it is the most straightforward scoping discipline to implement efficiently. (This turns out not to be the case when compiling. This is one reason that modern dialects of Lisp which are compiled as well as interpreted, such as Common Lisp, are lexically scoped.) In evaltrace diagrams, the nesting of contours accurately reflects the structure of the call stack of an interpreter, with the inner-most contour being the top of the stack and the global contour at the bottom. So, we will often refer to the nesting of contours as the call stack. Dynamic scoping allows the call stack to be used as the environment, which means that the contours in evaltrace diagrams are drawn only with hollow, never solid, arrows.

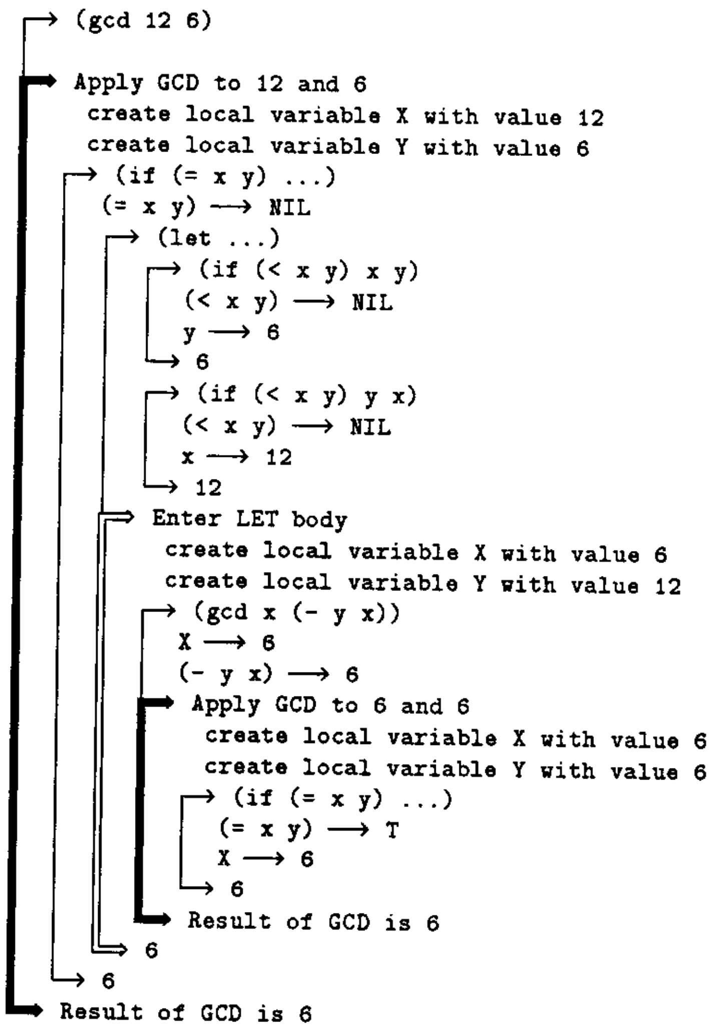
4 Scoping and Textual Inclusion

Earlier we saw that the contours created by the LET special function are drawn with hollow instead of solid arrows. This is because LET creates contours that are local to other contours. In order to explain this further, let's first examine the LET function in more detail.

LET can be used to create any number of variables, with the property that the creation of the variables and assignment of values to them is carried out "in parallel." The following (slightly unusual) definition of a function for computing the greatest common divisor of two integers demonstrates a use of this parallel feature.

```
(defun gcd (x y)
  (if (= x y)
      x
      (let ((x (if (< x y) x y))
            (y (if (< x y) y x)))
        (gcd x (- y x)))))
```

The LET expression in GCD is used to ensure that X is less than Y, swapping their values if necessary, before evaluating the recursive call in the body. Doing the swap correctly depends on the parallel nature of LET, as the following evaltrace shows.



```
→ (gcd 12 6)

→ Apply GCD to 12 and 6
   create local variable X with value 12
   create local variable Y with value 6
  → (if (= x y) ...)
    (= x y) → NIL
    → (let ...)
      → (if (< x y) x y)
        (< x y) → NIL
        y → 6
      → 6
      → (if (< x y) y x)
        (< x y) → NIL
        x → 12
      → 12
    ⇒ Enter LET body
       create local variable X with value 6
       create local variable Y with value 12
      → (gcd x (- y x))
        X → 6
        (- y x) → 6
      → Apply GCD to 6 and 6
         create local variable X with value 6
         create local variable Y with value 6
        → (if (= x y) ...)
          (= x y) → T
          X → 6
        → 6
      → Result of GCD is 6
    ⇒ 6
  → 6
→ Result of GCD is 6
```

The form of evaltrace diagrams for LET expressions exposes the fact that the expressions `(if (< x y) ... )` are evaluated in a context that cannot possibly be affected by the "new" versions of the variables

X and Y created in the LET body. Therefore, these variables are effectively created and assigned in "parallel."

Recall from before that the hollow arrow drawn for the LET body's contour indicates that its parent is the immediately enclosing contour. This nesting of contours (*i.e.*, the child-to-parent relationships between contours) is completely determined by the textual inclusion of LET bodies in functions. In the above example, the parent for the local contour created by the LET expression is GCD's contour because GCD textually includes the LET expression. This is, in fact, a general rule for lexical scoping in Lisp: a contour *A* can be the parent of another contour *B* only if the Lisp code corresponding to contour *A* textually includes the code for *B*.

This all implies, also, that a function's environment always mirrors the textual inclusions in the code. This fact is important for understanding closures, which we discuss in the next section.

Getting back to our discussion of LET, it should be noted that this parallel behavior of LET is not always what one desires when creating local variables. For example:

```
(defun price-change (name old new)
  (let ((diff (- new old))
        (proportion (/ diff old))
        (percentage (* proportion 100.0)))
    (list name 'changed 'by percentage 'percent)))
```

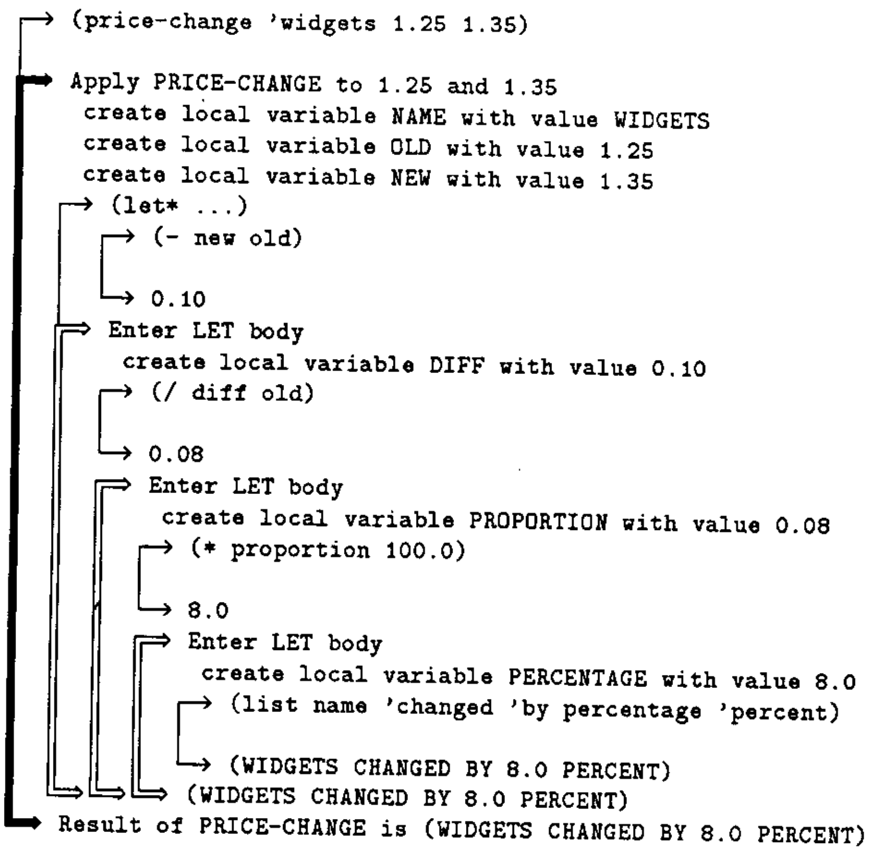
Calling this function leads to an error, as the following evaltrace illustrates:

```
┌→ (price-change 'widgets 1.25 1.35)
│
┣━→ Apply PRICE-CHANGE to WIDGETS, 1.25 and 1.35
┃    create local variable NAME with value WIDGETS
┃    create local variable OLD with value 1.25
┃    create local variable NEW with value 1.35
┃  ┌→ (let ...)
┃  │  ┌→ (- new old)
┃  │  │  new ⟶ 1.35
┃  │  │  old ⟶ 1.25
┃  │  └→ 0.10
┃  │
┃  │  ┌→ (/ diff old)
┃  │  │
┗━→└→ └→ Error!  DIFF unassigned variable.
```

The value of DIFF is needed in order to compute the value for PROPORTION, but the local variable DIFF has not yet been created. Therefore the reference to DIFF is interpreted incorrectly as a reference to a global variable by that name. The global variable DIFF has not been assigned a value (or one might say it doesn't exist), hence the error message. The problem can be remedied by carrying out the creation and assignment of the three local variables serially, using the LET* special function in place of LET.

```
(defun price-change (name old new)
  (let* ((diff (- new old))
         (proportion (/ diff old))
         (percentage (* proportion 100.0)))
    (list name 'changed 'by percentage 'percent)))
```

The evaltrace for this goes as follows:



```
→ (price-change 'widgets 1.25 1.35)

→ Apply PRICE-CHANGE to 1.25 and 1.35
   create local variable NAME with value WIDGETS
   create local variable OLD with value 1.25
   create local variable NEW with value 1.35
  → (let* ...)
     → (- new old)

     → 0.10
  ⇒ Enter LET body
     create local variable DIFF with value 0.10
     → (/ diff old)

     → 0.08
     ⇒ Enter LET body
        create local variable PROPORTION with value 0.08
        → (* proportion 100.0)

        → 8.0
        ⇒ Enter LET body
           create local variable PERCENTAGE with value 8.0
           → (list name 'changed 'by percentage 'percent)

           → (WIDGETS CHANGED BY 8.0 PERCENT)
  ⇒     ⇒     ⇒ (WIDGETS CHANGED BY 8.0 PERCENT)
→ Result of PRICE-CHANGE is (WIDGETS CHANGED BY 8.0 PERCENT)
```

LET* generates a new, nested contour for each variable it creates. The evaltrace diagram suggests that LET* expressions behave like nested LET expressions.

5 Closures

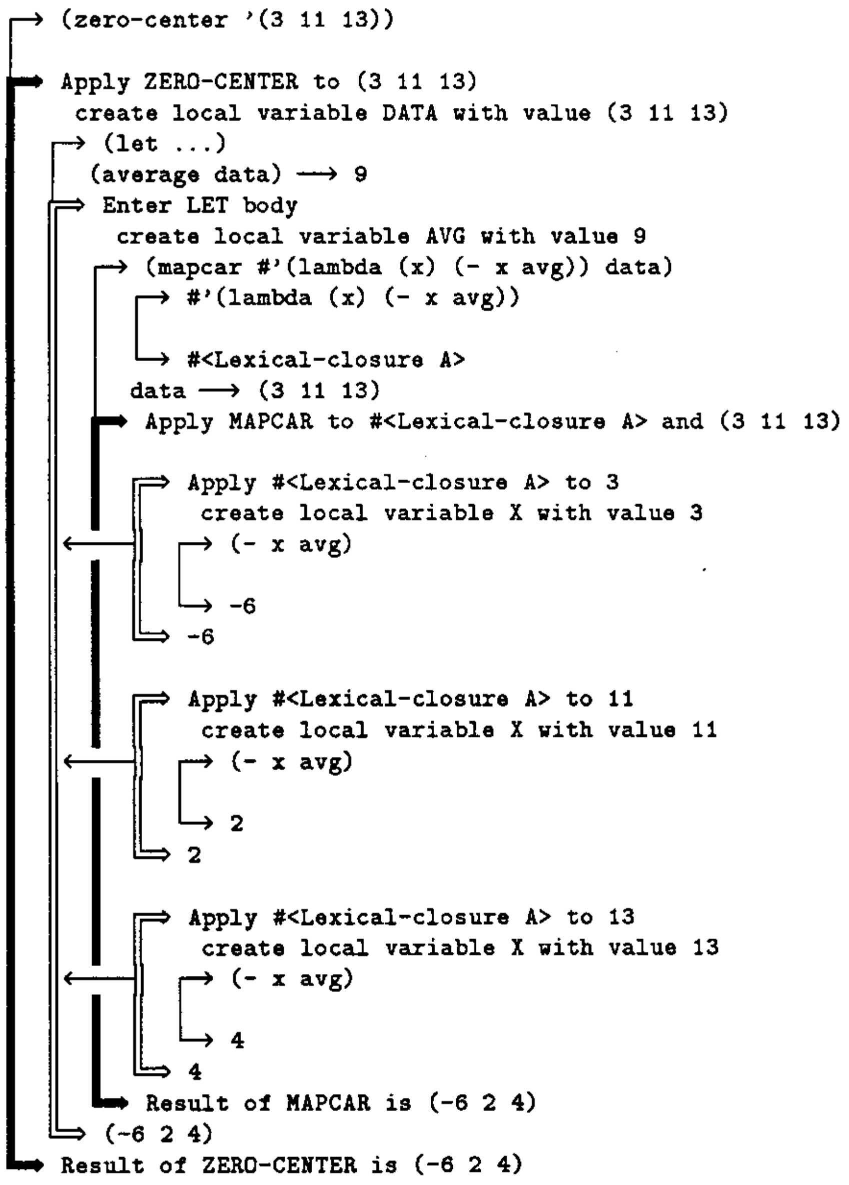
In Lisp, functions are first-class data objects: they can be created on the fly, passed as arguments, and returned as values. Function objects are created in Common Lisp by passing a lambda expression to the special function FUNCTION.[1] The result of FUNCTION is a new function object whose parent contour is the current contour, and whose parameter list and body are taken from the lambda expression. These function objects are also known as *lexical closures*. We will denote these function objects as `#<Lexical-closure A>`, `#<Lexical-closure B>`, and so on.

One of the most common uses for lexical closures is as arguments to applicative operators such as MAPCAR. In the example below, ZERO-CENTER takes a list of numbers as input and adjusts them so that their sum is zero without changing any of the distances between pairs of individual elements. It does this by subtracting the average of the list (computed by AVERAGE) from each element. ZERO-CENTER creates a function object (a lexical closure) to do this subtraction, and passes it to MAPCAR so that it can be applied to each element in succession.

[1]In Scheme, LAMBDA is itself a special function that creates function objects, so there is no need for a FUNCTION function. In Common Lisp, FUNCTION is usually abbreviated `#'` just as QUOTE is abbreviated `'`.

```
(defun average (seq)
  (/ (reduce #'+ seq) (length seq)))

(defun zero-center (data)
  (let ((avg (average data)))
    (mapcar #'(lambda (x) (- x avg)) data)))

(zero-center '(3 11 13)) ⟶ (-6 2 4)
```

The function object passed to MAPCAR contains a reference to the variable AVG in its body. Therefore, it is important that the closure's parent contour be the contour where the closure was created, not the global contour, in order for the closure to be able to access ZERO-CENTER's local variable AVG.



```
→ (zero-center '(3 11 13))

→ Apply ZERO-CENTER to (3 11 13)
   create local variable DATA with value (3 11 13)
  → (let ...)
   (average data) ⟶ 9
  ⇒ Enter LET body
     create local variable AVG with value 9
    → (mapcar #'(lambda (x) (- x avg)) data)
      → #'(lambda (x) (- x avg))

      → #<Lexical-closure A>
     data ⟶ (3 11 13)
    → Apply MAPCAR to #<Lexical-closure A> and (3 11 13)

      ⇒ Apply #<Lexical-closure A> to 3
         create local variable X with value 3
        → (- x avg)

        → -6
      ⇒ -6

      ⇒ Apply #<Lexical-closure A> to 11
         create local variable X with value 11
        → (- x avg)

        → 2
      ⇒ 2

      ⇒ Apply #<Lexical-closure A> to 13
         create local variable X with value 13
        → (- x avg)

        → 4
      ⇒ 4
    → Result of MAPCAR is (-6 2 4)
  ⇒ (-6 2 4)
→ Result of ZERO-CENTER is (-6 2 4)
```

Notice that MAPCAR's contour is interposed between the contours created by the LET body and the contour for each invocation of #<Lexical-closure A>. MAPCAR's parent contour is the global contour.

But this doesn't matter, because the closure doesn't find its parent by looking for the most recently created contour (*i.e.*, just below the top of the call stack). The parent is determined at the time the closure is defined, and it is remembered explicitly as part of the closure object itself. In the evaltrace diagram, each time the closure is invoked by MAPCAR, its parent contour is shown by a leftward-pointing arrow that appears to "jump over" the MAPCAR contour to point to the LET body where the closure was defined. What's really happening is that the closure is "remembering" that the LET body's contour is its parent; it ignores the contour created by the invocation of MAPCAR.

This behavior is consistent with our textual-inclusion explanation. The text for the lambda expression that gave rise to the closure appears inside the body of the LET, so the LET forms part of the closure's environment. There is no reason for the closure to be able to access any of the local variables of MAPCAR, since it does not appear within the body of the definition of MAPCAR.

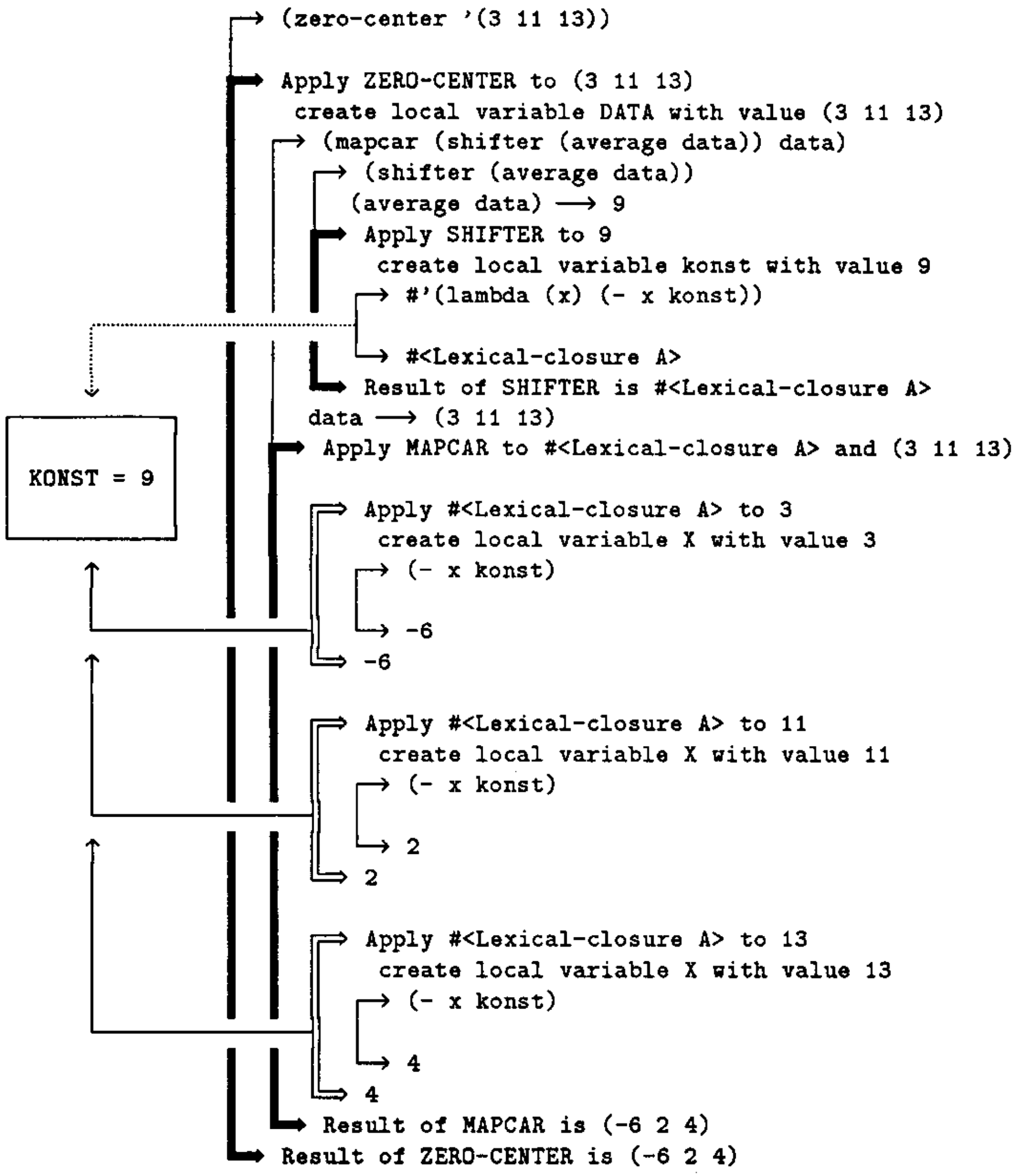
Functions created solely to be passed as arguments are known as *funargs* in Lisp. They have the property that their parent contour is always somewhere below them on the call stack, although as in the preceding example it will not be immediately below them. Early Lisp dialects (as well as many block-structured languages like Pascal and Ada) allowed function objects to be used only as funargs: a function object could be passed to other functions, but it could not be returned as a value by the function that created it, because then its parent contour would no longer be on the call stack. In Scheme and Common Lisp this restriction has been eliminated; it is possible for contours to remain in existence even after the function call that created them has returned. This is accomplished by locating the contour in heap storage rather than on the stack, when necessary. (Actually, real implementations use many different strategies for representing contours in memory. As a conceptual device, however, it is useful to think in terms of the representation of contours in the heap and stack.) This means that we need a special way to draw contours for functions that are returned as values. Consider the function SHIFTER below, which *returns* a closure that references SHIFTER's local variable KONST:

```
(defun shifter (konst)
  #'(lambda (x) (- x konst)))
```

Below we have rewritten the function ZERO-CENTER in order to demonstrate the use of the closures returned by SHIFTER. ZERO-CENTER calls SHIFTER to create a closure that will subtract the average value from its input. It then uses MAPCAR to apply this closure to ZERO-CENTER's input values, DATA.

```
(defun zero-center (data)
  (mapcar (shifter (average data)) data))
```

When SHIFTER returns a lexical closure, its parent contour, SHIFTER's contour, is preserved on the heap. This is shown in the evaltrace diagram below as a box on the left in which the local variables of the contour reside. The local variable KONST resides in this box. When the closure is invoked within the body of MAPCAR, its parent contour is this preserved contour. Within the body of the closure, the symbol X evaluates to its local value because X is a local variable in the closure's contour; the symbol KONST evaluates to 9 because there is a variable by that name visible in the parent contour.



```
(zero-center '(3 11 13))
Apply ZERO-CENTER to (3 11 13)
 create local variable DATA with value (3 11 13)
   (mapcar (shifter (average data)) data)
     (shifter (average data))
     (average data) ⟶ 9
     Apply SHIFTER to 9
      create local variable konst with value 9
        #'(lambda (x) (- x konst))

        #<Lexical-closure A>
     Result of SHIFTER is #<Lexical-closure A>
   data ⟶ (3 11 13)
   Apply MAPCAR to #<Lexical-closure A> and (3 11 13)

     Apply #<Lexical-closure A> to 3
      create local variable X with value 3
        (- x konst)

        -6
     -6

     Apply #<Lexical-closure A> to 11
      create local variable X with value 11
        (- x konst)

        2
     2

     Apply #<Lexical-closure A> to 13
      create local variable X with value 13
        (- x konst)

        4
     4
   Result of MAPCAR is (-6 2 4)
Result of ZERO-CENTER is (-6 2 4)
```

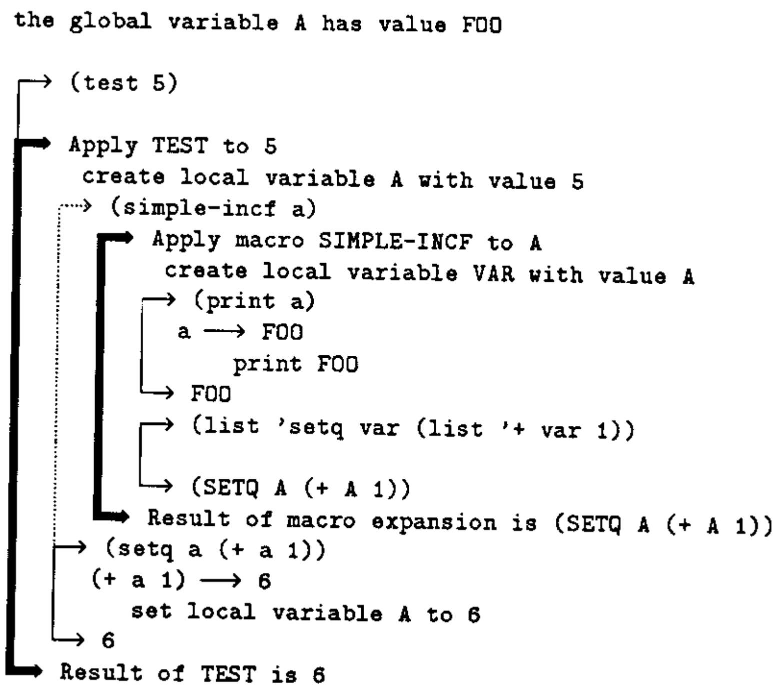
Just like other data objects that are allocated in the heap, the contour for the lexical closure can be garbage collected when there are no longer any pointers to it.

6 Macro Expansion

Macros provide a way to extend the syntax of Lisp. A macro function is applied to its *unevaluated* arguments. Its parent contour is the global contour. The result returned by a macro function is a Lisp expression which is then evaluated in the lexical context where the macro call appeared. As with the duality between EVAL and APPLY, it is often hard to teach beginners how macros work, but evaltrace notation can be used to show macros in action. Consider the SIMPLE-INCF macro, a simplified version of Common Lisp's INCF:

```
(defmacro simple-incf (var)
  (print a)
  (list 'setq var (list '+ var 1)))

(setf a 'foo)

(defun test (a)
  (simple-incf a))
```

An evaltrace of this macro is shown below. Note that the symbol A appearing in the body of the macro is taken as a reference to the global variable A, because the macro's parent contour is the global contour. But the A appearing in the SETQ expression returned by the macro is evaluated within the lexical context of TEST, and so refers to the local variable A visible in the body of TEST.



```
the global variable A has value FOO

→ (test 5)

→ Apply TEST to 5
   create local variable A with value 5
   → (simple-incf a)
      → Apply macro SIMPLE-INCF to A
         create local variable VAR with value A
         → (print a)
           a ⟶ FOO
               print FOO
         → FOO
         → (list 'setq var (list '+ var 1))

         → (SETQ A (+ A 1))
      → Result of macro expansion is (SETQ A (+ A 1))
   → (setq a (+ a 1))
     (+ a 1) ⟶ 6
        set local variable A to 6
   → 6
→ Result of TEST is 6
```

Macro expansions are drawn with a dotted line. The result of a macro expansion is evaluated normally, as shown by the thin solid line to which the dotted line connects. Notice that the argument to the macro is not evaluated, so the input to SIMPLE-INCF is the symbol A instead of the value 5.

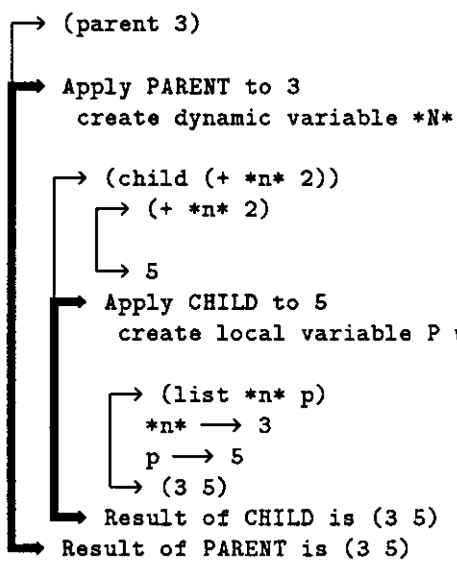
7 Special Variables

Variables in Common Lisp are lexically scoped by default, but it is still possible to have a kind of dynamically scoped variable called a *special variable*. There is no special evaltrace notation for dynamic variables; one simply notes whether a given name has been declared special or not, and once it has, all variables with that name will be dynamically scoped. By convention, special variable names are written with surrounding asterisks, so that they can be easily distinguished from lexical variable names. A variable name can be declared special with the DEFVAR form. Let's return to our earlier PARENT/CHILD example, but this time with a special variable.

```
(defvar *n* 1000)

(defun parent (*n*)
  (child (+ *n* 2)))

(defun child (p)
  (list *n* p))
```

Now, an evaltrace of PARENT goes as follows:

```
the top level dynamic variable *N* has value 1000
```



```
→ (parent 3)

→ Apply PARENT to 3
   create dynamic variable *N* with value 3

   → (child (+ *n* 2))
      → (+ *n* 2)

      → 5
   → Apply CHILD to 5
      create local variable P with value 5

      → (list *n* p)
        *n* ⟶ 3
        p ⟶ 5
      → (3 5)
   → Result of CHILD is (3 5)
→ Result of PARENT is (3 5)
```

```
*N* still has value 1000
```

The evaluation rule for dynamically scoped variables is that we search all the enclosing contours in the order they appear on the call stack, ignoring the contours' pointers to their normal (lexical) parents. The top level dynamic value is used only if we make it all the way out to the global contour, meaning that no other contour presently has a variable with the same name. PARENT "rebinds" the special variable *N*, meaning that it establishes a (temporary) new dynamic variable with the name *N* that lasts as long as PARENT remains on the call stack. Dynamic variables cannot be maintained in closures, since they are entirely dependent on the call stack. PARENT's binding of a variable *N* with value 3 is in effect when CHILD evaluates *N*, so that is the variable that CHILD sees. When PARENT exits, its binding of *N* disappears, and thus the previous binding of *N* becomes visible again.

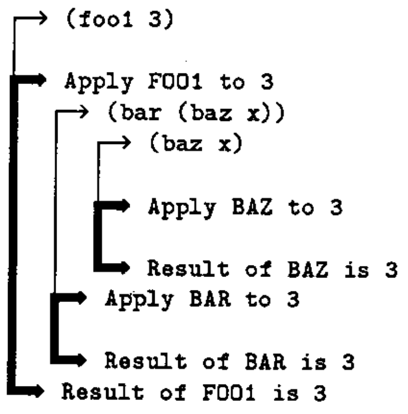
8 Conclusion

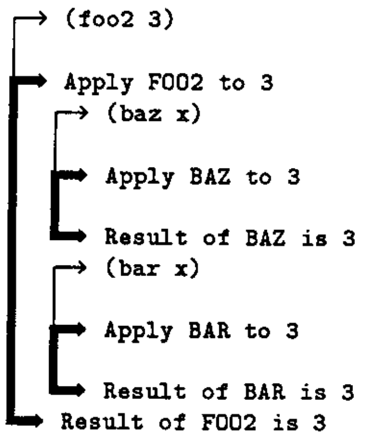
We think evaltrace notation is an effective tool for teaching the key concepts of evaluation in Lisp and other applicative languages. We also find that evaltrace is flexible—extensions to other features of Lisp such as tail-recursion elimination and first-class continuations (CALL/CC) are quite easy to make, though we have not included them here.

Because tracing has been a part of Lisp since the days of Lisp 1.5, we should point out how evaltrace diagrams differ from earlier tracing schemes. Lisp tracing programs usually work by replacing the body of the function to be traced; thus they are only able to show the APPLY part of the evaluation process. This can lead to ambiguities. For example, the functions FOO1 and FOO2 produce identical traces in most Lisp implementations:

```
(defun bar (y) y)

(defun baz (z) z)

(defun foo1 (x)
  (bar (baz x)))

(defun foo2 (x)
  (baz x)
  (bar x))

(trace bar baz foo1 foo2)

> (foo1 3)
 0: (FOO1 3)
   1: (BAZ 3)
   1: returned 3
   1: (BAR 3)
   1: returned 3
 0: returned 3

> (foo2 3)
 0: (FOO2 3)
   1: (BAZ 3)
   1: returned 3
   1: (BAR 3)
   1: returned 3
 0: returned 3
```

In both cases, BAZ is invoked before BAR. But in the first case BAZ is computing the argument to BAR, while in the second case the calls to BAZ and BAR are independent. This difference is clearly portrayed in the corresponding evaltrace diagrams.





Another notation for explaining some aspects of Lisp evaluation is the "fence" notation of Winston and Horn [6]. Fence notation focuses on the static structure of lexical contours; it is not concerned with the dynamic process of evaluation, function invocation, and return. While it can represent the environment of a closure as a series of nested fences, it cannot represent the application of closures, nor does it cover macro expansion. Like most Lisp TRACE notations, fence notation has no way to represent the difference between FOO1 and FOO2 above.

We believe evaltrace notation offers a significant improvement over both earlier notations, in terms of scope of coverage, graphical intuitiveness, and extensibility. Extensions for tail recursion elimination, first-class continuations, assignment, and multiple closures with a shared parent contour are discussed in [5]. We hope that both Lisp novices and educators find the notation to be as useful as we have. In support of this, we have made available a set of LaTeX macros for producing evaltrace diagrams similar to the ones in this paper. They are available via anonymous ftp in compressed tar format, from `a.ergo.cs.cmu.edu` (128.2.250.219), in directory `pub/evaltrace`, files README and `evaltrace.tar.Z`.

References

[1] Gabriel, R. P. (1987) Lisp. In S. C. Shapiro (ed.), *Encyclopedia of Artificial Intelligence*, vol. 1, 508–528. New York: John Wiley.

[2] Rees, J., and Clinger, W. (eds). (1986) The revised^3 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, vol. 21, no. 12, 37–79.

[3] Steele, G. L. Jr. (1984) *Common Lisp: the Language*. Burlington, MA: Digital Press.

[4] Touretzky, D. S. (1990) *Common Lisp: A Gentle Introduction to Symbolic Computation*. Redwood City, CA: Benjamin/Cummings.

[5] Touretzky, D. S., and Lee, P. (in preparation) A graphical representation for evaluation and scoping in applicative languages.

[6] Winston, P. H., and Horn, B. K. P. (1989) *LISP*, 3rd edition. Reading, MA: Addison-Wesley.