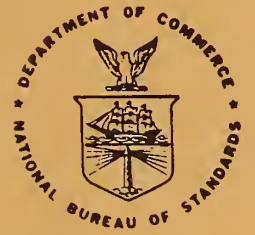
NBS

OED: Object-Oriented Editor

J. C. Boudreaux

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
National Engineering Laboratory
Center for Manufacturing Engineering
Automated Production Technology Division
Gaithersburg, MD 20899

March 1987



U.S. DEPARTMENT OF COMMERCE
AL BUREAU OF STANDARDS

NBSIR 87-3530

OED: OBJECT-ORIENTED EDITOR

J. C. Boudreaux

U.S. DEPARTMENT OF COMMERCE
National Bureau of Standards
National Engineering Laboratory
Center for Manufacturing Engineering
Automated Production Technology Division
Gaithersburg, MD 20899

March 1987

U.S. DEPARTMENT OF COMMERCE, Malcolm Baldrige, *Secretary*
NATIONAL BUREAU OF STANDARDS, Ernest Ambler, *Director*

OED: Object-oriented EDitor *

J.C. Boudreaux

Center for Manufacturing Engineering
National Bureau of Standards

1 Introduction

LISP is one of the oldest programming languages still in active use /5/. It was conceived by John McCarthy and his fellow workers in the late 1950's. Unlike its slightly older predecessor, i.e., FORTRAN, LISP has often been described as the first elegant programming language. Though many beginning users, put off by the piles of parentheses, may find this remark hardly creditable, I believe that it is substantially correct /1/. LISP recognizes one and only one family of compound objects, called *lists*, and this single family provides supports everything in the LISP domain. One measure of the expressive power of lists is that they permit the elimination of such traditional distinctions as those between programs and data.

In /1/ and /2/, I have discussed other features of LISP, including the use of typefree (universal) variables and the definability of higher-order functionals. In this paper, I will describe an object-oriented editor, called *OED*, using which one may create and modify lists. In the papers just cited, I focus on one LISP dialect, called FranzLISP, and will do so again here. However, the constructions given below can be carried with similar ease in other dialects.

Before building the editor, let's examine the notion of lists more closely. As a first approximation, we may say that lists are entities with *heads* and *tails*, such that any object, including list, can be a head; but only a list can be a tail. There is one headless list which is called **nil**. With the exception of **nil**, every list has one and only one head. The head of a list is returned by the primitive LISP function **car**. Every list has one and only one tail, which is returned by the primitive LISP function **cdr**. If **X** is any legitimate LISP object and **L** is a list, then the primitive LISP function **cons**, when applied to **X** and **L**, returns the list whose **car** is **X** and whose **cdr** is **L**. The formation rules for lists are summarized by the following grammar:

$\prec list\succ$ ::= **nil** | ($\prec head\succ \prec tail\succ$)
$\prec head\succ$ ::= $\prec any\_atomic\_entity\succ$ | $\prec list\succ$
$\prec tail\succ$ ::= $\prec list\succ$

The non-terminal $\prec any\_atomic\_entity\succ$ is usually more fully elaborated. Just as set theory can be based entirely on the null set so also is **nil** a formally adequate basis for LISP, thereby explaining its curious role of a component-less (atomic) list. But in practice such a parsimonious basis of atomic entities is extended to include such familiar entities as

*Certain commercial products are identified in this paper in order to adequately specify the experimental procedure. Such identification does not imply recommendation or endorsement by the National Bureau of Standards, nor does it imply that the products identified are necessarily the best available for the purpose.

symbols, strings, integers, and floating point numbers. So long as reasonable choices are made, an actual inventory of atoms will not be necessary. In fact, the following rule is sufficient: atomic entities are those objects which the LISP function **atom** maps onto the Boolean value **t**, i.e., *true*; and lists are those objects which the LISP function **listp** maps onto **t**. Once again, the only objects mapped onto **t** by both functions is **nil**.

The preceding grammar illustrates the structural simplicity of the LISP domain. But even this simple domain can be supported by several notational schemes. Since notational schemes take a bit of effort to appreciate, a few preliminary comments are now in order.

The domain of natural numbers may be defined as the smallest set containing the **0** object and closed under the successor operation **S**:

$$\prec natnum\succ \ ::= \mathbf{0} \mid \mathbf{S}\prec natnum\succ$$

From this grammatical definition, we may suppose that all natural numbers are represented in the following sequence:

0, S0, SS0, ...

But this supposition is only partly correct. The domain of natural numbers is closed under such familiar arithmetic functions as addition, multiplication, and so on. These functions can be expressed in the field of first-order arithmetic in the usual way. This further elaboration of the grammar for natural numbers exactly parallels the elaboration of the primitive LISP functions.

In a sense, the presentation of an object domain shows what features are essential, but to render the domain tractable it is often necessary to present a notational scheme. Of course, the grammar already does this, e.g., the natural number *three* is **SSS0**, but such tally-based notational schemes are wildly inefficient. The Arabic notational scheme is a major historical achievement, and one that required creative insight.

Though list objects are not nearly as familiar as natural number objects, the principles underlying the construction of efficient notational systems must still be applied. In this case, as in the preceding case, the natural grammar is not very efficient, which means that the suggested rules for putting parentheses requires many more than are actually needed. The customary notation for lists may best be seen in an example:

(one two three),

which is the list whose **car** is **one** and whose **cdr** is **(two three)**. In the natural grammar, this list object would be parenthesized:

(one (two (three nil)))

which shows the advantage of the modified scheme.

Finally, let's define the notion of equality on the domain of lists. As a first approximation, we may say that two lists **one** and **another** are equal if an only if the **car** of one is **equal** to the **car** of **another**, and the **cdr** of **one** is equal to the **cdr** of **another**. This definition is inadequate. To define **equal** recursively, then we need ground cases to stop the recursive descent. The ground cases are precisely the instances of $\prec any\_atomic\_entity\succ$. If we assume that an equality relation has already been defined on this symbolic collection, say **atom_equal**, then a more accurate definition of **equal** may be presented in LISP as follows:

```
(def equal
   (lambda (X Y)
      (cond
         ((and (atom X)(atom Y))
          (atom-equal X Y))
         ((and (listp X)(listp Y))
          (and
             (equal (car X)(car Y))
             (equal (cdr X)(cdr Y)))
         (t nil))))
```

2 The Editor Defined

In the conventional sense, an editor is a collection of functions to manipulate textfiles. In this paper, the term is being used in a related sense for a collection of functions to manipulate list objects.

The *Object-oriented EDitor* (OED) consists of a set of LISP functions which may be loaded into the LISP environment. Once loaded, an editing session is started by introducing the object to be edited as the argument of the function **oed-edit**.

```
(def oed-edit
   (lambda (obj)
      (setq $obj obj)
      (setq $tmp $obj)
      (setq $lmap nil)
      (cond ((full-list $obj)
                (setq $view (car $tmp)))
            (t (princ "Illegal Object") ))
))
```

Whenever the user supplies an argument, this function initializes the global variables: **$obj**, which designates the object being edited; **$tmp**, which designates the component actually being reviewed; **$view**, which designates the current position of the OED cursor; and **$lmap**, which designates a map-like trace of the current position. The peculiarities of **nil** as the only atomic list have already been noted. To block such LISP errors as would be occasioned by car-ing **nil** or any other atom, the function **oed-edit**, and other functions in this module, make use of the auxiliary function **full-list**:

```
(def full-list
   (lambda (lispval)
      (and (listp lispval)
         (not (equal lispval nil)))
))
```

This function is Boolean-valued and returns **t** if its argument is a non-nil list; otherwise, it returns **nil**.

To focus our attention on the issues raised by these functions, it is best to consider a specific example:

```
-> (oed-edit '(A (B C) D))
```

then the global variables are assigned the following values:

$obj = (A (B C) D)
$tmp = (A (B C) D)
$view = A
$lmap = nil

Henceforth, the semantic interpretation of OED functions will be explained in terms of the effect of the function on the global variables. This can be done very easily by using the special "box" symbol □ to mark the current position of the OED cursor:

(□A (B C) D)

The cursor will always be positioned immediately to the left of **$view**.

Many computer scientists, especially those who advocate structured or disciplined programming, would deplore the use of global variables. Since most dialects of LISP have no mechanism for controlling the visibility of global variables, it is impossible to deny the dangers of allowing uncontrolled access to variables. Some protection is provided by using special functions to retrieve global values:

```
(def oed-object (lambda () $obj))
(def oed-view (lambda () $view))
```

But this is at best only a partial solution. A much better solution, and one adopted in *AMPLE*, is the construction of *workspaces*. Workspaces are environments in which the visibility of and access to variables are carefully controlled by the end user. This construction will be described in a forthcoming publication.

2.1 Position Control

The LISP functions **oed-right**, **oed-left**, **oed-down**, and **oed-up** are used to control the position of the OED cursor. The most reasonable approach is to study first **oed-right** and **oed-left**, and then **oed-down** and **oed-up**.

```
(def oed-right
  (lambda ()
    (cond ((and (full-list $tmp) (full-list (cdr $tmp)))
           (setq $tmp (cdr $tmp))
           (setq $lmap (cons 'd $lmap))
           (setq $view (car $tmp)))
          (t t))
))
```

The effects of the positioning functions using the example given above are recorded in Figure 1. Notice that another **oed-right** after the final one shown would bring about no further modification of the global values. That is, since **(cdr $tmp)** now has the value **nil**, the only non-nil **cond** clause in the definition of **oed-right** is the *otherwise* clause, which simply returns **t** and has no effect on the values of the global variables.

Intuitively, **oed-left** should be the converse of **oed-right**, and it is: **oed-left** un-does the effects of the last **oed-right**. For this reason, **oed-left** is a more complicated function than **oed-right**, if only because it makes essential use of the global variable **$lmap**:

```
(def oed-left
   (lambda ()
      (cond ((equal (car $lmap) 'd)
             (setq $lmap (cdr $lmap))
             (cond ((null $lmap)
                    (setq $tmp $obj)
                    (setq $view (car $tmp)))
                   (t
                    (setq $tmp
                     (apply
                      (concat 'c (implode $lmap) 'r)
                      (list $obj)))
                    (setq $view (car $tmp))
                    )))
            (t t))
))
```

Recall that if **oed-right** has any effect at all on the global variables, then the character d is pushed on the stack **$lmap**. The outermost **cond** of this definition says that a **oed-left** move is permissible, only if the **car** of **$lmap** is equal to d; otherwise, the only effect of this command is the return of the value t. If the antecedent of the first **cond** clause is satisfied, then **$lmap** is popped, and the inner **cond** is evaluated. If **$lmap** is null, then we must have already returned to the top of the object, with the anticipated effect on the global variables. On the other hand, if **$lmap** is not null, then the effect is to **apply** the function whose name is constructed by evaluating

```
(concat 'c (implode $lmap) 'r)
```

to the global variable **$obj**. Given the special properties of the **c...r** context for the LISP interpreter, **$lmap** must always be a list of d's and a's, a fact which will be obvious when we consider the final positioning functions. The last transaction show in Figure 1 shows the effect of one application of **oed-left**.

The positional operations so far considered have been based on **cdr**, i.e., a **oed-right** move is accomplished by **cdr**-ing **$tmp**, and a **oed-left** move by inverse **cdr**-ing **$tmp**, if we are prepared to understand the second notion in the sense just defined. The next pair of cursor control functions are based in a similar way upon **car**-ing (**oed-down**) and inverse **car**-ing (**oed-up**). Let's first consider the definition of **oed-down**:

```
(def oed-down
   (lambda ()
      (cond ((atom $view) t)
            (t (setq $tmp $view)
               (setq $lmap (cons 'a $lmap))
               (setq $view (car $tmp))
            ))
      ))
```

According to this definition, if **$view** is an atom, then then **oed-down** simply returns the value **t** and has no effect on the global variables. But if **$view** is a list, then **$tmp** is set to value **$view**, the character a is pushed onto **$lmap**, and the new value of **$view** is set to the **car** of **$tmp**. The effect of this operation is shown in Figure 2. Note carefully the

current values of the global variables. If another **oed-down** move were to be attempted, it would result in no change to the global variables, since **$view** is atomic.

The inverse operation **oed-up** is more complicated. To see the reason for this complexity, let's try to describe this operation intuitively. Clearly, an **oed-up** operation presupposes a previous **oed-down**. Thus, if **$lmap** is null, then there are no preceding **oed-downs**, which means that the operation should re-initialize these values and return. If **$lmap** is not null, then we have to repeatedly pop this stack until we find the trace of the last **oed-down**, indicated by an **a**. Having discovered this trace – or emptied the stack – we then have to pop the stack once more, i.e., set **$lmap** to the **cdr** of **$lmap**, and then perform the same kind of inverse operation as we did above. This algorithm, including the iterative search for the last **oed-down** trace, is expressed in the following definition:

```
(def oed-up
   (lambda ()
      (prog ()
      here
      (cond ((null $lmap)
             (setq $tmp $obj)
             (setq $view (car $tmp))
             (return))
            ((equal (car $lmap) 'd)
             (go here))
            ((equal (car $lmap) 'a)
             (setq $lmap (cdr $lmap))
             (setq $tmp
              (apply
               (concat 'c (implode $lmap) 'r)
               (list $obj)))
             (setq $view (car $tmp))
             (return)))
)))
```

Iteration is expressed by the LISP **prog** feature, in the scope of which such unbound variables as **here** are understood to be location markers or labels. In fact, it is only in this micro-environment that this primitive but useful control structure, with its attendant **go**, is allowed.

2.2 Inserting and Appending

The position control functions do not effect the value of **$obj**. Indeed since there only power is to alter the position of an imaginary cursor, it would be wrong to allow them to introduce any such change. In this section we will define two functions which do enable us to change this global variable: **oed-insert** and **oed-append**. These functions differ in the relative placement of the added material. That is, **oed-insert** places all of the components of **text**, in order, just before the **$view**; and **oed-append** places then just after. These functions are iterative versions of the primitive auxiliary functions **basic-insert** and **basic-append**, which are defined immediately after the main functions.

```
(def oed-insert
   (lambda (text)
```

```
        (prog (inval)
           (setq inval (reverse text))
           here
           (cond ((null inval) (return))
              (t (basic-insert (car inval) $tmp)
                 (setq inval (cdr inval))
                 (go here))))
        (setq $view (car $tmp))
))

(def basic-insert
   (lambda (lispval place)
      (cond ((null lispval) nil)
            (t (attach lispval place)))
))
```

The function **oed-insert** is a function which obtains the text to be inserted, reverses the order of the text components, and then inserts each component by iteratively **cdr**-ing through **inval**. The function **basic-insert** is an operation which destructively modifies the second argument by *gluing* it as a **cdr** onto the LISP object

(cons x nil)

For completeness, I include a definition of **attach** which is adapted from Kaisler /4/:

```
(def attach
   (lambda (x lst)
      (cond ((listp lst)
                 (rplaca
                  (rplacd lst (cons (car lst)(cdr lst)))
                  x))
            (t t))
))
```

The transactions in Figure 3 show the effect of **oed-insert** in our familiar test case. The need for destructive manipulations is to ensure that the modifications are uniformly visible, even through the global variable **$obj**.

The second operation in this section is **oed-append** which differs from **oed-insert** with respect to the placement of the appended text. As before, it will be convenient to distinguish **oed-append** from the auxiliary function **basic-append**:

```
(def oed-append
   (lambda (text)
      (prog (apval)
         (setq apval (reverse text))
         here
         (cond ((null apval) (return))
            (t (basic-append (car apval) $tmp)
               (setq apval (cdr apval))
               (go here))))
))
```

```
(def basic-append
   (lambda (lispval place)
      (rplacd place (cons lispval
                  (cdr (copy place))))
))
```

2.3 Replacing and Removing

The preceding sections have concentrated on position control and on the addition of new material either by **oed-insert** or **oed-append**. In this section we will be considering the converse operations which allow the elimination or deletion of material. The first *deletion* operation, called **oed-replace**, makes use of an important relationship between global variables, specifically, the fact that **$view** is always the **car** of **$tmp**. This relation allows a very simple definition of **oed-replace**:

```
(def oed-replace
   (lambda (text)
      (rplaca $tmp text)
      (setq $view (car $tmp))
))
```

In this case, the object being viewed is replaced by the argument of **oed-replace**. Moreover, this change is visible throughout the environment because it is brought about by surgery on the actual lists.

The second *deletion* operation, called **oed-remove**, is a more general and more useful operation. It causes the deletion of the current viewed object, and then it cause the object to be re-glued together. The re-gluing needs careful consideration:

```
(def oed-remove
   (lambda ()
   (cond (( equal (cdr $tmp) nil)
         (cond ((equal (car $lmap) 'd)
               (setq $lmap (cdr $lmap))
               (cond ((null $lmap)
                     (setq $tmp $obj)
                     (setq $view (car $tmp))
                     (rplacd $tmp nil))
                    (t
                     (setq $tmp (apply (concat
                     'c (implode $lmap) 'r) (list $obj)))
                     (setq $view (car $tmp))
                     (rplacd $tmp nil)
                    )))
              ((equal (car $lmap) 'a)
               (setq $lmap (cdr $lmap))
               (setq $tmp (apply (concat
               'c (implode $lmap) 'r) (list $obj)))
               (rplaca $tmp (cadr $tmp))
               (rplacd $tmp (cddr $tmp))
               (setq $view (car $tmp)))
```

```
            (t nil)))
        (t (rplaca $tmp (cadr $tmp))
           (rplacd $tmp (cddr $tmp))
           (setq $view (car $tmp))))
))
```

The definition first determines whether or not **$view** is the rightmost component of **$tmp**. If it is not, then **oed-remove** uses **rplaca** and **rplacd** to replace the **car** and **cdr** of **$tmp** with the **cadr** and **cddr** of **$tmp**, thereby effectively deleting the original viewed object. The **oed-remove** operation is then completed by setting the value of **$view** to the **car** of the modified **$tmp**, as usual.

On the other hand, if **$view** is the rightmost component, i.e., if the **cdr** of **$tmp** is null, then a successful deletion requires additional information. We need to know whether or not **$view** has a component to its **oed-left**. Fortunately, this question can be answered by popping **$lmap**, which must yield one and only one of the following outcomes: the character **d**, which identifies **oed-right** as the previous operation and implies that **$view** has a **oed-left** neighbor; the character **a**, which identifies **oed-down** as the previous operation and implies that **$view** has no **oed-left** neighbor; or an *empty-stack* error, which implies that we are at "root" level and that **$obj** is a list with one and only one component, i.e., a list of the form:

(cons x nil)

for some LISP value **x**. In each case, the appropriate deletion and re-gluing operation is described above. In the third case, the value **nil** is returned, which is precisely the values that is obtained by removing the head of a list with a null tail. The transactions in Figure 4 show the effect of **oed-remove**.

3 An Application

There are many potential applications for OED, but the one which is very important for the AMPLE project is the use of OED to support procedural and data abstraction. The primary benefit of abstraction is to isolate the user from implementational details, while presenting clearly all of the information that the user actually needs. In this section, I will illustrate OED within the context of a deliberately simplified variant of the *AMPLE* lexical processor, called *Lexx*.

As explained in /2/ and /3/, *AMPLE/core* is the collection of LISP representations of entities which are important in the manufacturing domain, including parts, devices, sensors, and manufacturing processes, as well as more familiar entities as data types. Unless carefully protected, building appropriate representations in *AMPLE/core* is a project which would force users to directly manipulate LISP objects. Though this requires a very modest level of skill, it tends to place emphasis on just the wrong set of issues by focusing on minute and tedious implementational details that are of little intrinsic importance. The approach to be discussed in this section introduces a sharp distinction between the LISP representation and the interactive *display form* which is used to create the representation and then to view it. In /2/, an Ada-like display form was adopted, and it will also be illustrated in this section.

For the purposes of this example, let's consider a lexical function, called **buildtype**, which may be used to build a type definition. Two items of information must be supplied: a *symbol*, which is the type name, and a *type constructor*, say **array**. Once this information

is provided, the users must be prepared to respond appropriately to whatever questions the system asks. To keep this discussion within manageable bounds, I have decided not to present the extensive error-checking mechanisms upon which *Lexx* depends; and I have also decided to consider only the **array** type constructor:

```
(def buildtype
  (lambda (type-name)
    (princ
      (concat "Type constructor for " type-name ": "))
    (setq category (read))
    (cond
      ((same category 'array)
       (buildarray type-name))
      (t (princ
       (concat category "Not Implemented."))))
))
```

The function **buildtype**, as presently configured recognizes one type constructor, namely, **array**. This supply can be extended by adding new **cond** clauses, including clauses for such familiar type constructors as **enumeration** and **record**, as well as those less familiar types constructors, such as **net** and **device** that have been introduced in *AMPLE*. Once a type constructor is recognized, control is passed to an auxiliary function which has been designed to build objects of that type. In this case, the auxiliary function is **buildarray**:

```
(def buildarray
  (lambda (type-name)
    (prog (c-type dim)
    (terpr)(terpr)
    (oed-edit ( cons type-name (list 'array)))
    (oed-right)
       (princ (concat "Component type of " type-name ": "))
       (setq c-type (read))
       (terpr)
       (princ (concat "Dimension of " type-name ": "))
       (setq dim (read))
    (return
      (oed-append (list (cons dim (list c-type)))))
)))
```

This function obtains all necessary information from the user and then invokes OED to constructs the appropriate object, which may then be entered into *AMPLE/core*. Notice that **buildtype** initiates OED with a list whose head is **type-name** and whose tail is a list containing only the type constructor symbol **array**. After the user makes appropriate responses, the edited object is then returned to the calling environment. It should also be noticed that the symbols **c-type** and **dim**, are declared immediately after **prog**, which identifies them as local variables. The following session illustrates the use of this lexical function:

```
-> (buildtype 'VECTOR)
Type constructor for VECTOR: array
Component type of VECTOR: float
Dimension of VECTOR: (1 3)
(VECTOR array ((1 3) float))
```

In this case, the type object is simple and easy to read; but in more complicated cases, some less severe display form would probably be clearer and easier to interpret. Though there are many alternatives, the following function introduces a display form which is superficially Ada-like:

```
(def displaytype
  (lambda (type-object)
    (prog (t-name t-constructor t-definition)
    (setq t-name (car type-object))
    (setq t-constructor (cadr type-object))
    (setq t-definition (cddr type-object))
    (cond
     ((equal t-constructor 'array)
      (terpr)(terpr)
      (princ (concat
               "typedef "
               t-name
               " is array "))
      (princ (caar t-definition))
      (princ " of " )
         (princ (cadar t-definition))
         (princ ";")
      (terpr)(terpr)
      (return t)
     )
     (t (terpr)(terpr)
      (princ "Not Implemented")
      (terpr)(terpr))
      (return nil)
     )
)))
```

The next session shows the effect of **displaytype** on the type definition just constructed:

```
-> (displaytype (oed-object))
typedef VECTOR is array (1 3) of float;
```

4 Conclusion

Subsequent reports in this series will be investigating other components of the *AMPLE* system. This report is being released early on because OED is a fundamental tool which all of the components depend upon. That is, OED is the only mechanism available for the creation and modification of object representations in *AMPLE/core*.

Bibliography

1. Boudreaux, J.C. "Problem Solving and the Evolution of Programming Languages," *The Role of Language in Problem Solving-1*, edited by R. Jernigan, B.W. Hamill, and D.M. Weintraub, North-Holland, 1985; 103-126.

2. Boudreaux, J.C. "AMPLE: A Programming Language Environment for Automated Manufacturing," *The Role of Language in Problem Solving - 2*; edited by J.C. Boudreaux, B. Hamill, and R. Jernigan, North Holland, Amsterdam, 1986; 359-375.

3. Boudreaux, J.C. "The AMPLE Project," *National Bureau of Standards Interagency Report, NBSIR 86-3496.*

4. Kaisler, S.H. *INTERLISP: The Language and Its Usage*, John Wiley and Sons; 1986.

5. Wilensky, R. *LISPcraft*, W.W. Norton; 1984.

-> (oed-edit '(A (B C) D))
(□A (B C) D)

-> (oed-right)
(A □(B C) D)

-> (oed-right)
(A (B C) □D)

-> (oed-left)
(A □(B C) D)

$obj = (A (B C) D)
$tmp = ((B C) D)
$view = (B C)
$lmap = (d)

Figure 1. The first session shows the effect of **oed-left** and **oed-right** on the position of the (imaginary) cursor. The values of the global values after the execution of the last command are given in the table.

-> (oed-down)
(A (□B C) C)

$obj = (A (B C) D)
$tmp = (B C)
$view = B
$lmap = (a d)

-> (oed-up)
(A □(B C) D)

-> (oed-up)
(□A (B C) D)

Figure 2. This session continues the previous one and shows the effects of **oed-up** and **oed-down** on cursor position.

```
-> (oed-insert '(X Y))
(□X Y A (B C) D)

          $obj = (X Y A (B C) D)
          $tmp = (X Y A (B C) D)
          $view = X
          $lmap = nil

-> (oed-right)
(X □Y A (B C) D)

-> (oed-insert '((Z)))
(X □(Z) Y A (B C) D)

          $obj = (X (Z) Y A (B C) D)
          $tmp = ((Z) Y A (B C) D)
          $view = (Z)
          $lmap = (d)

-> (oed-append '(1 2))
(X □(Z) 1 2 Y A (B C) D)

          $obj = (X (Z) 1 2 Y A (B C) D)
          $tmp = ((Z) 1 2 Y A (B C) D)
          $view = (Z)
          $lmap = (d)
```

Figure 3. This session illustrates the effects of **oed-insert** and **oed-append** on the results obtained in the second session. The changes in the global variables clearly exhibit the distinction between these two operations.

-> (oed-remove)
(X □1 2 Y A (B C) D)

-> (oed-remove)
(X □2 Y A (B C) D)

-> (oed-remove)
(X □Y A (B C) D)

-> (oed-remove)
(X □A (B C) D)

-> (oed-left)
(□X A (B C) D)

-> (oed-remove)
(□A (B C) D)

Figure 4. This sessions shows the effects of applications of **oed-remove**. Notice that the final object is precisely the same as the one that we started with.

NBS-114A (REV. 2-80)

U.S. DEPT. OF COMM. **BIBLIOGRAPHIC DATA SHEET** (See instructions)	1. PUBLICATION OR REPORT NO. NBSIR 87-3530	2. Performing Organ. Report No.	3. Publication Date MARCH 1987

4. TITLE AND SUBTITLE

OED: Object-oriented Editor

5. AUTHOR(S)

J.C. Boudreaux

6. PERFORMING ORGANIZATION (*If joint or other than NBS, see instructions*)

NATIONAL BUREAU OF STANDARDS
DEPARTMENT OF COMMERCE
WASHINGTON, D.C. 20234

7. Contract/Grant No.

8. Type of Report & Period Covered

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (Street, City, State, ZIP)

10. SUPPLEMENTARY NOTES

[ ] Document describes a computer program; SF-185, FIPS Software Summary, is attached.

11. ABSTRACT (*A 200-word or less factual summary of most significant information. If document includes a significant bibliography or literature survey, mention it here*)

In this paper an object-oriented editor, called OED, is defined in the FranzLISP programming language. Though editors are usually associated with sets of functions to manipulate textfiles, in this work the term is being used to characterize a family of LISP functions which create and modify formal representations of objects in AMPLE/Core.

12. KEY WORDS (*Six to twelve entries; alphabetical order; capitalize only proper names; and separate key words by semicolons*)

AMPLE/Core; formal representation; FranzLISP; object-oriented editor

13. AVAILABILITY

[X] Unlimited

[ ] For Official Distribution. Do Not Release to NTIS

[ ] Order From Superintendent of Documents, U.S. Government Printing Office, Washington, D.C. 20402.

[X] Order From National Technical Information Service (NTIS), Springfield, VA. 22161

14. NO. OF PRINTED PAGES

18

15. Price

$9.95