Procedure closure in EL1*

Ben Wegbreit

Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts 02138, USA

Most programming languages allow the use of free variables in procedures. The mechanism for connecting such free variables with their intended meanings has significant impact on the convenience of the language for the programmer and the efficiency of the resulting programs. The EL1 programming language in the ECL programming system provides a closure mechanism which allows the programmer considerable control over the binding of free variables and serves as an aid to efficient implementation. In this paper, the closure mechanism for EL1 is explained, its rationale presented, and various applications displayed.

(Received March 1972)

1. Introduction

Most programming languages allow, to some degree, the use of procedures† as values of the language. FORTRAN IV (IBM, 1966), ALGOL 60 (Naur, 1963), PL/I (IBM, 1969), ALGOL 68 (van Wijngaarden, 1969), LISP 1.5 (McCarthy, 1962), and SNOBOL4 (Griswold, 1968) each allow procedure values, with differing restrictions on their use. Also, most languages allow the appearance of free‡ variables in procedures and in procedures which are used as values.

Depending on the language, the values of variables and therefore of free variables may fall into one of a wide variety of functional categories, including numbers, Booleans, strings, lists, structures, etc. as well as procedures, operators, data types, I/O ports, and process handles. Hence, the mechanism for connecting free variables with their intended meaning has significant impact on the convenience of the language as a programming tool and on the potential efficiency of its evaluators.

There are two basic approaches a language can employ in regard to such free variables: 1. Choose some fixed *identification* rule for connecting a free variable with its meaning; 2. Provide sufficient power in the language that a program can specify the meaning of free variables (with possibly different identification rules for different free variables).

In regard to the first choice, three common alternatives are:

(1.1) *static* (or lexical) *identification:* The nesting of procedure in the text determines the identification of its free variables. Used in ALGOL 60, PL/I, SIMULA 67 (Dahl, 1968), and ALGOL 68.

(1.2) *dynamic identification:* Variables are scoped according to flow of control; a free variable is identified with the most recently defined variable of that name. Used in SNOBOL4, and APL (IBM, 1970).

(1.3) *global* (or compool) *identification:* Free variables are identified with entities in one or more common global pools. Used in FORTRAN and JOVIAL (Shaw, 1963).

In regard to the second choice, two viable alternatives are:

(2.1) *computed identification:* The language provides a means whereby the environment of each procedure activation can be computed during execution. This is realised in LISP 1.5 by the FUNCTION device; see Moses (1970) for a particularly thorough discussion. A generalisation of this which handles control as well as identification environments is discussed in Bobrow and Wegbreit (1972).

(2.2) *closure mechanism:* The language evaluator includes a processor for explicit connection of free variables with their intended meanings. If appropriate records are kept, it is often possible to carry out considerable optimisation by exploiting the invariance of these connections.

Here, the two alternatives are not mutually exclusive; they are each useful and serve complementary roles.

The purpose of this paper is to discuss the closure mechanism of EL1. Section 2 discusses the semantics of closure and outlines the implementation. Section 3 discusses the use of closure in program development and its relation to other treatments of free variables. Section 4 presents a series of examples showing how the various forms of closure can be used and the sort of optimisations made possible.

2. The closure facility of EL1

EL1 is an operational programming language.§ It is the language component of the ECL programming system which is currently under continuing development at Harvard University. A description of the language, its design philosophy, and the ECL system is found in Wegbreit (1971); Wegbreit (1972) is a programmer's manual. Appendix A of this paper gives a brief description of EL1 syntax, as needed for reading the examples.

2.1. *Relevant facets of EL1*

Two facets of EL1 require discussion here since they are pivotal to all that follows. These are *storage* allocation and the treatment of *procedures*.

In EL1 (as in ALGOL 68), storage can be allocated either on a

*This work was supported in part by the U.S. Air Force, Electronics Systems Division, under Contract F19628-71-C-0173 and by the Advanced Research Projects Agency under Contract F19628-71-C-0174.

†Throughout this paper, the term *procedure* (or *proc*) is used generically to denote a subroutine-like entity. That is, a unit of code, usually defined by the programmer, that may be called with arguments and which may return a value and/or cause side effects. This includes 'procedures' of ALGOL 60, 'routines' of ALGOL 68, 'functions' of APL, and 'Statement Functions', 'FUNCTION Subprograms', and 'SUBROUTINE Subprograms' of FORTRAN IV.

‡A variable is said to be *bound* in a procedure if it is a formal parameter, or declared local. An appearance of a variable is bound if it is in the scope of a formal parameter or local declaration of that variable name. An appearance of a variable is *free* if it is not bound. A variable is said to be *free* in a procedure if there is at least one free appearance of that variable.

§The current version of EL1 in the ECL programming system runs on the DEC PDP-10 under the TOPS and TENEX monitors. Versions for other machines are contemplated.

stack which grows and shrinks on block entry and exit or in the *heap* where storage blocks remain so long as actively referenced. (Garbage collection reclaims storage blocks no longer actively referenced.) A variable can be created having its storage in either place, depending on the variable's specification. Consider

```
DECL X1: COMPLEX BYVAL CONST(COMPLEX OF
                                13, −14);
DECL X2: COMPLEX SHARED VAL(ALLOC(COMPLEX
                                OF 23, +24));
```

The first declaration creates a variable X1 of type COMPLEX on the *stack* (with initial value $13 - 14i$); the second declaration creates a variable X2 of type COMPLEX in the *heap* (with initial value $23 + 24i$).

In EL1, procedures are values. There are proc-valued constants, proc-valued expressions, and proc-valued variables.

Proc-valued *constants* are of the form

EXPR(formal parameter list; result type) procedure body

Any of these may contain free variables. In the *simplest case*, these will be dynamically identified. Consider, for example,

```
EXPR(M:INT, N:INT; REAL)
        BEGIN M*N > U ⇒ V + W; V − W END
```

This is a proc-constant which takes two INTs M and N and returns a REAL number: either the sum of V and W, or their difference—depending on whether or not M*N exceeds U. The free variables U, V, and W are identified every time they are used in executing the procedure.

Proc-valued *variables* have the data type ROUTINE. They may be created as formal parameters, or by declaration, e.g.,

```
DECL P, F1, F2:ROUTINE;
```

They may be assigned procedure values, either the value of proc-constants

```
P ← EXPR(M:INT, N:INT; REAL)
        BEGIN M*N > U ⇒ V + W; V − W END;
```

or the results of procedure-valued expressions (discussed in section 2.2).

In EL1, all procedure values are *pointers* to the defining code block. In particular, a proc-valued constant is a fixed pointer to a fixed code block; a ROUTINE is a location which can contain a pointer to a code block; assignment to a ROUTINE sets the contents of this location to point to a code block.

2.2. *Closure*

The free variables of a procedure value may be explicitly bound by constructing a *closure*. A closure is also a procedure value. It is the result of executing a *closure expression*† which has the form

```
CLOSURE(proc-value, ⟨closurelist⟩)
```

A CLOSURE specifies that those variables which are free in the proc-value and are members of the closurelist are to be identified when the closure expression is executed. The other free variables in the proc-value remain free. For example,

```
CLOSURE(P, ⟨U, V⟩)
```

takes the value of P (a proc-value)‡ and closes it *with respect to* U and V. Hence, U and V are identified at this point, while W is still free. A proc-value may be closed several times, each time with respect to some set of free variables. Each closure is a distinct, separate *instance* of the general proc-value.

It is useful to extend the concept of closure to include form a parameters. The closure of a proc-value with respect to k of its formal parameters is a new function which takes k fewer arguments. For example,

```
CLOSURE(P, ⟨M, U, V⟩)
```

is a procedure which takes a single argument N and has three variables identified at this point—two of them formerly free variables and the third the 'captured' formal parameter.

A *closure* can be used like any other procedure value. It can, for example, be passed as an argument, assigned to a proc-valued variable, or used as the ⟨proc-value⟩ in another closure expression.

There are a number of different options for closure, each specifying a different class of invariant. The next three subsections outline the major options. A more complete description can be found in the user's manual for closure (Perkovic, 1973).

2.2.1. *Constant closure*

The simplest form of closure binds a free variable to some specific value. For example,

```
CLOSURE(G, ⟨X CONSTANT Y⟩)
```

forms the closure of G with respect to X. The constant value used for X in G is the current value of Y at the point the closure is formed.§

In general, a free variable may be identified with the value of an arbitrary expression evaluated during closure. For example,

```
CLOSURE(G,
        ⟨F CONSTANT BEGIN P1 ∨ P2 ⇒ F1; F3 END⟩)
```

evaluates the BEGIN-END block which delivers the value of F1 or F3 and identifies the free variable F with that value.

2.2.2. *Share closure*

A second form of closure binds the location of a free variable to some specific place, but makes no commitment as to the value of the variable. The effect is to establish a sharing relation. For example,

```
CLOSURE(G, ⟨Q SHAREVAL P⟩)
```

identifies the location of free occurrences of Q in G with the location of the object pointed to by the pointer P. The value of such Q's will be the value of this object; assignments to such Q's will change the value of this object; assignments to this object by other access paths will change the value of such Q's.

In general, share closure has the format

```
CLOSURE(proc-value, ⟨fvar SHAREVAL expr⟩)
```

where fvar is some free variable in proc-value and expr is any expression which evaluates to a pointer value. Applications of this are discussed in section 4.1.

2.2.3. *Mode closure*

A third form of closure specifies the data type, i.e. *mode*, of a free variable. This is the weakest form of invariance, since each of the preceding forms of closure implicitly specify mode. There are cases, however, when nothing else can be bound and mode invariance has a significant pay-off. For example, consider

```
CLOSURE(G, ⟨Z MODEIS COMPLEX⟩)
```

If Z appears as an argument to a generic routine such as plus, knowledge of its mode may enable selection of the appropriate

†For expository simplicity, the closure directives are illustrated as invoked with the syntactic form CLOSURE; however, most of the semantic features can also be obtained as explicit compiler directives.

‡Any expression which produces the same proc-value could have been used in place of P. In particular, we could have written

```
CLOSURE(EXPR(M:INT, N:INT; REAL) BEGIN M*N > U ⇒ V + W; V − W; END, ⟨U, V⟩)
```

§This is the default case used above. Closure with no additional specification is treated as the constant closure of a variable with itself, e.g., CLOSURE(G, ⟨X⟩) means CLOSURE(G, ⟨X CONSTANT X⟩).

generic alternative. Other simplifications associated with the elimination of run-time type testing should be obvious.

2.3. *Example*

As an example of the various forms of closure, consider the procedure

```
F ← EXPR(X:VECTOR(N, REAL); REAL)
    BEGIN
      DECL SUM: REAL BYVAL 0;
      FOR I FROM 1 TO N DO SUM ← SUM + X[I]*
        BASE[I];
      SCALE*SUM
    END
```

F takes an array of N REALs, forms the inner product of that array with the array BASE, and returns the inner product multiplied by SCALE. The variables N, BASE, and SCALE are free in F. Consider a closure of F where N is *constant* closed, BASE is *share* closed, and SCALE is *mode* closed

```
F4 ← CLOSURE(F, ⟨N CONSTANT 4, BASE SHAREVAL
                              PB, SCALE MODEIS INT⟩)
```

This fixes N to the value 4, establishes that the variable BASE is to be stored at the location pointed to by the pointer PB, and declares that SCALE is an integer. The resulting procedure forms the scaled 4-vector inner product of its argument and the current value of the previously chosen base vector.

2.4. *Implementation*

CLOSURE is an executable routine which takes a procedure and a closure list and delivers a new procedure which differs from the input proc as follows:

(*a*) free variables closed with *constant* binding are replaced by the appropriate constant value;

(*b*) free variables closed with *share* binding are replaced by a reference to the designated object;

(*c*) free variables closed with *mode* binding are unchanged but simplifications based on knowledge of their modes may be made.

The generation of this new procedure value by the CLOSURE routine is somewhat simplified by the system's representation of procedures. ECL has two language processors: an interpreter which is driven by a list structure representation of the program text and a compiler which accepts the same list structure representation. CLOSURE accepts procedure values represented as list structure and creates a copy of that list structure with appropriate modifications. *Constants* are represented in the obvious fashion. The objects being *shared* with are represented as pointers from the list structure into the heap. The difficult part of forming the closure lies in detecting simplifications made possible by closure invariants. These typically include removal of tests whose value is known, removal of code which cannot then be reached, and removal of declarations for variables which are then unused.

The proc-value delivered by CLOSURE can either be interpreted directly, or compiled (by calling the compiler as a procedure) and the result executed. In the case of compilation, rather good code can be generated. The compiler sees all *constant* identified variables as constants (e.g., it can use immediate instructions for small integers) and *share* identifications as known addresses.

3. Discussion

It is a truism that the programming of any sizeable project begins by decomposing the task into functional modules. Less obvious, perhaps, is the subsequent activity of connecting these modules to produce a complete system, and the need for tools to aid in the connection process. In forming the connections, it is desirable to (*a*) ensure that communication between modules is confined to the intended access paths and to (*b*) minimise the overhead resulting from the prior decomposition. Share closure is used primarily for the first purpose while constant and mode closure are used primarily for the second.

The minimisation of overhead is particularly important when the program is written in a structured top-down fashion or in an extension to a language with definitional facilities. In the former case, routines are often written somewhat more generally then actually required; in the latter case, extensive use is made of routines which define the unit operations of the problem area. In either case, it is desirable to consider the program as a whole and employ mechanical means for contracting the various layers to produce a more efficient final product. In this contraction, a key activity is the instantiation of routines by macro expansion followed by the simplification of these instances. A major part of the closure implementation is devoted to performing such specialisation and other simplifications made possible by the binding invariants.

Comparing closure with computed identification (*cf.* Section 1), it may be seen that the two are complementary and that neither is a substitute for the other. A facility for computed identification allows great dynamic flexibility in the resulting identifications. However, this flexibility has certain disadvantages. Optimisation of the program based on binding invariances is difficult since bindings are established late. Similarly, it is difficult to verify that communication pools (e.g. designated variables) are used only by authorised modules, since any module activation can connect to any other. Closure, precisely because it imposes more structure on the identification process, handles these and related situations rather well.

4. Applications

4.1. *Share closure*

Share closure identifies a free variable with a storage block having a location, a data type, and a value. The value can be used (as with constant closure) or changed. If a procedure shares a storage block with no one, it has an *own* variable. If several independent procedures share storage, they have a private means of communication (e.g. to be used as a consumer/producer buffer or a mailbox). We consider these in turn.

4.1.1. *Own variables*

Share closure identifies a free variable with some object. If this object is referenced by nothing else, the object is private to the closure and the formerly free variable serves as an *own* variable of ALGOL 60. Such objects may be created by the routine ALLOC, e.g.

```
ALLOC(COMPLEX OF 3, −4)
```

generates a new complex number in the heap with value $3 - 4i$ and returns a pointer to it. Suppose the value of G is a proc with free variable Z. Then

```
G2 ← CLOSURE(G, ⟨Z SHAREVAL ALLOC(COMPLEX
                              OF 3, −4)⟩)
```

assigns to G2 a closure in which Z is an *own* variable initialised to the value $3 - 4i$. Changes to Z in one activation of this closure affect the value of Z seen by other activations. Several share closures of G with respect to Z using distinct ALLOC generations may be created. Each closure is a separate *instance* of the general G proc and has its *own* private Z.

This treatment of own variables has two advantages over that found in ALGOL 60: (*a*) the provision for initial values; (*b*) the provision for several instances of a proc, each with its own *own* variable(s).

4.1.2. *Local compools*

It is frequently desirable for two or more routines to share some common data structures. To reduce the possibility of program-

ming errors and, conversely, to aid the process of proving program correctness, it is desirable to ensure that no other routines can access the data. Further, if the data are referenced frequently by the privileged routines, rapid access is important. This can be accomplished directly by share closure.

Suppose, for example, that APOOL is to be shared between routines F1 and F2 under that name, while BPOOL is to be shared between F2 and F3. Let PA and PB be the only pointers to APOOL and BPOOL, respectively. Then

```
F1 ← CLOSURE(F1, ⟨APOOL SHAREVAL PA⟩);
F2 ← CLOSURE(F2, ⟨APOOL SHAREVAL PA, BPOOL
     SHAREVAL PB⟩);
F3 ← CLOSURE(F3, ⟨BPOOL SHAREVAL PB⟩)
```

establishes the desired sharing, while

PA ← PB ← NIL

guarantees that access is restricted.

4.2. *Constant closure*

Constant closure identifies a free variable with a constant value, thereby *freezing* the value of that variable. We have previously displayed the use of constant closure in freezing the values of integers used as array bounds and index limits. Similar applications include freezing the values of real, integer, and Boolean operands to arithmetic, relational, and Boolean operators. In the case of simple expressions where all operands are so frozen to constant values, the value can be calculated during closure.

4.2.1. *Constant closure of mode-valued variables*

A less familiar example is freezing the values of mode-valued* identifiers. In EL1, variables can be declared to take on mode values which are computed by mode-valued expressions. Such variables can be used wherever a mode value is required, e.g. as the data type specification in a declaration.

Consider, for example, a general concatenation routine which takes two arguments—both SEQuences whose elements are of the same mode M—and which constructs a new array which is their concatenation. The element data type, M, is a free variable

```
CONCAT ← EXPR(X:SEQ(M), Y:SEQ(M); SEQ(M))
         BEGIN
         DECL R: SEQ(M) BYVAL
                 CONST(SEQ(M) SIZE LENGTH
                 (X) + LENGTH(Y));
         FOR I FROM 1 TO LENGTH(X) DO
           R[I] ← X[I];
         FOR I FROM 1 TO LENGTH(Y) DO
           R[I + LENGTH(X)] ← Y[I];
         R
         END
```

The result of executing

CLOSURE(CONCAT, ⟨M CONSTANT CHAR⟩)

is a procedure which concatenates two SEQuences of CHARacters (i.e. STRINGs). In general, if $\mathscr{F}$ is a form whose value is some mode $\mathscr{M}$, then

CLOSURE(CONCAT, ⟨M CONSTANT $\mathscr{F}$⟩)

is a concatenation routine tailored to arrays of $\mathscr{M}$s.

4.2.2. *Closure of procedure-valued variables*

If a variable is proc-valued, then in forming a constant closure with respect to it all appearances of the variable used in procedure calls may be replaced by the appropriate constant value. For example, if FOO is *frozen* to EXPR(. . .; . . .) $\mathscr{B}$, then in a closure, forms such as

FOO(A, B, C)

may be replaced by

(EXPR(. . .; . . .) $\mathscr{B}$) (A, B, C)

where the constant explicit procedure value is directly applied to the arguments A, B, and C. The explicit procedure application may† itself be replaced by an equivalent BEGIN-END block. This has the effect of replacing the overhead of procedure call by the comparatively cheaper cost of block entry and exit.

As an example, consider the procedure

```
P ← EXPR(A:SEQ(REAL), B:SEQ(REAL); INT)
    BEGIN
    DECL COUNT: INT BYVAL 0;
    DECL MAXVAL: REAL BYVAL FUM(OLDMAX);
    FOR I TO LENGTH(A) DO FOR J TO LENGTH(B) DO
      BEGIN
      RELATION(A[I], B[J]) ⇒ COUNT ← COUNT + 1
      END;
    COUNT
    END
```

This takes two real arrays, A and B, calls RELATION on all pairs (A_i, B_j), and returns the COUNT of the number of times RELATION is satisfied. RELATION is used as a free variable and hence is identified dynamically. A possible relation is given by

```
R ← EXPR(X:REAL, Y:REAL; BOOL)
    BEGIN
    ABS(X − Y) ⩽ MAXVAL ⇒ TRUE;
    TEST ∧ SIZELIM
    END
```

This is TRUE if X and Y differ by no more than MAXVAL or if TEST and SIZELIM are both TRUE.

With these definitions, consider

P2 ← CLOSURE(P, ⟨RELATION CONSTANT R⟩)

which forms the closure of P with the free variable RELATION constant identified with the current value of R. P2 has the value

```
EXPR(A:SEQ(REAL), B:SEQ(REAL); INT)
BEGIN
DECL COUNT: INT BYVAL 0;
DECL MAXVAL: REAL BYVAL FUM(OLDMAX);
FOR I TO LENGTH(A) DO FOR J TO LENGTH(B) DO
  BEGIN
    BEGIN
    ABS(A[I] − B[J]) ⩽ MAXVAL ⇒ TRUE;
    TEST ∧ SIZELIM
    END
    ⇒
    COUNT ← COUNT + 1
  END;
  COUNT
  END
```

In P2 the call on RELATION has been expanded in-line. This produces several efficiences. There is no function call overhead. Also, register utilisation in the code is better, since the inner block can be analysed in concert with the FOR loops

*EL1 uses the term 'mode' to mean data type. A mode-valued variable is one whose values are data types (e.g. the type INT, the type COMPLEX, the type STRING).

†Since the substitution of a BEGIN-END block for a procedure call entails inline substitution, this action trades space for time. Only the programmer can decide, in general, whether or not the substitution is worthwhile. Hence, two forms of *constant* closure are provided for proc variables: CONSTANT, which does the BEGIN-END block substitution, and SCONSTANT, which does not. The semantics of the two are identical: a free proc variable is identified with a constant proc value. Pragmatically, they differ since all SCONSTANT identifications share the single proc value, while each CONSTANT identification has its own substituted BEGIN-END block.

Further, the declarations of X and Y have been removed since A[I] and B[J] can be used in their place. Finally, note that in-line expansion of this sort may cause additional variables to be identified. For example, MAXVAL is free in the above definition of RELATION. However, in the closure, MAXVAL of the in-line expansion becomes identified with the MAXVAL of the outer block so that MAXVAL may now be treated as a local in all its appearances.

4.2.3. *Evaluation of forms during closure*

It was observed above that simple expressions may be evaluated during closure if the operands were frozen (e.g. by constant closure). This extends to forms of all sorts. Let P be a procedure being closed and let P contain a proc call of the form

$$F(A_1, \ldots, A_n)$$

Suppose that F, $A_1, \ldots, A_n$ are all frozen in the closure of P, that F depends *only* on the values of $A_1, \ldots, A_n$, and that F has no side effects. Then the proc call may be executed during closure with the result replacing the proc call in forming the closure of P. Provided that $F(A_1, \ldots, A_n)$ is actually executed in the program, this preserves functionality while speeding up execution. The requirement that $F(A_1, \ldots, A_n)$ be executed is necessary to ensure functionality as well as speed-up. If $F(A_1, \ldots, A_n)$ does not terminate but P is written so that the call on F is never actually reached, then it may be that the value of P is well-defined in the original program, yet the attempt at closure-time optimisation diverges.

For F to be thus independent of its environment, F must contain no free variables, perform no I/O, make no destructive assignments to arguments passed by reference, and not 'escape' its scope by constructing and then executing program units which do any of the above. These questions are, of course, undecidable for any non-trivial language. It is possible, however, to check that F is *guaranteed safe*—thereby excluding, perhaps, procedures which might be safe but for which the verification is too difficult. For many practical purposes, this will suffice.

4.2.4. *Partial function application*

In any closure of a procedure, the variables identified may be either free variables or formal parameters of that procedure. Taking the constant closure of a formal parameter has a particularly useful interpretation: partial application of a procedure.

As an example, consider

```
DIST ← EXPR(W:COMPLEX, Z:COMPLEX; REAL)
        BEGIN ((W.RE − Z.RE)**2 + (W.IM − Z.IM)
          **2)**.5 END
```

which computes the distance between two complex numbers W and Z. Let U be a complex number with value $-i$. Then

D2 ← CLOSURE(DIST, ⟨Z CONSTANT U⟩)

is a new procedure which takes a single argument W and computes the distance between that point and the point $(0, -i)$.

5. Conclusion

The closure mechanism for EL1 allows the programmer to specify binding information along several axes. For each axis, an attempt is made to provide the best closure consistent with that specification. In particular, considerable effort is made to use binding information to simplify the closures. Hence, the closure mechanism serves two complementary roles: removal of free variables with attendant independence of invocation environment and specialisation of procedure values to exploit this independence.

Acknowledgements

The closure mechanism is being implemented by Paul Perkovic, the compiler has been implemented by Glenn Holloway; their contributions to the design and realisation of closure in EL1 are gratefully acknowledged.

Appendix A. A brief description of the EL1 syntax

To a first approximation, the syntax of EL1 is like that of ALGOL 60 or PL/I. Variables, subscripted variables, arithmetic expressions, Boolean expressions, assignments and procedure calls can all be written as in ALGOL 60 or PL/I. Further, EL1 is—like ALGOL 60 or PL/I—a block structured language. Executable statements in EL1 can be grouped together and delimited by BEGIN-END brackets to form blocks. New variables can be created within a block by declaration; the scope of such variable names is the block in which they are declared.

The syntax of EL1 differs from that of ALGOL 60 or PL/I most notably in the form of conditionals, declarations, data type specifiers, and procedure definitions. Also, there are a number of built-in procedures in EL1. For the purposes of this paper, it will suffice to explain only these points of difference.

A1. *Conditionals*

Conditionals in EL1 are a special case of BEGIN-END blocks. In general, each EL1 block has a value—the value of the last statement executed. Normally, this is the last statement in the block. Instead, a block can be conditionally exited with some other value $\mathscr{V}$ by a statement of the form

$$\mathscr{B} \Rightarrow \mathscr{V};$$

If $\mathscr{B}$ is TRUE then the block is exited with the value of $\mathscr{V}$; otherwise, the next statement of the block is executed. For example, the ALGOL 60 conditional

if $\mathscr{B}_1$ **then** $\mathscr{V}_1$ **else if** $\mathscr{B}_2$ **then** $\mathscr{V}_2$ **else** $\mathscr{V}_3$

is written in EL1 as

BEGIN $\mathscr{B}_1 \Rightarrow \mathscr{V}_1$; $\mathscr{B}_2 \Rightarrow \mathscr{V}_2$; $\mathscr{V}_3$ END

(Unconditional statements of an EL1 block are simply executed sequentially.)

A2. *Declarations*

The initial statements of a block may be declarations. A declaration has the format

DECL $\mathscr{L}$: $\mathscr{M}$ $\mathscr{S}$;

where $\mathscr{L}$ is a list of identifiers, $\mathscr{M}$ is the data type, and $\mathscr{S}$ specifies the initialisation. For example,

DECL X, Y: REAL BYVAL A[I];

This creates two REAL variables named X and Y and initialises them to separate *copies* of the current value of A[I]. The specification $\mathscr{S}$ may assume one of four forms:

(*a*) empty—in which case a default initialisation determined by the data type is used.

(*b*) BYVAL $\mathscr{V}$—in which case the variables are initialised to *copies* of the value of $\mathscr{V}$.

(*c*) SHARED $\mathscr{V}$—in which case the variables are declared to be synonymous with the value of $\mathscr{V}$.

(*d*) LIKE $\mathscr{V}$—in which case the variables are synonymous with the value of $\mathscr{V}$ if possible (i.e. modes match); used primarily for variables which are not modified.

A3. *Data types*

Built-in data types of the language include: BOOL, CHAR, INT, REAL, and ROUTINE. These may be used as data type specifiers to create *scalar* variables.

Array variables may be declared by using either of two built-in procedures: VECTOR and SEQ. For example,

DECL B: VECTOR(80, CHAR) BYVAL $\mathscr{V}$;

creates a variable named B which is an array of 80 CHARacters. The initial value of B is determined by the value of $\mathscr{V}$. An array whose size is *not* fixed by declaration but rather is determined by its initial value may be written

DECL C: SEQ(CHAR) BYVAL $\mathscr{V}_2$;

If the value of $\mathscr{V}_2$ is an array of k characters then C is an array of k characters.

Structures may be defined by the built-in procedure STRUCT. For example, the data type COMPLEX is defined to be

STRUCTURE(RE: REAL, IM: REAL)

That is, variables of data type COMPLEX have two components, named RE and IM, respectively, both of which are of data type REAL. If Z is a variable of data type COMPLEX, its components may be accessed by the operation of selection—denoted by a period. For example, Z.RE is the RE component of Z, while

Z.IM ← Z.RE**2

assigns the square of the RE component of Z to the IM component of Z.

A4. *CONST and ALLOC*

Structured and array values may be constructed (on the stack) during program execution by calling the built-in procedure CONST. Such values can be used anywhere a value is required (e.g. in expressions or in the initial value specification of a declaration). For example,

CONST(COMPLEX OF 5, X + 3)

constructs a complex number whose RE component is 5 and whose IM component is the value of X + 3. Similarly,

CONST(SEQ(INT) OF 2, 3, 5)

constructs an integer array whose components are 2, 3, and 5.

It is also possible to construct objects whose values are defaulted according to data type. In the case of arrays, it may be necessary to specify the *size* of the array. This may be done by a SIZE specification. For example,

CONST(SEQ(INT) SIZE F(X) + 12)

generates an array of F(X) + 12 integers, all given the integer default value of zero.

ALLOC is like CONST except that the value generated has its storage allocated in the free storage region and the result returned by ALLOC is a pointer to that value.

A5. *Procedure definitions*

A procedure may be defined by assigning a procedure value to a procedure-valued variable (e.g. a ROUTINE). For example,

IPOWER ← EXPR(X:REAL, N:INT; REAL)
BEGIN DECL R:REAL BYVAL 1;
FOR I TO N DO R ← R*X; R END

assigns to IPOWER a procedure which takes two arguments, a REAL and an INT (assumed positive), and computes X^N.

A6. *Operator definitions*

A procedure-valued variable can be declared to be an operator of several sorts, thereby establishing certain *syntactic* properties. A *procedure-valued variable* declared to be a *prefix* operator can be applied to a single argument without enclosing the operand in parentheses. A procedure taking two arguments declared as an *infix* operation can be used accordingly. (The standard operators such as +, *, −, /, ←, =, and others are defined in this way as part of an initial, system-provided operator set.) A procedure taking n arguments can be declared a *matchfix* operator, in conjunction with a right matching token, For example, if ⟨ is a matchfix operator with right matching token ⟩, then

⟨X, Y + Z, F(A, B/2)⟩

denotes the application of the routine ⟨ to the three arguments X, Y + Z, and F(A, B/2).

A7. *Miscellaneous built-in procedures*

It is frequently useful to determine the number of components in an array. The procedure LENGTH accomplishes this. If, for example, B is an array of 10 integers then LENGTH(B) equals 10.

The object referenced by a pointer may be obtained by applying the procedure VAL. If, for example, P is a pointer referencing a REAL, then VAL(P) is that REAL.

References

BOBROW, D. G. and WEGBREIT, B. (1972). A Model and Stack Implementation of Multiple Environments, BBN Report No. 2334, Bolt Beranek and Newman Inc., Cambridge, Mass. (to appear in *CACM* Oct. 1973).

DAHL, O. J. *et al.* (1968). *SIMULA 67 Common Base Language*, Publication No. S-2, Norwegian Computing Centre, Oslo, Norway.

DIJKSTRA, E. (1967). Recursive Programming, in *Programming Systems and Languages*, Ed. by S. Rosen. New York: McGraw-Hill Book Co.

GRISWOLD, R. E. *et al.* (1968). *The SNOBOL4 Programming Language*, Englewood Cliffs, N.J.: Prentice-Hall, Inc.

IBM (1966). IBM System/360, *FORTRAN IV Language*, Form C28-6515-4.

IBM (1969). IBM System/360, *PL/I Language Reference Manual*, Form C28-8201-2.

IBM (1970). *APL/360 User's Manual*, GH20-0683.

LOMBARDI, L. and RAPHAEL, B. (1964). LISP as the Language for an Incremental Compiler, in *The Programming Language LISP*, ed. by Berkeley & Bobrow. Cambridge, Massachusetts: The M.I.T. Press.

MCCARTHY, J. (1962). *Lisp 1.5 Programmer's Manual*. Cambridge, Mass.: The M.I.T. Press.

MOSES, J. (1970). The function of FUNCTION in LISP, *SIGSAM Bulletin*, July, pp. 13-27.

NAUR, P. (Ed.) (1963). Revised report on the algorithmic language ALGOL 60, *CACM*, Vol. 6, No. 1, pp. 1-17.

PERKOVIC, P. (1973). *A User's Manual for the CLOSURE Function*, Technical Report, Centre for Research in Computing Technology, Harvard University.

SHAW, C. J. (1963). A specification of JOVIAL, *CACM*, Vol. 6, No. 12, pp. 721-735.

VAN WIJNGAARDEN, A. (Ed.) (1969). Report on the algorithmic language ALGOL 68, *Numerische Mathematik*, Vol. 14, pp. 79-218.

WEGBREIT, B. (1971). The ECL programming system, *Proc. FJCC*, Vol. 39, pp. 253-262.

WEGBREIT, B. *et al.* (1972). *The ECL programmer's manual*, Technical Report No. 21-72, Centre for Research in Computing Technology, Harvard University.