ESD-TR-71-341

THE TREATMENT OF DATA TYPES IN ELI

Ben Wegbreit

August 1971

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

Sponsored by: Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

ARPA Order No. 952

(Prepared under Contract No. F19628-68-C-0379 by Harvard University, Cambridge, Massachusetts 02138.)

AD 736414

ESD-TR-71-341

THE TREATMENT OF DATA TYPES IN ELI

Ben Wegbreit

August 1971

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

Sponsored by: Advanced Research Projects Agency
1400 Wilson Boulevard
Arlington, Virginia 22209

ARPA Order No. 952

(Prepared under Contract No. F19628-68-C-0379 by Harvard University, Cambridge, Massachusetts 02138.)

FOREWORD

This report presents the results of research conducted by Harvard University, Cambridge, Massachusetts in support of ARPA Order 952 under contract F19628-68-C-0379. Dr. John B. Goodenough (ESD/MCDT-1) was the ESD Project Monitor.

This technical report has been reviewed and is approved.

[illegible]

EDMUND P. GAINES, JR., Colonel, USAF
Director, Systems Design & Development
Deputy for Command & Management Systems

ABSTRACT

In constructing a general purpose programming language, a key issue is providing a sufficient set of data types and associated operations in a manner that permits both natural problem-oriented notation and very efficient implementation. The language EL1 contains a number of features specifically designed to simultaneously satisfy both requirements. The resulting treatment of data types includes provision for programmer-defined data types and generic routines, programmer control over type conversion, and very flexible data type behavior, in a context that allows efficient compiled code and very compact data representation.

TABLE OF CONTENTS

THE TREATMENT OF DATA TYPES IN EL1*

Section 1: Introduction

The prime function of a problem-oriented language is to provide a set of data types and associated operations sufficient to represent the unit objects and operations of its problem domain. This representation must on the one hand be very natural to the programmer and on the other be implementable on computing machines in a very efficient fashion. The success of FORTRAN, SNOBOL, and COBOL is due principally to their respectively providing such representation for scalars and arrays of numbers, strings, and data processing records. Each language has an envelope of applications in which program creation is natural and program execution is efficient. This envelope is determined primarily by the set of data types and operations it provides.

In recent years there has been considerable effort to construct languages with significantly larger performance envelopes [1, 2, 3, 4, 5], that is, languages to serve many or all problem domains. In constructing such languages, the principal problem is providing a sufficient set of data types and associated operations in a manner that affords both naturalness of notation and efficiency of representation. We wish to stress that both considerations are absolutely vital. Either alone can be satisfied fairly easily. However, simultaneously achieving very efficient representation and natural notation for a wide variety of data types is a quite difficult matter. We stress the importance of both considerations because the really significant problems of computer science generally entail difficult programming projects where considerable effort is required for program creation and development and

* Also included as number 4-71 in the publication series of the Harvard Center for Research in Computing Technology.

where the final result must be a very efficient running program.

The purpose of this paper is to discuss how this problem is solved in the programming language EL1. EL1 is a working programming language[†] currently under further development as part of a research project in extensible languages at Harvard University. It has a number of features specifically designed to make possible a very flexible, yet very efficient, treatment of data types. These features, their shaping of data type handling, and their interaction with other aspects of the language are the topics of this paper.

This paper is divided into nine sections of which this is the first. Section 2 is a brief sketch of the language EL1, outlining its main features and establishing the notation to be used in the rest of the paper. EL1 gives a somewhat unconventional treatment to the union of data types; since this concept arises in several contexts, it is examined in detail in section 3. Section 4 describes the basic data type definition facilities of EL1. Section 5 discusses the evaluation of data type definitions and the implications of this to compilation. Section 6 discusses the treatment of generic routines (roughly, routines whose action depends on the types of their arguments). Section 7 discusses type conversion and its interaction with generic routines. Section 8 deals with the more sophisticated aspects of the EL1 data type definition facility: the mechanisms which allow the programmer detailed control over data type behaviors. Section 9 turns from the specific to the general—abstracting the techniques used in EL1 and examining to what extent they can be applied to other problem-oriented languages.

Section 2: A Brief Sketch of EL1

In written appearance, EL1 is a fairly conventional programming language in the Algol 60 tradition. It includes variables and subscripted variables,

[†] The present version of EL1 runs on the PDP-10 under the 10/50 monitor. A version for TENEX PDP-10 is imminent. Versions for other machines are contemplated.

prefix and infix operations, labeled statements and gotos, block structure, procedure calls, and assignments, all written in standard fashion. Many standard forms are somewhat generalized in EL1. For example, assignment is treated as a binary operator whose value is its left-hand operand. Also, blocks have values — the value of the last statement executed. Hence,

```
X ← BEGIN B[J←J+1]←COS(W); FUM(B[J],Y) END
```

adds one to J, then assigns COS(W) to B[J], then applies FUM to B[J] and Y, and finally assigns the result of FUM to X. Conditionals are specific types of statements in blocks. An if-then statement is written with a right-pointing arrow. That is, the Algol 60 construct "if $\mathscr{P}$ then $\mathscr{E}$" is written in EL1 as

$$\mathscr{P} \rightarrow \mathscr{E}$$

For example, in

```
BEGIN
       I ← 0;
    L: A[I←I+1] ← 0;
       I<N → GOTO L
END
```

the loop is repeated until I reaches N. There is a second form of conditional statement, written with a double-shafted arrow, interpreted as: if the left-hand side is true, execute the right-hand side and exit the block with that value. Hence, the Lisp conditional (COND $(\mathscr{P}_1 \mathscr{E}_1)(\mathscr{P}_2 \mathscr{E}_2) \ldots (\mathscr{P}_n \mathscr{E}_n)$) is written in EL1 as

```
BEGIN
       𝒫₁ ⇒ ℰ₁;
       𝒫₂ ⇒ ℰ₂;
         . .
            .
       𝒫ₙ ⇒ ℰₙ
END
```

Simple conditionals, block-exit conditionals and unconditional statements can be freely intermixed. For example, the following block computes an approximate square root of a number A with initial approximation X to within EPS

```
BEGIN
   L: ABS(X**2-A)<EPS => X;
      X ← (X+A/X)/2;
      GOTO L
END
```

The block is exited only when the left-hand side of the first statement is true; when the exit is taken, the value of the block is X.

Variables are either formal parameters to a routine or variables local to a block. In either case, a variable is declared to be of some specific data type and is restricted to contain values of that type throughout its lifetime.

Data types, termed "modes" in EL1, include the following **built-in types**: BOOL (Boolean), CHAR (character), INT (fixed point), REAL (floating point), REF (pointer unrestricted as to the mode of the object it can point to), SYMBOL (corresponding to non-numeric atoms in Lisp), MODE (the data type "data type"), FORM (the Lisp S-expression), and ROUTINE (procedure or operator). From the standpoint of creation, assignment, and use as arguments or formal parameters, all these modes are equally valid. Hence:

```
DECL  I,J: INT;
```

creates integer-valued variables named I and J, while

```
DECL  M1, M2, COMPLEX: MODE;
```

declares three mode-valued variables, and

```
DECL  F1, FOO, FUM, CSIGN: ROUTINE;
```

creates four routine-valued variables. While all these modes are equally valid, they vary considerably in complexity. For example, a BOOL value is a single bit while a MODE value has associated with it all the information needed by the language to implement a data type. However, from the standpoint of the programmer, the complexity is largely invisible. He is concerned only with the behavior of values having these modes: a BOOL value can be used in the left arm of a conditional while a MODE value can be used in declaring the type of a variable.

Objects are distinct from variables in EL1. Variables may name objects, but the mapping is not one-to-one. That is, while each variable names some object, several variables may name the same object (e.g., when an argument is passed by reference to a routine), several variables may name different parts of a single object, and an object may be named by no variable. An object lies either on a block-structured stack (like that of Algol 60) or in the free storage region termed the heap (like that of Lisp or Algol 68). In the former case, the lifetime of an object is concomitant with that of the block in which it was created. In the latter case, an object remains until no variable names it and no pointers reference it. Garbage collection periodically reclaims objects in the heap no longer in use and returns them to the free storage pool.

An object has a mode determined at the time of object creation. An object is created in one of two ways, either implicitly as the result of a declaration, or explicitly by means of the generators CONST and ALLOC. Objects created implicitly by declaration reside on the stack. Objects created by the explicit generator CONST also reside on the stack; objects created by the explicit generator ALLOC reside in the heap. As an example, suppose that the mode COMPLEX has been defined (a definition in the language will be given in section 3), then consider

```
CONST(COMPLEX OF X, Y)
ALLOC(COMPLEX OF X, Y)
```

The first line constructs a complex number on the stack and returns this complex as its value; the second line constructs a complex number in the heap and returns a pointer to the complex number as its value. As an example of how the latter value may be used, consider

```
DECL  P:REF;
        .
        .
        .
P ← ALLOC(COMPLEX OF 3., 4.);
```

The first line creates a pointer-valued variable P unrestricted to the sort of object it can point to; the second line assigns to P a pointer to the complex number 3+4i. Given a pointer such as P, the object pointed to can be accessed by applying the function VAL, e.g.,

```
VAL(P)
```

is the complex number 3+4i. The only means for creating a new pointer value is ALLOC. Hence, pointers point only to the heap, never to the stack.

In EL1, the notion of a routine embraces both procedures and operators. A routine-valued variable may be assigned a routine-value, e.g.,

```
CSIGN ← EXPR (X:REAL; CHAR)
        BEGIN
            X > 0 ⇒ 'P;
            X < 0 ⇒ 'N;
            'Z
        END
```

Here the routine has a single parameter[†] named X of mode REAL, delivers a CHAR value and has a body consisting of a block which computes the sign of its argument and yields the character P, the character N or the character Z. Any routine may be written as a function and applied to its arguments

```
CSIGN(A[J])
```

In addition, a routine-valued variable can be declared to be a prefix operator and then applied to a single argument without enclosing the operand in parentheses. A routine taking two arguments can be declared as an infix operation and used accordingly. The standard operators such as +, *, -, /, >, <, and others are defined in this way as part of an initial, system-provided

[†]The formal parameter X is bound by reference (in the sense of Fortran or PL/I), i.e., an assignment to X would change the value of the argument. Where this is not desired, one can declare that a formal parameter is bound by value (in the sense of Algol 60), in which case a private copy of the value of the argument is made. Assignments to the formal will then affect only the formal and will not change the value of the actual argument.

extension set.

The principal concern of this paper is with mode-valued constants, mode-valued variables, and mode-valued routines. The primitive data types (BOOL, CHAR, INT, REAL, and REF) mentioned previously are examples of mode-valued constants. To make the notion clear, consider

```
DECL P: BOOL;
    .
     .
      .
P ← TRUE;
```

This creates a Boolean valued variable P and later assigns it the Boolean value true (denoted by the Boolean constant TRUE). Analogously, consider

```
DECL M1: MODE;
     .
      .
       .
M1 ← INT;
```

This creates a mode-valued variable M1 and later assigns it the mode-value integer (denoted by the mode constant INT). In addition to the mode constants mentioned earlier, there are two others: NONE and ANY. The former is the mode of the empty object. The latter is the union of all possible modes; section 3 discusses this and other mode unions.

A mode-valued variable may be used in any position where a mode value is required. For example, suppose that after the above assignment has been executed the following block is entered

```
BEGIN
    DECL J: M1;
        .
         .
          .
END
```

In this block, J is an integer valued variable. If M1 had some other mode value, say $\mathscr{M}$, then J would be an $\mathscr{M}$-valued variable.

The concept of mode-valued routine is a logical consequence of treating modes as values. The simplest such routine is MD which takes a single argument and delivers its mode. For example, MD(FALSE)=BOOL and MD(SIN(X))=REAL. A more significant application of mode-valued routines is mode construction. That is, a set of primitive mode-valued routines provides the means for constructing new modes. These primitive mode constructors take modes as arguments and define new modes in terms of these. From the primitive mode constructors, the programmer can define other mode-constructing routines by means of functional composition, iteration, conditionals, and recursion. Mode constructors, primitive and programmer-defined, are discussed in sections 4, 5, and 8.

The above sketch of EL1 treats those facets of the language required for the purposes of this paper. (A more complete explanation of the language and a formal definition of both its syntax and semantics is found in [6].) However, the traditional discussion of a language per se neglects many facets of its usage and implementation. For EL1 those considerations are particularly important, and an understanding of several "extralingual" facets is essential to an understanding of this paper. EL1 is the language component of a programming system called ECL.† The system is used on-line with two fully compatible language processors – an interpreter and a compiler.†† Compiled and interpreted routines may be freely intermixed with no restrictions.

One key point of the ECL system is that there is no rigid "compile time," "load time," "run time" distinction. Routines are interpreted until explicitly

†The system includes the usual facilities for on-line interaction such as a text editor, a trace feature, and a debugging package. It also includes multitasking, multitasking control primitives, and programmer-controlled interrupt processing. An overview of the entire system is given in [7].

††Although there is an interpreter and compilation is optional, the language has been strongly shaped by the expectation that production programs will be eventually compiled. For example, an interpretable-only language could be "type-less" with all variables free to take on values of any type. However, efficiency considerations lead one to a compiler and with it typed variables. Most of the declarative data type information is really of interest only to the compiler. However, to maintain compatibility between language processors, the interpreter verifies that the data type constraints are satisfied.

compiled. Compilation is carried out by calling the compiler as a subroutine and passing as argument the routine to be compiled. The compiler can be called at any point, e.g., while executing a routine. Hence, it is possible to compile a routine several times with very precise control over the degree of "binding." For each compilation of a routine, one can compute certain invariants of that compilation instance and then compile code which reflects these invariants. That is, suppose P is a routine with free variables $I_1 \ldots I_n$. Suppose some k of these $I_{j_1} \ldots I_{j_k}$ are bound to specific values $V_1 \ldots V_k$ and the result is compiled. The code generated will be better, often substantially better, than the code for P had all variables been left free.

The EL1 compiler is called with two arguments – a procedure P and a list L of variables free in P which are to be so bound. For each variable I on L, each appearance of I in P is replaced by its value at the time of compilation. Such a variable is said to be frozen. If a routine identifier, the free variables of the routine, and the arguments to the routine are all frozen (or are otherwise constant) then the routine is evaluated during compilation and the value thus produced takes the place of the call. Applying this rule recursively, arbitrarily large amounts of a routine being compiled may collapse into values, i.e., become frozen. For example, if FOO, FIE, X and Y are all on L, then

FOO(3, Y, FIE(X, Y, 3.2))

may be so frozen. In general, any syntactic unit made up only of constants or frozen forms is said to be evaluatable; it is replaced by its value and becomes frozen. Hence, depending on L, compilation may leave none or all free variables in P and generate very tight or very loose bindings.

Section 3: Mode Union

The concept of mode union is treated rather specially in EL1. As an example of this treatment, recall that the mode constant ANY denotes the "union" of all modes. Consider an assignment to the routine-valued variable F

```
        F ← EXPR (X:ANY; BOOL)      BEGIN ... END;
```

F then takes as argument a value of any mode, e.g.,

```
        F(3), F(3.), F("W"), F(TRUE), F(REAL)
```

are all legal calls on F. In each case, the X of this invocation of F is bound to the argument. The critical point of the EL1 treatment of unions is as follows: in each case the X of this invocation of F takes on the mode of the argument and henceforth cannot change throughout its lifetime. Hence, in the first call on F, X is bound to 3 and becomes an integer. The value of X can be changed by an assignment, e.g.,

```
        X ← 4;
```

but the mode of X is fixed.[†]

In section 4 we discuss modes which act as "restricted" unions, that is, mode_1 or mode_2 or ... or mode_n. There, as here, a formal parameter declared to have such a mode is bound at the time of call to some specific alternative from the set of possibilities; just which alternative is determined by the argument. Subsequent to creation, the formal parameter cannot change from that alternative mode. For example, a negation routine may be defined to have a single parameter whose formal mode is int or real or complex. The mode of the argument determines which one of these is the actual mode.

[†]An analogy may be made with the length of an Algol 60 array. Consider, for example, the Algol 60 fragment

```
        begin real array A[1:N];
               .
                .
                 .
        end
```

A is declared to be an array of length N. For each instance of the block (i.e., block activation), the length of A is fixed to the then current value of N at the time of block entry. Subsequently, the value of A may change but not its length.

The EL1 union differs from a set theoretic union in exactly one respect: after an object is created, its mode is fixed to be a specific alternative. A set theoretic union would allow objects whose modes as well as values could be changed throughout their lifetime.† Clearly, the EL1 union is a subcase of the more general set theoretic union. This restriction is imposed for two reasons: implementation efficiency and linguistic simplicity. By requiring objects to assume some definite (i.e., non-united) mode, there is never the need to allocate extra stack storage to provide for the contingency of a variable changing its mode and thereby assuming a larger size. Among other consequences, this makes possible a stack implementation of the mode ANY; (this, of course, would not be possible for a set theoretic ANY). Since union is treated as postponement of a mode choice, the concept of union does not exist for the evaluator; i.e., each object has a definite unchanging mode. Hence, there is no need for special semantic rules to deal with unions.†† This simplifies both the learning and use of the language. Finally, the treatment of union in EL1 integrates smoothly with the use of generic routines; this is discussed in section 6.

Section 4: Mode Construction

Given sets $t_1, \ldots, t_n$, one can form new sets from these in several ways:

(1) Cartesian product:	$t_1 \times t_2 \times \ldots \times t_n$
(2) self product – definite and indefinite:	t_i^k and $\bigcup_{k=0}^{\infty} t_i^k$
(3) union:	$t_1 \cup t_2 \cup \ldots \cup t_n$
(4) mappings:	$(t_1 \times \ldots \times t_n) \rightarrow t_{n+1}$

†This is, for example, the treatment of mode union given in [8].

††As an example of the sort of issue we thereby avoid, consider the following. Let A be a true set theoretic union of <u>int</u> and <u>bool</u> and let its current value be an <u>int</u>. Suppose A is passed by <u>reference</u> (in the sense of PL/I) to a routine F which takes an <u>int</u> formal parameter named X. Since A currently has an <u>int</u> value, presumably this is legal. What if F uses A free and assigns a <u>bool</u> value to it? Does this affect X?

These have the natural interpretations: structures (in the sense of COBOL or PL/I), arrays, unions, and routines. These four formation rules define classes of modes and, by the usual abuse of notation, four classes of objects belonging to these modes. Corresponding to each formation rule there is a primitive EL1 routine — STRUCT, ARRAY, ONEOF, PROC — which generates new modes of that class. The basic mode definition mechanism of EL1 is the set of primitive mode-valued constants and the set of primitive mode-valued routines. All other modes are generated from these.

STRUCT takes as arguments a list of pairs ($name_i$: $mode_i$), where $mode_i$ is the mode of the ith component and $name_i$ is the symbolic name. For example,[†]

```
LIGHT_BULB ← STRUCT(HOURS_USED:REAL,
                    WATTS:INT,
                    COLOR:CHAR,
                    BURNT_OUT:BOOL)
```

defines a mode of class structure consisting of four fields: a real, an integer, a character, and a Boolean named HOURS_USED, WATTS, COLOR, and BURNT_OUT, respectively. The mode thus defined is assigned to the mode-valued variable LIGHT_BULB. Subsequent to the assignment, the variable may be used as a type declarer

```
DECL  X, Y, Z : LIGHT_BULB;
```

creating variables X, Y, and Z of mode LIGHT_BULB, and three associated objects (in the stack) named by the identifiers. The individual components can be referred to by qualified naming (in the style of PL/I) so that

```
Z.COLOR
```

[†]The promised definition of the data type complex is:

```
COMPLEX ← STRUCT(RE: REAL, IM: REAL)
```

If Z is a complex variable, then Z.RE and Z.IM denote its two REAL components.

is a character field. Alternatively, a component can be selected by an integer subscript so that

```
Z[J]
```

is identical to Z.COLOR if and only if J has the value 3. Assignment of one LIGHT_BULB to another is denoted in the usual fashion

```
X ← Z
```

and copies all components of the structure.

ARRAY generates either definite or indefinite self product, depending on how it is called.

ARRAY(K,$\mathscr{M}$)

generates the mode $\mathscr{M}^K$ while

ARRAY($\mathscr{M}$)

generates the mode $\bigcup_{k=0}^{\infty} \mathscr{M}^k$. In the latter case, the mode is said to be <u>length unresolved</u>. While the mode is length unresolved, any particular instance of such a mode has a fixed length determined at the time the instance is created. For example,

```
CARD ← ARRAY(80, CHAR);
```

defines the mode "array of 80 characters" and assigns it to the mode-valued variable CARD. Any variable of mode CARD

```
DECL  C: CARD;
```

has exactly 80 components which may be accessed by subscripting

```
C[I]
```

The mode "length unresolved array of characters" may be defined by

```
STRING ← ARRAY(CHAR);
```

This creates a mode whose instances may be of any length. The length of each instance is, however, fixed at the time of creation

```
DECL S:STRING BYVAL CONST(STRING SIZE 200);
```

This creates a variable S of mode STRING and initializes it to a STRING of 200 components. Subsequently, the values of S's components may change but not the number of components. The number of components in an array may be determined by applying the primitive routine LENGTH, e.g., LENGTH(S)=200. As with structures, assignment of arrays is written using the assignment operator and copies all components.

ONEOF ($t_1, t_2, \ldots, t_n$) defines a "union" of n alternative modes $t_1 \ldots t_n$, where "union" is used in the sense described in section 3. That is, a variable declared to be of such a mode takes on some specific alternative determined by its initial value. For example,

```
ARITH ← ONEOF(INT,REAL);
SIGN ← EXPR(X:ARITH; ARITH)  BEGIN ... END
  .
   .
    .
SIGN(-13)
  .
   .
    .
DECL Y: ARITH BYVAL P(X)
```

In the second line, the routine is declared to take a single argument which is either INT or REAL. In the call to SIGN, in line 3 an INT is used so that X in this invocation of SIGN is an INT. In the fourth line, Y is declared to be either an INT or a REAL — which one is determined by the mode of P(X) on each execution of this line.

PROC ($t_1, \ldots, t_n; t_{n+1}$) defines the mode "mapping from $t_1 \times \ldots \times t_n$ into t_{n+1}". For example,

```
TRIG ← PROC(REAL; REAL)
```

defines the mode "set of routines which map reals into reals," while

```
CODE ← PROC(CHAR;INT)
```

defines the mode of routines which convert characters into integers.

In addition to the four classes of modes described above, there is a fifth class which arises from other than set theoretic considerations: the class <u>pointer</u>. PTR(t) is the mode "pointers restricted to point to objects of mode t" and PTR($t_1, \ldots, t_n$) is the mode "pointers restricted to point to t_1's or t_2's or ... or t_n's". Here, unlike the situation with EL1 unions, no commitment is made when such a pointer is created. Such a variable may first point to a t_1, later to a t_n, and still later to a t_2. For example,

```
DECL  SP : PTR(INT, REAL, COMPLEX);
```

creates a variable SP whose mode is "pointer to INT or REAL or COMPLEX." Like all pointers, SP is given the default initial value NIL, meaning a pointer to nothing. Assignments to SP may change this value

```
SP ← ALLOC(COMPLEX OF 3., 4.)
```

so that SP points to a complex number whose value is 3+4i.

<u>Section 5: The Evaluation of Mode Definitions</u>

All the primitive mode generators share one common trait — they evaluate their arguments and these arguments may be <u>any</u> syntactic form which yields an appropriately typed value. This trivially leads to multidimensional arrays such as

```
REAL_MATRIX ← ARRAY(ARRAY(REAL));
```

multilevel structures, arrays of structures such as

```
ARRAY(4, STRUCT(RE : REAL, IM : REAL))
```

and structures of arrays and pointers such as

```
STRUCT(A : INT, B : PTR(INT), C : ARRAY(INT))
```

This illustrates only one case of evaluated arguments to mode generators.

Mode-valued variables, conditionals, and other routines are equally acceptable. For example,

```
ARRAY(N**2, F(X))
```

defines the mode: "array of N^2 F(X)'s" where N and F(X) are determined at the point that ARRAY is called. Turning to a more complex example, the following loop computes the mode: "complete binary tree of depth N whose terminal nodes are integers"

```
BEGIN
    DECL  TEMP: MODE BYVAL INT;
    DECL  I: INT BYVAL N;
L:  (I ← I-1) < 0 ⇒ TEMP;
    TEMP ← STRUCT(L: TEMP, R: TEMP);
    GOTO L
END
```

The declaration creates a local variable TEMP of type MODE and initializes it to the value <u>integer</u>. The loop assigns to TEMP successive elements from the sequence

```
STRUCT(L: INT, R: INT),
STRUCT(L: STRUCT(L: INT, R: INT),
          R: STRUCT(L: INT, R: INT)),
              .
               .
                .
```

It should be noted that the EL1 treatment of mode definition is quite different from that found in other programming languages, such as Algol 68. Traditionally, mode definition has been a static operation carried out at compile time. By treating the mode-defining operators as executable routines which evaluate their arguments, EL1 obtains a more flexible and more powerful means of mode creation. The most important single consequence is the notion of programmer-defined, mode-valued routines. Consider, for example, converting the above binary tree generator into a routine.

```
TREE ← EXPR(I: INT BYVAL, M: MODE BYVAL; MODE)
          BEGIN
              L:  (I←I-1) < 0 ⇒ M;
                  M ← STRUCT(LEFT: M, RIGHT: M);
                  GOTO L
          END
```

TREE takes the depth (I) and leaf mode (M) as arguments — both passed by value. The loop is the same as before, except that now it generates the sequence depending on the value of the leaf mode. Hence, TREE(I, M) is the mode binary tree of depth I and leaves of mode M, for any integer I and mode M.

A second example may be of use. Consider defining the mode "multi-dimensional array of order K of Ms." (For illustrative purposes, we use a recursive definition; an iterative one would, in fact, do just as well.)

```
MULTI_ARRAY ← EXPR(K:INT, M:MODE; MODE)
              BEGIN
                 K=0 ⇒ M;
                 ARRAY(MULTI_ARRAY(K-1, M))
              END
```

MULTI_ARRAY of K M's is either M (if K is 0) or is an ARRAY of the result obtained by applying MULTI_ARRAY to K-1 and M. The definition is obvious and would be somewhat uninteresting were it the only one possible. However, there are other ways of constructing multidimensional arrays which, for some purposes, are far superior to the one given above. If, for example, a frequent operation is exchanging entire rows, then it will be advantageous to use an array of pointers to the constituent rows. The generalization of this to order K is defined

```
P_ARRAY ← EXPR(K: INT, M: MODE; MODE)
          BEGIN
             K=0 ⇒ M;
             K=1 ⇒ ARRAY(M);
             ARRAY(PTR (P_ARRAY (K-1, M)))
          END
```

The K=0 case should be obvious; for K=1 we define a conventional array; for higher K's we construct an ARRAY of PTRs to the result of P_ARRAY applied to K-1 and M.

The point of programmer-defined mode routines is that they permit significant functional abstraction. Instead of talking loosely about some collection of related modes, one can define a collection precisely by means of a routine which generates it. Mode sets such as matrices, binary trees, lists, rings, hash tables, etc., of various element types can be defined by their generators. This permits the creation of mode-definition libraries. More important, it allows the programming of algorithms which act on a class such as binary trees without regard to the constituent elements. Only during compilation is it necessary to freeze free variables to determine which specific sort of binary tree. Finally, it allows one to prove properties of mode sets independent of their particular elements by appealing to the properties of the mode set generating routine.

It must be stressed that the considerable generality provided by dynamic execution of mode definitions does not exact a price in inefficient code. If anything, the facility allows for far better code generation. It was explained in section 2 that in compiling a routine R1, one could freeze the values of free variables to their then current values. Suppose we execute

$$M \leftarrow P\_ARRAY(\mathcal{E}_1, \mathcal{E}_2)$$

M then assumes some definite mode value. If R1 uses M as a type declarer, leaving M as a free variable, then R1 may be compiled with M frozen. The specific mode value of M will be used. It is therefore possible and practical for a program to compute the modes it uses and compile parts of itself specific to these computations.

Section 6: Generic Routines

The main reason for having united modes (e.g., ANY and those generated by ONEOF) is to type formal parameters for routines which accept several distinct types of arguments. Such routines (e.g., the operator + in Algol 60) do not convert these arguments to fixed types but rather perform different actions dependent on argument types. Such routines are termed generic. Almost all languages have such routines, but almost always as built-in operations. In this section we discuss how the programmer defines his own generic routines in EL1.

The basics have already been discussed: one needs the ability to declare formal parameters having united modes and a means of testing the actual modes of these parameters. The latter is provided by the primitive routine MD; for any expression $\mathcal{E}$, MD($\mathcal{E}$) is the mode of the value of $\mathcal{E}$. To illustrate a possible, if far from satisfactory technique, consider defining + to act on INTs, REALs, and COMPLEXs.

```
SCALAR ← ONEOF(INT, REAL, COMPLEX);
+ ← EXPR(X:SCALAR, Y:SCALAR; SCALAR)
      BEGIN
         (MD(X)=INT) AND (MD(Y)=INT) ⇒ FIXADD(X,Y);
         (MD(X)=COMPLEX) AND (MD(Y)=COMPLEX) ⇒
               CONST(COMPLEX OF X.RE+Y.RE, X.IM+Y.IM);
         (MD(X)=INT) AND (MD(Y)=REAL) ⇒
               FLOATADD(FLOAT(X),Y);
         etc.
      END
```

The + routine is here declared to take two arguments — each of which may be one of {INT, REAL, COMPLEX}. The routine body tests the types of its arguments on each invocation and dispatches to the appropriate code section.

This has two principal defects: (1) The type testings and their conjunctions are redundant and hence tedious to read and write. (2) It is difficult for the compiler to exploit knowledge it may have concerning the modes of arguments. For example, if A has been declared to be an INT, then A+3.2 will invoke the third alternative, but how is the compiler to know this? It could, of course, make the deduction by "interpreting" the + routine. While this will work in principle, it seems an unnecessarily difficult approach. Instead, we impose additional structure on the program — structure which simultaneously makes the code more readable by man and more comprehensible by the compiler.

The traditional means for imposing structure in a programming language is with a new syntactic form, here the GENERIC form. A GENERIC form[†] is

[†]A GENERIC form may appear anywhere within a routine; the left-hand arm of each statement is always compared with the arguments to the routine itself. This is useful for routines which take one or more generic arguments, but

delimited by the brackets "GENERIC" and "END" and contains a set of conditional-like statements whose left arm is a set of modes and whose right arm is an alternative value of the generic form. For example, the above + routine may be directly recoded using a GENERIC form as its body

```
+ ← EXPR(X:SCALAR, Y:SCALAR; SCALAR)
    GENERIC
         [INT, INT] ⇒ FIXADD(X, Y);
         [COMPLEX, COMPLEX] ⇒ CONST(COMPLEX OF X.RE+
               Y.RE, X.IM+Y.IM);
         [INT, REAL] ⇒ FLOATADD(FLOAT(X), Y);
         etc.
    END
```

This may be read as an analogue to a set of conditionals:

<u>if</u> the 1st arg is an INT and the 2nd arg is an INT <u>then</u> FIXADD(X, Y);

<u>else</u> <u>if</u> the 1st arg is a COMPLEX and the 2nd arg is a COMPLEX <u>then</u>
construct a COMPLEX of X.RE+Y.RE and X.IM+Y.IM;

<u>else</u> <u>if</u> . . . <u>then</u> . . .

.
.
.

The alternatives are considered in turn until one is found which matches the actual modes of the arguments on this invocation of +. The last statement can optionally be of the form

ELSE ⟨expression⟩

If this is present and none of the alternative sets match, the last ⟨expression⟩ is taken as the value of the GENERIC; if this is not present and there is no match, a system error routine is called. The importance of the GENERIC to compilation is, of course, that "considering the alternatives" can usually be carried out during compilation so that compiled calls on + can usually be replaced by a call on the right-hand side of the appropriate alternative statement.

---

contain substantial computation $\mathcal{E}$ which does not depend on the modes of these arguments (e.g., computation based on the non-united arguments). Such routines can be written with an <u>embedded</u> GENERIC form. The code for $\mathcal{E}$ is then effectively shared among the various generic alternatives.

It should be obvious that if there are several generic formals, the number of possible combinations can grow to unwieldy size. Even with a concise notation for expressing alternatives, this is unacceptable. It need not, however, arise. An element of the mode set of a GENERIC statement can be an arbitrary syntactic form, so long as the value it produces is a mode. Hence, a GENERIC statement such as

[CHAR, ONEOF(INT, STRING), ANY] ⇒ ...

will cover (i.e., match against) each of the following sets of argument modes

{CHAR, INT, REAL}, {CHAR, STRING, INT}, {CHAR, STRING, MUMBLE}, ...

In general, a mode G in a mode set covers an argument mode A if any of the following hold:

(1) G = ANY

(2) G is a generic mode ONEOF($t_1 \ldots t_n$) and A = t_i for some i

(3) G = A

Even collections of modes will, in some cases, prove too restrictive in performing generic selection. Consider, for example, a print routine which takes a single argument X — an object to be printed

EXPR(X:ANY; NONE) ...

X can be of ANY mode and no result is returned. The routine is to print X in one of three formats, depending on whether X is a structure, an array, or a pointer. Testing this is straightforward: EL1 includes a primitive routine STRUCTP which is a predicate true of structures only. Hence

STRUCTP(X) ⇒ ...

is the test which checks for the first print class. The trick is to make the discrimination while compiling a call on the print routine.

This can be done if the left-hand side of a generic statement is generalized to

$[\mathscr{M}_1, \ldots, \mathscr{M}_N]\,\mathscr{P}$

where the $\mathscr{M}$s are forms which evaluate to modes to be compared to the argument modes and $\mathscr{P}$ is an arbitrary form producing a Boolean value. An alternative is chosen only if all modes match and $\mathscr{P}$ is true.[†] For example, the desired print routine has the structure

```
EXPR(X:ANY; NONE)
GENERIC
    STRUCTP(X) ⇒ . . . ;
    ARRAYP(X)  ⇒ . . . ;
    ELSE . . .
END
```

Any predicate whatever can be used as part of a generic alternative. This provides a very general mechanism for the programmer to control compilation, i.e., to perform once at compile time a choice which would otherwise be made repeatedly during execution. Hence, the generalized GENERIC form is quite powerful. The facility it provides is related to the freezing of free variables during compilation. The difference is this. Freezing allows one to produce individual compilations of a routine, each tailored to some specific environment; the GENERIC form allows a single routine to take several alternative actions, yet allows choice among the alternatives to be made when compiling a call on the routine.

The compiler gives special treatment to GENERICs under two circumstances: (1) in compiling a routine whose body is a single GENERIC form, and (2) in compiling another routine which contains one or more calls on routines of type 1. We consider these in turn.

When given a routine such as + whose body is a single GENERIC form, the compiler produces:

[†] $\mathscr{P}$ may be absent, in which case it is taken as TRUE. Similarly, any of the $\mathscr{M}$s can be absent, in which case they are taken as ANY. Note that the mode set is semantically unnecessary since all mode checks could be carried out in the predicate. However, factoring the selection into two parts — a simple pattern match and an arbitrary predicate — is useful for pragmatic reasons.

(1) a main body for the routine,

(2) a set of alternative bodies — one for each alternative GENERIC statement.

The main body consists of

(1.1) a flag indicating that this is the main body of a GENERIC,

(1.2) executable code,

(1.3) a table of alternative mode/predicate sets,

(1.4) an array of pointers to the alternative bodies.

The main body has the original formal mode set, e.g., for the + routine this is (SCALAR, SCALAR; SCALAR). It can be called directly (e.g., from interpreted code), in which case it tests the alternative mode/predicate sets against the arguments and dispatches to the appropriate alternative body. That is, calling the main body simply invokes type testing during execution. The alternative bodies are themselves complete code blocks[†] which can be called directly.

Consider next the actions of the compiler on another routine, say FOO, which contains a call on a generic routine (e.g., the + routine in X+FUM(Z)). Since the + body is flagged as being GENERIC, an attempt is made to discover which alternative would be chosen *were* the decision deferred until run-time. There are two possibilities: (1) Some alternative is chosen, say the ith, in which case the compiler generates a call directly on the ith body. (2) The

---

[†]Mode information derived from the generic alternatives can be used to considerable advantage in compiling these bodies. For example, consider

```
        EXPR(X:ANY, Y:ONEOF(BOOL, CHAR, STRING); REAL)
        GENERIC
             [INT, BOOL] ⇒ 𝔉₁;
             [ONEOF(REAL, CHAR), STRING] ⇒ 𝔉₂;
                                      .
                                       .
                                        .
        END
```

The first alternative, $\mathfrak{F}_1$, has the formal modes (INT, BOOL; REAL) and is compiled under the assumption that X is an INT and Y is a BOOL. Similarly, $\mathfrak{F}_2$ has the formal modes (ONEOF(REAL, CHAR), STRING; REAL) and is compiled accordingly. It may be that $\mathfrak{F}_1$ and $\mathfrak{F}_2$ are textually identical, yet the different assumptions of argument modes will lead to different compiled code, each block being tailored to its formal modes.

compiler discovers that it cannot make a choice, in which case it generates a call on the main body. Which case applies is determined by the modes of the arguments and the alternative mode/predicate sets in the GENERIC.

Consider first the modes of the arguments. When the + routine is called, its two arguments will have some definite (i.e., non-generic) mode. However, the compiler has access only to declarative information and from this must deduce what we shall term _compilation modes_. In some cases, these will be less precise than the actual argument mode. For example, a formal parameter which is ONEOF(INT, REAL) has a united compilation mode. Similarly, a block such as

```
        BEGIN  P(X) ⇒ "IN";   0.  END
```

returns either a SYMBOL or a REAL and hence has compilation mode ONEOF(SYMBOL, REAL).

Since the _compilation mode_ of an argument may be united, the generic selection mechanism must in general be prepared to take a united argument mode. Hence, the definition of _covers_ given above must be expanded to include this case. In general, a mode G in a mode set _covers_ an argument mode A if any of the following hold:

(1) G = ANY

(2) G is a generic mode ONEOF(t_1 ... t_n) and A=t_i for some i

(3) G = A

(4) G and A are both generic and each alternative of A is an alternative of G (i.e., G $\supseteq$ A).

The fourth clause raises the possibility of the compiler deciding that it cannot make a compile-time choice. If G is ONEOF(t_1 ... t_n) and A is ONEOF($\hat{t}_1$ ... $\hat{t}_m$), it may be that G does not cover A but G covers one or more of the $\hat{t}$'s. We say that G _partially covers_ A. If the _actual_ mode is one of the t's in G, then G will cover the actual mode and the alternative may be chosen, otherwise it will not.

The compiler cannot tell which will be the case and hence must postpone generic selection until execution of the function call.

Given a set of compilation modes ($\mathscr{M}_1 \ldots \mathscr{M}_n$) for the arguments and a table of alternative mode/predicate sets each of the form $[\mathfrak{F}_{i1} \ldots \mathfrak{F}_{in}]\mathscr{P}_i$, choosing the appropriate GENERIC alternative proceeds as follows. The alternative sets are considered in turn, starting with the first. If each formal mode $\mathfrak{F}_{ij}$ <u>covers</u> $\mathscr{M}_j$ and if $\mathscr{P}_i$ is evaluatable and true, then the i^{th} alternative is chosen. If for some j, $\mathfrak{F}_{ij}$ <u>partially</u> <u>covers</u> $\mathscr{M}_j$, or if $\mathscr{P}_i$ is not evaluatable, then generic selection cannot be made during compilation. Otherwise, the next alternative set is considered.

A key point here is determining whether a predicate is evaluatable. This is handled by a variation of the technique for free variable freezing discussed in section 2. The compiler treats specially those primitive routines whose value might be known (e.g., STRUCTP, LENGTH, MD). Each such routine has a set of enabling conditions which depend only on data known during compilation. When these are satisfied, values can be calculated and substituted into the computation tree of the predicate. Whenever a routine and its arguments in the computation tree of the predicate are thus fixed, the routine is applied, propagating the values upward. The compiler applies this process wherever possible until all possible upward propagation has been carried out. If the entire computation tree of the predicate collapses into a single value, then the predicate is evaluatable and its value is known; otherwise, the predicate is not evaluatable.

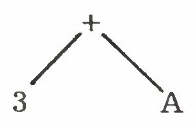
<u>Section 7: Type Conversion</u>

The most important single point concerning type conversion is that it is different from the generic mechanism. The two concepts are, in fact, almost orthogonal. Both are mechanisms which allow a routine to be called with arguments belonging to a set of possible modes, but here the similarity stops. With the generic mechanism, the routine has a corresponding *set* of possible parameter modes. With the type conversion mechanism, the routine has a *single* parameter mode and values of different modes are converted to that mode.

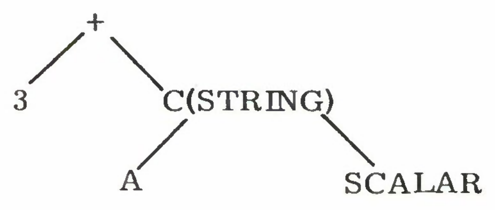
Traditionally, the choice of conversion routines to be used is fixed for all time by the language designer. Even in traditional languages, there is little reason for this early freezing. Where it is possible for the programmer to define new types, it becomes essential that he be permitted to specify the associated conversions.

Hence, the treatment of type conversion in EL1 is designed to satisfy two goals: (1) smoothly meshing type conversion with the generic mechanism and (2) allowing the programmer to specify what the type conversion will be. In outline, the technique used in EL1 is as follows. First a test is made to see if the formal mode *covers* the argument mode (i.e., either the modes are equal or the formal is generic and one of its alternatives is equal to the argument mode). Failing this, the argument is converted to a value belonging to the formal mode using a type conversion routine associated with the argument mode.

For example, suppose that the + routine has been defined on pairs of SCALARS (defined as ONEOF(COMPLEX, REAL, INT)) and that + is called with a STRING-valued argument A. In computation tree terms, we have the direct tree for the computation, say 3+A



Since the formal mode of the second parameter (SCALAR) does not cover the argument mode (STRING), the direct tree is replaced with the implicit tree



where C(STRING) is the conversion routine associated with string. As with all conversion routines, this takes two arguments: the object to be converted and the desired mode of the converted result. Here, the desired mode is the formal mode of the second parameter – SCALAR.

The association of a conversion routine with a mode is performed by an assignment. A mode such as STRING can be treated as a structure having a set of components, one of which is named CONVERT. Assignment of a routine to the CONVERT field of a mode establishes that routine as the conversion function for the mode. For example, a possible conversion routine for STRING could be set up by:

```
STRING.CONVERT ←
        EXPR(X:ANY, FM:MODE; ANY)
        GENERIC
            FM COVERS INT ⇒ STRING_TO_INT(X);
            FM COVERS SYMBOL ⇒ HASH(X̄);
                        .
                          .
                            .
        END
```

Like most conversion routines, this is a GENERIC. The first alternative consists of a single predicate which uses the infix operator COVERS. This is true if its left-hand operand is a mode which covers its right-hand operand. Hence, the first GENERIC is chosen whenever the desired mode includes INT. This is the case in the above call on +, so STRING_TO_INT is applied to A. The result, an integer, is taken as the actual argument to the + routine. As with other generics, the significant point is choosing the appropriate alternative of STRING.CONVERT when compiling the call on +. We address this pragmatic question after examining a few semantic issues.

In so doing, a summary of the discussion thus far may be of use. During evaluation, the interpreter may have in hand some actual value of $\mathscr{A}$ of mode $\mathscr{M}_A$ and some desired formal mode $\mathscr{M}_F$ such that $\mathscr{M}_F$ does not cover $\mathscr{M}_A$. The programmer can arrange that in all such circumstances

$\mathscr{A}$ be converted by applying a routine F by associating with $\mathscr{M}_A$ a GENERIC conversion routine having the form

```
EXPR(X:ANY, FM:MODE; ANY)
GENERIC
     FM COVERS 𝓜_F ⇒ F(X);
               .
                .
                 .
END
```

This allows point-to-point conversion between any actual value and any target mode (which can, if desired, be generic).

One could apply this schema to all ⟨source, destination⟩ pairs for which conversion was desired. However, even using generic destination modes to cut down on the number of distinct destinations, the number of pairs could be undesirably large. Further, when defining a new mode $\mathscr{M}_A$, it would be necessary to explicitly define each desired conversion to an existing mode. Again, the number of such conversions may be undesirably large.

In the EL1 framework, such an exhaustive enumeration of point-to-point conversions can be avoided by judicious use of functional composition. Consider, for example, the conversion from a CHAR argument to a REAL formal mode. The conversion CHAR → REAL is almost surely the composition of CHAR → INT and INT → REAL. Similarly, COMPLEX → INT is surely COMPLEX → REAL followed by REAL → INT. This suggests an analogous functional composition of conversion routines. For example, the conversion routine for CHAR might be written

```
EXPR(X:ANY, FM:MODE; ANY)
GENERIC
     FM COVERS STRING ⇒ CONST(STRING OF X);
     FM COVERS INT ⇒ CHAR_CODE(X);
     (FM COVERS REAL) OR (FM COVERS COMPLEX) ⇒
          INT.CONVERT(CHAR_CODE(X), FM);
                          .
                           .
                            .
END
```

The third alternative tests whether either REAL or COMPLEX is acceptable; if so, X is converted to an INT and the INT conversion routine is called to complete the work. The test in this case is used to insure that the process does not run into a dead end.

In general, specifying conversion by composition allows the programmer to factor the conversion bush stemming from a data type. This does not address the question of what the paths should be when there is more than one path logically possible. An answer to the question can only come from a knowledge of what the data types represent; i.e., a decision must be based on the specific application. The point of the factoring scheme is to provide a concise notation for expressing the desired paths, once a decision has been made.

Thus far the discussion has centered around a hypothetical evaluator with an actual argument requiring conversion in hand. That is, we have described the actions of the interpreter and neglected the compiler. In explaining the semantics of type conversion, this benign neglect is actually quite appropriate. One general rule of EL1 semantics is that the evaluator model dictates actions in complex circumstances; the compiler is constrained to produce code that does the same thing. Applying the general rule to type conversion results in the following dictum: the compiler must generate code for type conversions which has results identical to those which would have been obtained using the interpreter.

To take a concrete example, consider a function call

FOO(MUMBLE(X))

where the formal parameter of FOO has mode $\mathcal{M}_F$ and the formal result type of the MUMBLE routine is $\mathcal{M}_R$. There are three cases:

(1) $\mathcal{M}_F$ covers $\mathcal{M}_R$

(2) $\mathcal{M}_F$ does not cover $\mathcal{M}_R$ and $\mathcal{M}_R$ is non-generic

(3) $\mathcal{M}_F$ does not cover $\mathcal{M}_R$ and $\mathcal{M}_R$ is generic.

In case 1, no conversion is required.

In case 2, the function call is treated as if the program had read

FOO($\mathcal{M}_R$.CONVERT(MUMBLE(X), $\mathcal{M}_F$))

Most likely $\mathscr{M}_R$.CONVERT is GENERIC so that the compile-time GENERIC selection mechanism is invoked to choose the appropriate alternative. There is one new point here: one or more of the predicates may contain expressions of the form

FM COVERS $\mathscr{M}_i$

where FM is the second formal parameter — the desired mode. To make a compile-time selection here, the compiler must handle generic conversions somewhat specially and recognize that in such cases it knows the value of FM so that the predicate may be evaluatable. This is related to a point mentioned earlier: the compiler must recognize that it may know the value of functions such as STRUCTP and LENGTH appearing in predicates of other GENERICs. The compiler has, of course, considerable specific knowledge of this sort. Once an alternative code body is chosen, compiling the call is straightforward. Here, as with any other call, there is a choice to be made between generating code to call the existing code block and generating an in-line expansion. The choice is best made by the compiler, based on the size of the code to be copied and the setting of parameters which control the space/time trade-off.

Case 3 is somewhat subtle. It is known that the <u>actual</u> result mode, $\mathscr{M}_A$, will be one of the alternatives of $\mathscr{M}_R$. Either (I) $\mathscr{M}_F$ covers $\mathscr{M}_A$, in which case no type conversion is required, or (II) it does not, in which case the conversion routine $\mathscr{M}_A$.CONVERT must be invoked. However, the compiler has no way of determining what $\mathscr{M}_A$ will be. Hence, it generates code which tests the mode of MUMBLE(X) at run time and invokes the conversion routine for that mode if II holds. The use of run time type testing here is vital. It would <u>not</u> be acceptable to interpret case 3 as FOO($\mathscr{M}_R$.CONVERT(MUMBLE(X))). The generic mode $\mathscr{M}_R$ could call for conversions quite different from those invoked by the <u>actual</u> result mode; compiled code could then produce quite different results from the same program run interpretively.

The case of a generic compilation mode $\mathscr{M}_R$ which is not covered by a formal mode $\mathscr{M}_F$ is not confined to the values of routines. Similar situations are produced by blocks, variables, etc. – wherever the compiler cannot completely determine the data type of a construct. In the important case of blocks, the different data types frequently arise from different block exits. Hence, the compiler can distribute the type conversion in space so as to use the efficient case 2 treatment. For example, suppose that FOO is called with the argument

```
BEGIN
   P(X) ⇒ Y;
   ELSE J
END
```

where MD(Y)=REAL, MD(J)=INT, so that the compilation mode $\mathscr{M}_R$ of the block is ONEOF(REAL, INT). Suppose $\mathscr{M}_F$ does not cover this. A commonly used technique in language design is to "widen" the result of the block to REAL and compile in code for REAL to $\mathscr{M}_F$ conversion. This is poor design, since the technique can result in unnecessary conversion steps. The solution used in the EL1 compiler is to treat each statement that can lead out of a block independently. Hence, the block is compiled as if it had been

```
BEGIN
   P(X) ⇒  REAL.CONVERT(Y, 𝓜_F);
   ELSE    INT.CONVERT(J, 𝓜_F)
END
```

This is better, in both time and space, than forcing an artificial widening. It is particularly attractive when $\mathscr{M}_F$ covers either REAL or INT so that the appropriate conversion routine is omitted.

Section 8: Programmer Specified Mode Behavior

The treatment of mode definition given in section 4 centered on the construction of modes. That is, a mode WALDO can be defined to be a set of objects having fields A, B, C of types t_1, t_2, t_3, etc. Such a syntactic specification is, however, only one aspect of mode definition. Indeed, at a sufficiently high level of abstraction,[†] a level which is often only implicit in programming, the syntactic specification is irrelevant; what is of interest is how an object behaves. In this view of programming, a syntactic mode specification is a lower level concept which serves to implement some higher level set of behavioral laws. The definition mechanism of section 4 is then a necessary prerequisite, but only as a basis on which to build a sophisticated mode definition mechanism. Given that this is the direction to be taken in providing a truly problem-oriented language, the issue is what constitutes a higher level mode definition and how to state such a definition in a convenient way.

To some extent, section 6 has shown the approach to be taken. Consider two modes M1 and M2, defined

```
M1 ← SI::ARRAY(16,BOOL);
M2 ← MANT::ARRAY(16,BOOL);
```

This introduces a sixth primitive mode constructor denoted by the infix operator "::". This takes a variable name as its left-hand operand and a mode as its right-hand operand and constructs a new labeled mode distinct from all modes which may be structurally similar but have different or no labels. In the above example, SI and MANT serve as labels for their respective modes, so that the two modes are not equal and neither is equal to ARRAY(16,BOOL). Since M1 and M2 are different modes, they can be assigned different conversion routines, say C1 and C2, respectively. Suppose X1 and X2 are variables of modes M1 and M2, respectively. Structurally, they are identical. However, if used in a position where conversion is required (e.g., 3+X1 or 3+X2), they may act quite differently. To take a simple example, SI::ARRAY(16,BOOL) may represent a signed integer whose magnitude is less than 2^{15} using 16 bits in two's complement notation, while MANT::ARRAY(16,BOOL) may represent a real number

[†]Such layers of abstraction are directly related to the strata of Dijkstra's structured programming [9]. We pursue this point later.

between 1 and 0. These facts about representation are stored in the routines that handle conversion from these modes to other modes (here, SCALAR). The algorithm which uses X1 and X2 itself displays none of these representational issues. It performs the abstract operation of addition and the data type definitions of X1 and X2 determine the rest. It should be noted that two mechanisms are employed — the implicit type conversion for the arguments to the + routine and the GENERIC mechanism in the + routine itself. For the purpose of this discussion, the former is the more important since it is more global in scope — the conversion routine for M1 will be applied in <u>any</u> situation where a value of mode M1 causes a type fault.

This separation of abstract process-oriented algorithm from detailed mode-dependent manipulations is clearly a step in the right direction. To push this further, we need only find other "global" situations in which mode-specific manipulations should be called into play. Two[†] others have been chosen for consideration in EL1 — selection and assignment. To pursue the above example, programmer control over the meaning of assignment would allow one to specify that

$$X1 \leftarrow -34.2$$

is to cause the real value to be converted to an integer and that value packed into 16 bits.

A second example may be useful to illustrate the power of this technique. For debugging and other purposes, it is frequently useful to be able to monitor the value of a variable and take some special action (e.g., output of an error message) under certain abnormal conditions (e.g., when its value exceeds the value of another variable). Let X be such a variable of mode M, let P(X) be a predicate which tests for abnormal conditions, and let A(X) be the action to be taken.

[†]It would be easy to justify additional ones, such as generation or storage reclamation. Some experience will perhaps be required before a completely satisfactory set is found.

One can define a new mode M' in terms of M, P, and A as follows:

(1) M's are structurally identical to Ms.

(2) Whenever a formal mode M is required and an actual value of mode M' is in hand, the M' is treated as if it were an M.

(3) Whenever an object of mode M' is assigned a new value X, the predicate P(X) is evaluated; if the result is true, A(X) is executed.

One can go further and automate this process by defining a routine SENSITIVE_MODE (as a function of one mode M and two routines P and A) that constructs the new mode M'. Having written this one routine, the programmer has at his disposal the notion of "sensitive object" for any mode M. Redeclaring any variable to be a SENSITIVE_MODE(M, P, A) inserts the monitoring probes with no other changes to the program required.

To provide some substance to the discussion, we turn to a third example which we treat in detail. Consider defining the mode "ring buffer of characters." If X is such a buffer, its chief characteristics are:

(1) An assignment of a CHAR value to X pushes the character onto the back end of the buffer if there is room, else error.

(2) Use of X where formal mode CHAR is desired pops a character from the front end of the buffer if the buffer is non-empty, else error.

Very likely, other properties would be desirable:

(3) The buffer can be treated as if it maintained at all times a correct count of the number of characters it holds and X.COUNT accesses this component.

(4) The top element of the buffer can be inspected without popping it by selecting the TOP element, i.e., X.TOP.

A possible structure for such buffers is

```
STRUCT(FRONT:INT, BACK:INT, BODY:ARRAY(K, CHAR));
```

where K is some constant — the maximum number of characters the buffer is to hold. (To simplify the discussion, we suppose a given value for K, say 200.) FRONT and BACK will be indices to the front and back ends of the buffer, with the convention that characters go in the back end and out the front end. Hence, FRONT chases BACK backward around the ring with modulo K arithmetic.

To establish the desired behavior, it will be necessary to use rather special assignment, selection, and conversion functions. We do not want these applied to arbitrary objects that happen to have the above structure; hence, in actually defining the desired mode, we use a label to create a unique mode

RBUFF ← RB: :STRUCT(FRONT:INT, BACK:INT, BODY:ARRAY(K, CHAR));

RBUFF is a mode-valued variable. Its value is defined (by the assignment) to be RB: :STRUCT(FRONT:INT, BACK:INT, BODY:ARRAY(200, CHAR)) which differs from other modes having identical structure but different (or no) label.

The desired behavior of RBUFF is established by assignment to the SELECT, ASSIGN, and CONVERT fields of this mode. We consider these in turn. We have established that if X is an RBUFF, there are to be two and only two "fields" which may be selected:

X.COUNT	which gives the number of items in the buffer
X.TOP	which gives the top item of the buffer.

That these fields do not actually exist as such is irrelevant, so long as the mode definition creates the desired behavior (illusion if you will). Further, when using an RBUFF as an RBUFF, there is no need to directly access the fields FRONT, BACK, and BODY. The job of the selection routine is to define the desired fields "COUNT" and "TOP" in terms of the fields which actually exist and simultaneously render these latter fields unavailable to direct access. The language evaluator provides a triggering mechanism for this definition. It calls the selection routine[†] of RBUFF on any selection of the form

X.⟨fieldname⟩

[†]If no explicit definition is made, a system-generated selection function is used. It is this that establishes the "normal" meaning of selection for a defined mode.

where X is of mode RBUFF. It will be passed two arguments – the object being selected from and the name of the field represented as a symbol. Consider

```
    RBUFF.SELECT ←
            EXPR(X:RBUFF, FD:SYMBOL; INT)
            GENERIC
              FD="TOP"    ⇒ BEGIN
                               UR(X).FRONT≠UR(X).BACK ⇒
                                  UR(X).BODY[UR(X).FRONT]
                               ELSE BUFF_EMPTY(X);
                            END
              FD="COUNT" ⇒ BEGIN
                               DECL F:INT BYVAL UR(X).FRONT;
                               DECL B:INT BYVAL UR(X).BACK;
                               F>B   ⇒   F-B;
                               F<B   ⇒   200-B+F;
                               ELSE  ∅
                            END
              ELSE SELECTION_FAULT(RBUFF, FD)
            END
```

The routine tests the field name by comparing it to the symbol-valued constants "TOP" and "COUNT." Based on this comparison, the main conditional discriminates between three main cases: (1) the ⟨fieldname⟩ is TOP, (2) the ⟨fieldname⟩ is COUNT, and (3) neither of the above. The last case is treated as an error and a system error routine is called. Consider the case FD="TOP." We adopt the convention that FRONT is the index of the first good element to be emptied on output and BACK is the index of the next element to be filled on input; hence the buffer is non-empty whenever FRONT≠BACK. However, it is <u>not</u> possible to make the required test by writing

X.FRONT≠X.BACK

Since X is an RBUFF, this would invoke the selection routine for RBUFF recursively. What we need is the selection routine not for RBUFF but rather for the underlying representation STRUCT(FRONT:INT, BACK:INT, BODY:ARRAY(200, CHAR)). The primitive routine UR maps X onto an object having the same pattern of bit values but a different mode, the mode of the <u>underlying representation</u>. In fact, no copying need be done: X and UR(X) refer to the same object, they just ascribe <u>different</u> modes to this object. With this explanation of UR, the rest of the code should be fairly clear: if the buffer is not empty, its top element is selected. As to the case F="COUNT", the block uses two local variables simply to avoid writing UR(X).FRONT and UR(X).BACK repeatedly; the number of characters is calculated in the obvious way.

Consider next the conversion routine for RBUFF. Assuming for the sake of simplicity that the only conversion to be considered is RBUFF → CHAR, we obtain

```
RBUFF.CONVERT ←
        EXPR(X:RBUFF, FM:MODE; CHAR)
        BEGIN
            DECL F:INT BYVAL UR(X).FRONT;
            DECL TEMP:CHAR;
            FM≠CHAR ⇒ TYPE_FAULT(RBUFF,FM);
            F = UR(X).BACK ⇒ BUFF_EMPTY(X);
            TEMP ← UR(X).BODY[F];
            F ← UR(X).FRONT ← F-1;
            F=0 → UR(X).FRONT←200;
            TEMP
        END
```

This creates a local variable F initialized to the value of UR(X).FRONT and a TEMP of mode CHAR to hold the result of the routine. The next two lines test that the desired mode is CHAR and that the buffer is not empty. Then TEMP is assigned the top element and UR(X).FRONT is decremented. If the new value is zero, UR(X).FRONT is wrapped around the buffer. Finally, the block (and hence routine) returns TEMP as its value.

The assignment routine is similar and should be self-explanatory:

```
RBUFF.ASSIGN ←
        EXPR(X:RBUFF, Y:CHAR; CHAR)
        BEGIN
            UR(X).BODY[UR(X).BACK] ← Y;
            UR(X).BACK ← UR(X).BACK-1;
            UR(X).BACK=0 → UR(X).BACK←200;
            UR(X).BACK≠UR(X).FRONT ⇒ Y;
            ELSE BUFF_OVERFLOW(X)
        END
```

One point should be noted. Suppose B1 and B2 are both RBUFFs and consider the assignment

```
B1 ← B2;
```

Since the left-hand operand is an RBUFF, the RBUFF assignment routine is called. Binding its formals to its arguments proceeds as follows. The formal X is an RBUFF so this is bound directly to B1. However, the formal Y is a CHAR and the argument B2 is an RBUFF. Hence, the conversion routine for RBUFFs is called with arguments B2 and CHAR. The result is to pop an element from B2 and bind the formal Y to this. Hence, the assignment causes an element to be popped from the front of B2 and added to the back of B1.

Turning from the specific to the general, several points should be noted. (1) It is straightforward to treat the buffer size K and the mode M of its constituent elements as parameters and write a routine BUFFER(K, M) that produces a mode for any values of K and M. Such a BUFFER routine can be viewed as a realization of one implementation technique of the concept of buffer. From another point of view, one can ignore the implementation and take BUFFER as an abstract set of data types with certain properties. (2) The notion of underlying representation has a natural extension. We have just used the mode STRUCT(FRONT:INT, BACK:INT, BODY:ARRAY(200, CHAR)) as a basis for defining RBUFF so that the former is the underlying representation of the latter; we could equally well use an RBUFF as a basis for defining a new mode, for which RBUFF would be the underlying representation.

We illustrate this notion with an example. A character stream is frequently used to encode a virtual character set greater than that actually available by using one or more characters as escape characters whose appearance changes the interpretation of characters which follow them. In such cases, a virtual character is an array of actual characters. It is then useful to consider a class of buffers into which single CHARs can be pushed at the back end but which deliver STRING's at the front end. Let this mode be called STRBUF. One could in principle define STRBUF in terms of the underlying representation STRUCT(FRONT:INT, BACK:INT, BODY:ARRAY(200, CHAR)). However, it is far more convenient to use RBUFF as the underlying representation. Then STRBUF can be defined in terms of COUNT, TOP, and the operations of assigning to and converting from RBUFFs. In this definition, if X is an STRBUF, then UR(X) is to be interpreted as an RBUFF. In this fashion, any existing data type can be used as the basis for defining a new type.

The syntax for carrying this out is simple since the labeling of modes can be cascaded. If $\mathscr{M}$ is a mode, and $\mathscr{N}$ is an identifier, then

$$\mathscr{N} :: \mathscr{M}$$

is a new mode with label $\mathscr{N}$ based on the mode $\mathscr{M}$. For example,

WALDO::RBUFF

defines a new mode equal to WALDO::RB::STRUCT(FRONT:INT, BACK:INT, BODY:ARRAY(200, CHAR)). If X is an instance of this mode, UR(X) is an RBUFF. This new mode can itself be used as the second argument to the :: operator to build up a hierarchy of mode definitions. Such a definition scheme has a number of consequences closely related to Dijkstra's structured programming.

In writing a program in the style of structured programming, one builds a "string of pearls." Each pearl has its own set of abstract operations and data types defined in terms of lower level pearls. Realizing this for operations is straightforward; higher level routines are composed from lower level routines. Programmer control over mode behavior, as discussed in this section, provides an analogue for data. A mode $\mathscr{M}_i$ at one level in the string can be _based on_ one or more modes at lower levels. A well-engineered definition set will use the UR routine _only_ in the mode specification routines (selection, conversion, and assignment). Operations at that level see _only_ the behavior of the defined mode $\mathscr{M}_i$, not its definitions in terms of lower level notions. Hence, the actual representation used to achieve this behavior is irrelevant at this and higher levels. Without changing other parts of the program, one can vary this representation at will. This affords a very powerful means to (1) modify the program to perform related tasks, and (2) optimize performance for a given task.

Section 9: Conclusion

The treatment of data types in EL1 rests on nine points:

(1) modes as values in the language,

(2) the facility for freezing free variables during compilation,

(3) the generic interpretation of mode union,

(4) routines as a unification of operators and procedures,

(5) inclusion of both generic routines and type conversion,

(6) programmer-defined generic routines and programmer control over type conversions,

(7) interpreter-based semantics for generic selection and type conversion which the compiler is constrained to follow,

(8) programmer control over mode behavior,

(9) the notion of underlying representation and the basing of one mode on the behavior of another.

Some of these are independent of one another and of the EL1 language; these can be applied directly to other languages. Other points depend strongly on the language; carrying these over is a bit tricky. However, none of these is particularly difficult to implement.

The most radical points are the first and second. These are also the most significant. The inclusion of modes among the legitimate values in a language allows modes to be computed, providing a very powerful definitional capability. A direct consequence is the concept of programmer-defined, mode-valued routines and, hence, the functional abstraction these provide.

The first point demands the second (or a functional equivalent). Computed mode values are of interest only if they can be used as the types of variables in compiled routines. Hence, there must be some mechanism to specify that a particular non-primitive mode value is to be used as a data type. The specific technique used in EL1, the "freeze list," is somewhat immaterial. Other equivalent techniques could be used to the same effect. What is important is the concept of evaluability and the upward propagation of computation-tree collapse. The utility of this mechanism goes beyond its use in connection with data types. It allows the programmer to nail down invariants of all sorts and have the program reflect the consequences of these invariants.

Turning to point 3, we note that there are many possible interpretations of "union" as applied to data types. The one used in EL1 was chosen on the basis of simplicity, implementability, and because it meshes most smoothly with the generic routines. This interpretation of union — the generic — treats a united mode as the postponement of a commitment until execution. Hence, during execution the concept largely disappears, simplifying the semantic description.

The use of routines as a unification of operators and procedures hardly requires comment. Apart from external syntax, there is no real difference between the two. A language which allows the programmer to define both should surely provide identical semantics for the two. The alternative is harder to communicate, learn, and implement. Regrettably, it is the common practice.

Having both generic routines and type conversion is almost a necessity. Neither alone provides the right flexibility; neither is a good substitute for the other. The scheme used in EL1 may be briefly summarized as: if a formal mode cov .he argument mode, binding is direct; otherwise, the argument is converted to some mode that is covered.

Given that a language includes generic routines and type conversion, it should follow that these be controllable by the programmer, i.e., that he be permitted to define generic routines (in addition to the built-in set) and that he be permitted to specify the type conversions. This is not hard to implement. Most of the necessary mechanisms are already present to handle the built-in definitions. Implementation of programmer control is mostly a matter of employing these mechanisms on programmer-supplied definitions.

The only subtle point is choosing a semantic model. Generic selection and type conversion can become complex, since conversions are typically cascaded and often applied to the arguments of generic routines. A compiler model is awkward, since it either imposes language restrictions to insure compile-time knowledge of modes or, lacking this, produces a description which depends not on modes but the compiler's knowledge of modes. The use of an interpreter (i.e., run-time) model greatly simplifies the descriptive task.

Giving the programmer control over the behavior of the modes he defines is again easy to implement, provided the implementation is done correctly. Any system which allows the definition of new data types must construct tables or their equivalent to give meaning to subsequent assignments, selections, type conversions, etc., using variables of these types. It is a short step and a considerable improvement from such tables to system-generated routines tailored to each new data type. The next step is to allow the programmer to specify the routines <u>he</u> wants invoked. His routines must define their operation in terms of a machine-independent underlying representation. If this is done correctly, the layering of underlying representations falls out naturally.

In summary, the treatment of data types in EL1 is based on a set of fairly straightforward notions, most of which are simple to implement. Much of this treatment can be carried over to other high level programming languages. The linguistic power they add is considerable.

REFERENCES

1. PL/I Language Reference Manual. Form C28-8201-2. IBM Syst. Ref. Lib. (1969).

2. Abrahams, P.S., et al. The LISP2 Programming Language and System. FJCC 1966, vol. 29, 661-676.

3. Van Wijngaarden, et al. Report on the Algorithmic Language ALGOL 68. Mathematisch Centrum, Amsterdam, MR 101, February 1969.

4. Garwick, J.V. GPL, A Truly General Purpose Language. CACM, vol. 11, no. 9 (September 1968), 634-638.

5. Hoare, C.A.R. Record Handling. In Programming Languages, ed. by F. Genuys. Academic Press, New York, 1968, pp. 291-347.

6. Wegbreit, B. Studies in Extensible Programming Languages. ESD-TR-70-297, Harvard University, Cambridge, Mass., May 1970.

7. Wegbreit, B. The ECL Programming System, Technical report, Division of Engineering and Applied Physics, Harvard University, Cambridge, Mass., April 1971. (To appear in Proc. FJCC 1971.)

8. Reynolds, J.C. A Set-Theoretic Approach to the Concept of Type. Working material for the NATO Conference on Techniques in Software Engineering, Rome, Italy, October 1969.

9. Dijkstra, E.W. Notes on Structured Programming. T.H. Report 70-WSK-03, Technological University Eindhoven, The Netherlands, April 1970.

UNCLASSIFIED
Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Harvard University Center for Research in Computing Technology Cambridge, Massachusetts 02138	2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED
	2b. GROUP N/A

3. REPORT TITLE

THE TREATMENT OF DATA TYPES IN EL1

4. DESCRIPTIVE NOTES (Type of report and inclusive dates)
None

5. AUTHOR(S) (First name, middle initial, last name)

Ben Wegbreit

6. REPORT DATE August 1971	7a. TOTAL NO. OF PAGES 49	7b. NO. OF REFS 9
8a. CONTRACT OR GRANT NO. F19628-68-C-0379	9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-71-341	
b. PROJECT NO.		
c.	9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
d.		

10. DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

11. SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY Deputy for Command and Management Systems Hq Electronic Systems Division (AFSC) L G Hanscom Field, Bedford, Mass. 01730

13. ABSTRACT

In constructing a general purpose programming language, a key issue is providing a sufficient set of data types and associated operations in a manner that permits both natural problem-oriented notation and very efficient implementation. The language EL1 contains a number of features specifically designed to simultaneously satisfy both requirements. The resulting treatment of data types includes provision for programmer-defined data types and generic routines, programmer control over type conversion, and very flexible data type behavior, in a context that allows efficient compiled code and very compact data representation.

DD FORM 1 NOV 65 1473

UNCLASSIFIED
Security Classification

UNCLASSIFIED
Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Data types						
Modes						
Mode unions						
Type conversion						
Coercion						
Generic functions						
Extensible languages						
Data type definition						
Data description language						
Compilation						

UNCLASSIFIED
Security Classification