ESD ACCESSION LIST
TRI Call No. 75091
Copy No. 1 of 2 cys.

TRI FILE COPY

ESD-TR-71-343

THE ECL PROGRAMMING SYSTEM

Ben Wegbreit

August 1971

ESD RECORD COPY
RETURN TO
SCIENTIFIC & TECHNICAL INFORMATION DIVISION
(TRI), Building 1210

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

Approved for public release;
distribution unlimited.

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

AD0736413

LEGAL NOTICE

When U. S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

ESD-TR-71-343

THE ECL PROGRAMMING SYSTEM

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

Approved for public release;
distribution unlimited.

(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.

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

ABSTRACT

ECL is a programming language system designed to provide an environment conducive to effective programming. It consists of a programming language and a system built around that language to serve as a tool and vehicle for program manufacture. The language contains comprehensive data types, operators, control structures and storage management. The system is oriented to interactive program composition and debugging with smooth transition to efficient compiled code. Most important, the system allows the programmer to tailor the environment to suit his needs.

TABLE OF CONTENTS

SECTION I. INTRODUCTION 1

SECTION II. SYSTEM ORGANIZATION AND DESIGN PHILOSOPHY 4

SECTION III. SYSTEM FACILITIES 10

3.1 Syntax Extension 10

3.2 Storage Management 12

3.3 Data Type Extensions 13

3.4 Compilation 15

3.5 Errors and Interrupt Handling 17

3.6 Control Structures: Paths and Multiprogramming 19

SECTION IV. SUMMARY 21

REFERENCES 22

LIST OF FIGURES

Figure No.		
1	Primary System Modules	5
2	Program Development in ECL	8

SECTION I

INTRODUCTION[1]

ECL is a programming language system currently being implemented as a research project at Harvard University[2]. Its goal is an environment which will significantly facilitate the production of programs. In this paper, we describe the motivation for this project, present the approach taken in its design, and sketch the resulting ECL system. Detailed treatment of specific aspects of the system is found elsewhere [1], [2].

Programmers, whether professionals or casual users, manufacture a unique product, programs: objects, often large, which must be coded, modified, debugged, verified, made efficient, and run on data. In providing an environment for this manufacturing, four goals are considered primary:

1. To allow problem-oriented description of algorithm, data and control over a wide range of application areas.

2. To facilitate program construction and debugging.

3. To allow and assist in the development of highly efficient programs.

4. To facilitate smooth progression between initial program construction and the final realization of an efficient product.

ECL consists of a programming language and a system built around this language to meet these goals.

The language component, called EL1, includes most of the concepts of ALGOL 60, LISP 1.5, and COBOL. It provides standard arithmetic capability on scalars and multidimensional arrays, dynamic storage

---

[1] This paper has also been included in the publication series of the Harvard Center for Research in Computing Technology as Technical Report 3-71.

[2] The current implementation is on a PDP-10 running under the 10/50 monitor. Versions for the IC9000 and other machines are contemplated.

allocation with automatic storage reclamation, record handling, and algorithm-independent data description. Further, it provides facilities which allow the programmer to define extensions to the language to tailor it to each particular problem area. New data types, new operators, new syntax and new control structures can be added to the language enabling the program to model directly the objects, unit operations, relations, and control behavior of each problem domain. For example, list processing, matrix arithmetic, string manipulation by pattern matching and replacement, and discrete simulation can all be carried out in EL1 by appropriate extensions.

To aid program construction and debugging, the ECL system has been designed for use in an interactive, on-line fashion[1]. Programs can be composed at the console using a text editor and run interpretively with appropriate levels of error checking, tracing, and conditional suspension. With execution suspended, the programmer can examine data or program, modify either, and resume. Any variable may be declared "sensitive"; changes to its value are monitored and an interrupt generated whenever a programmer-specified predicate associated with the variable becomes true.

Several system facilities contribute to the construction of efficient programs. One is the compiler. Variables can be data typed so that the compiler can perform type checking, compile in type conversion, and choose among alternative procedure bodies on the basis of argument data types. The compiler can be called at any time, so it is possible to write procedures which compile themselves or other procedures. To allow economical use of storage, the language allows packed data (e. g., bits, bit strings, bytes, and byte strings) and operations on such data objects. This is carried out in a machine-independent notation and representation so that programs using this are not tied to a particular machine. To allow the construction of efficient programs which include asynchronous components, ECL includes multiprogramming and a programmer-controllable interrupt system.

Efficiency, in any metric, is seldom gained at one fell blow; programs are only relatively stable. Even after code is checked out with the interpreter and compiled, it is usually changed and frequently requires debugging. Further, it is sometimes necessary to compile part of a program in order to get sufficient speed to test an algorithm against a large data base. Since the road is filled with relapses, it is important to allow smooth progression and regression between initial

---

[1] This is not to the neglect of batch processing. Any interactive language can be used in batch mode if the job control commands that would normally come from the console are taken from a file and results which would normally appear on the console are written to a second file. ECL allows such switching of command streams, so that batch processing falls out as a subcase of its normal mode of operation.

construction and final product. It should scarcely need saying that the languages acceptable to the interpreter and compiler are identical and that compiled and interpreted code may be freely intermixed with no restrictions. For example, the result of compiled code may be used as an argument to interpreted code; a goto in interpreted code may lead back into compiled code; variables local to compiled code may be accessed by interpreted code, etc. A less familiar concept, but equally fundamental, is the notion that compilation is not all or nothing. In ECL, compilation can be carried out to any level depending on the amount of information supplied to the compiler: specifically, the number of program components that the programmer is willing to accept as being invariant. The more invariants, the better the compiled code. As with interpreted code, the execution of compiled code may be broken (either by an internal condition or an external interrupt) to allow intervention by the programmer, e. g., for debugging purposes.

The primary motivation for, and the intended use of, the ECL programming system is "difficult" programming efforts. That is, projects which could otherwise be carried out only with considerable waste of human or machine resources. It is our intention that ECL be usable for production programming, hence the emphasis on machine efficiency. This is not to say that the requirements of interactive usage have been slighted in system design. Quite the contrary, we view good interaction capability and a well-engineered debugging facility as significant tools in tackling a difficult programming project. The utility of on-line debugging should be clear. Equally important is the use of an interactive capability in developing and refining algorithms. Still more important is the use of interaction in allowing measurement of program behavior and the attendant optimization based on knowledge of this behavior.

SECTION II

SYSTEM ORGANIZATION AND DESIGN PHILOSOPHY

Before discussing ECL in detail, it will be useful to outline its internal organization and discuss the philosophy which underlies its design.

Normally, one uses ECL on-line, communicating with the system via a console. As seen by the programmer, ECL is an executor of input commands. Syntactically, commands correspond roughly to statements of an algebraic language; semantically, commands embrace all actions expressible in the system. Hence, commands include: conventional algebraic statements, definitions used to construct new procedures and operators, and the "job control" statements of a batch processing system such as instructions to compile procedures, transact with data sets, create and destroy processes, etc.

As seen by ECL, the programmer is a source of input commands. We will take the system's point of view. It reads and parses each command, interprets it, and turns to the next command. Since commands include calls on procedures which may be programmer-defined, the interpretation portion of the cycle may set off the running of a compiled program.

At the heart of ECL is the command handler — the routine which controls the above command loop. It has two main components: the parser and the interpreter (c. f. Figure 1). The parser calls on a lexical analyzer to decompose the input stream into lexemes. The parser then analyzes the lexeme stream as directed by parse tables previously derived from a syntactic specification of the language. Both the input source and the parse tables may be changed by commands, so that the source of commands and the language in which commands are expressed are subject to change by the programmer. The output of the parser is a representation of the command as a linked list. Constituent syntactic units are represented by sublists, recursively. The command handler calls on the interpreter to execute the command. When this is completed, control returns to the command handler which outputs the result and then calls on the parser for the next command.

The list structured representation has two uses. On the one hand, it can be executed directly by the interpreter; on the other, it is a convenient form of input to the compiler. This achieves several economies. A program need be parsed only once, on input. Hence the interpreter does not reparse a line each time it is encountered during execution, e. g., in a loop. Also, the compiler is considerably simplified since it is not at all concerned with parsing.

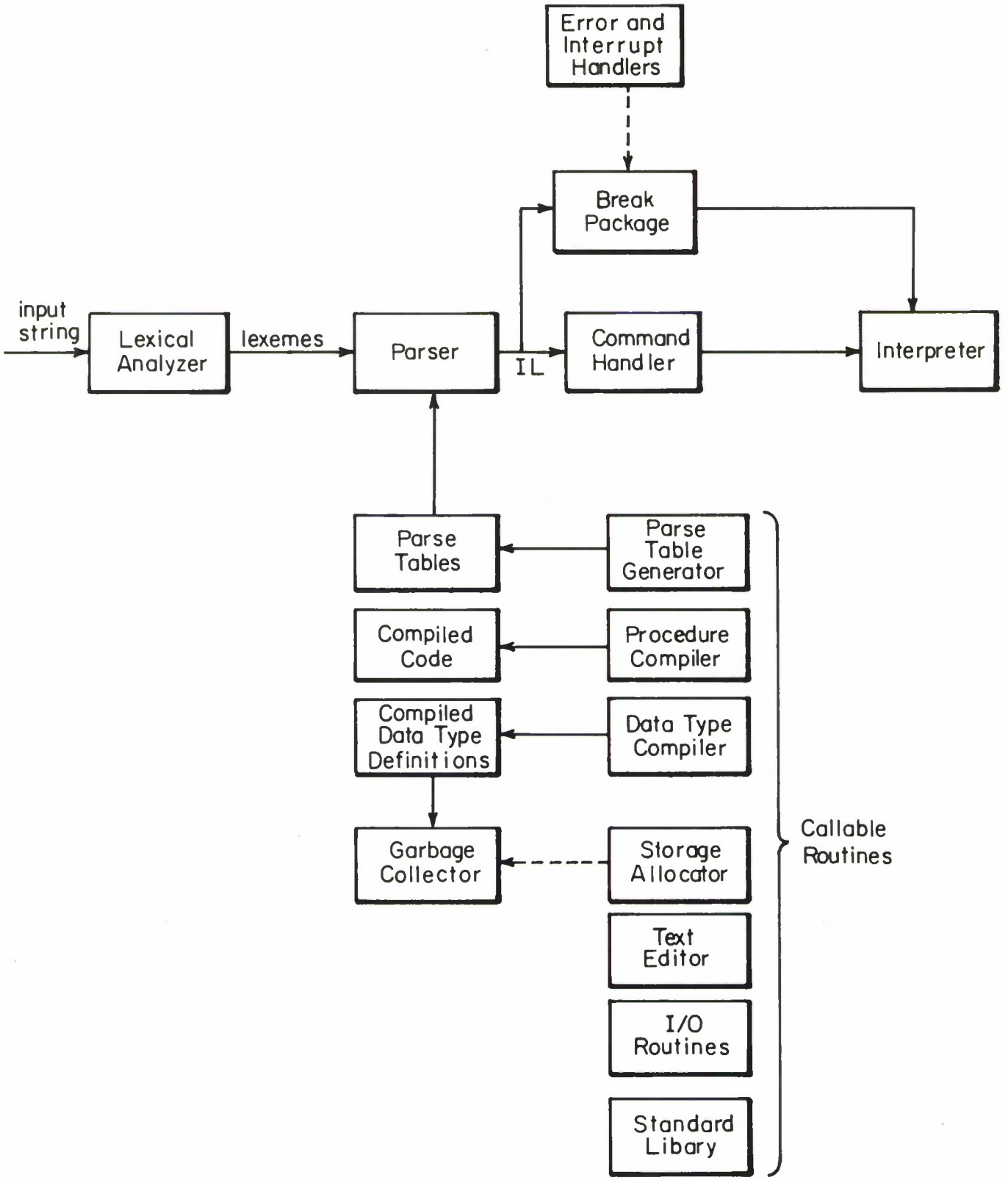

FIG. 1 PRIMARY SYSTEM MODULES

Most commands will be function calls, i. e., the application of a routine (procedure or operator) to a set of arguments. Routines initially available in ECL include:

1. The conventional arithmetic, relational, and trigonometric routines.

2. A set of I/O routines.

3. A routine for defining new procedures and operators.

4. The compiler.

5. Routines to define new data types.

6. Routines to change the parse tables, thereby changing the syntax of the language.

7. Routines to allocate storage, and a garbage collector to reclaim storage no longer in use.

8. Routines to create, run and destroy processes.

The first three sets require no explanation; the others will be discussed individually in subsequent sections.

It should be clear that ECL is an unusually eclectic system. This is unavoidable; a complete programming environment necessarily includes many components, each fairly complex. There is a certain danger in this. Such a system can easily become very large, hence prohibitively expensive to implement and maintain. No less dangerous is the possibility that a system may be unwieldy for the casual users. Finally, there is the danger that the system may impose too much or the wrong kind of structure on the programmer. With each decision made incorrectly, a language system inconveniences some class of users. With many decisions to make, a system is certain to inconvenience all programmers some of the time.

In ECL, these very real dangers of an eclectic system have been avoided by judicious application of four concepts: (1) extension mechanisms, (2) sustained variability, (3) bootstrapping, and (4) system uniformity.

The first of these has been mentioned earlier. The idea is to construct a small initial system consisting mostly of powerful definition facilities for self-extension. Only the initial system — the nucleus — need be implemented and maintained by the system's creators. The rest is built on this by the programmer or programming group to suit its needs and taste. The ECL provides definition mechanisms for extension along three axes: syntax, data types, and control structures.

A second key concept, distinct from language extension, is systematic variability. That is, the deliberate provision for access by the programmer to key points at which he can control system behavior. All well-designed systems have key points of control; usually, however, these points are deeply embedded in the system either on grounds of supposed efficiency or because actions to be taken were believed to be incapable of sustaining intelligent variation. Seldom is the burial justified. Allowing programmer control over such issues provides a surprising amount of power. In ECL, three points have been singled out for attention: error and interrupt handling, input/output stream direction, and data type conversion on binding formal parameters of routines to their arguments.

Bootstrapping, i. e., using the system to define parts of itself, provides system variability at another level. In ECL, bootstrapping has been a fundamental implementation technique. The data type extension facility was used to create the system data types needed by the interpreter itself. Further, large parts of the system are coded in the language, most notably the compiler. Such system modules can be run either interpreted or compiled; the compiler, of course, is compiled by itself using the interpreter. For the system implementor, this technique avoids a large amount of machine language coding with the attendant benefits of rapid production, better system organization, and ease of change. For sophisticated system users, this bootstrapping provides an additional point of variability: those portions of the system coded in the language are accessible to change.

The fourth concept in the ECL system is uniformity. Insofar as possible, the entire environment of the programmer is treated as a single homogeneous space without special times, cases, or preferred objects. Correspondingly, the implementor has to deal with a system notable for its lack of special cases and "funny" situations.

All data types (called <u>modes</u> in ECL) are treated equally. Each class of objects in the system has a mode; for each mode there are values of that mode; declarations are used to create variables which can be assigned values of that mode. A procedure is like any other object in this respect. It is a value, it has a mode, and may be assigned to be the value of a procedure-valued variable. Programs can be treated as data and data as programs. Programs which generate other programs are straightforward. Files (somewhat generalized) are another mode in the system, so that programs can compute the source of or sink for input/output and can arrange for arbitrary transformation of the data during transmission. Finally, there is no preferred status for the data type <u>mode</u>. A mode (e. g., <u>integer</u>) is just as legitimate a value as, say, 3. 1. Hence, mode values may be computed, assigned to variables of data type <u>mode</u>, passed as arguments to routines, etc. There are a number of system-defined routines which take modes as arguments and produce new modes. Additional routines for computing modes may be defined by the programmer from these. Hence, a

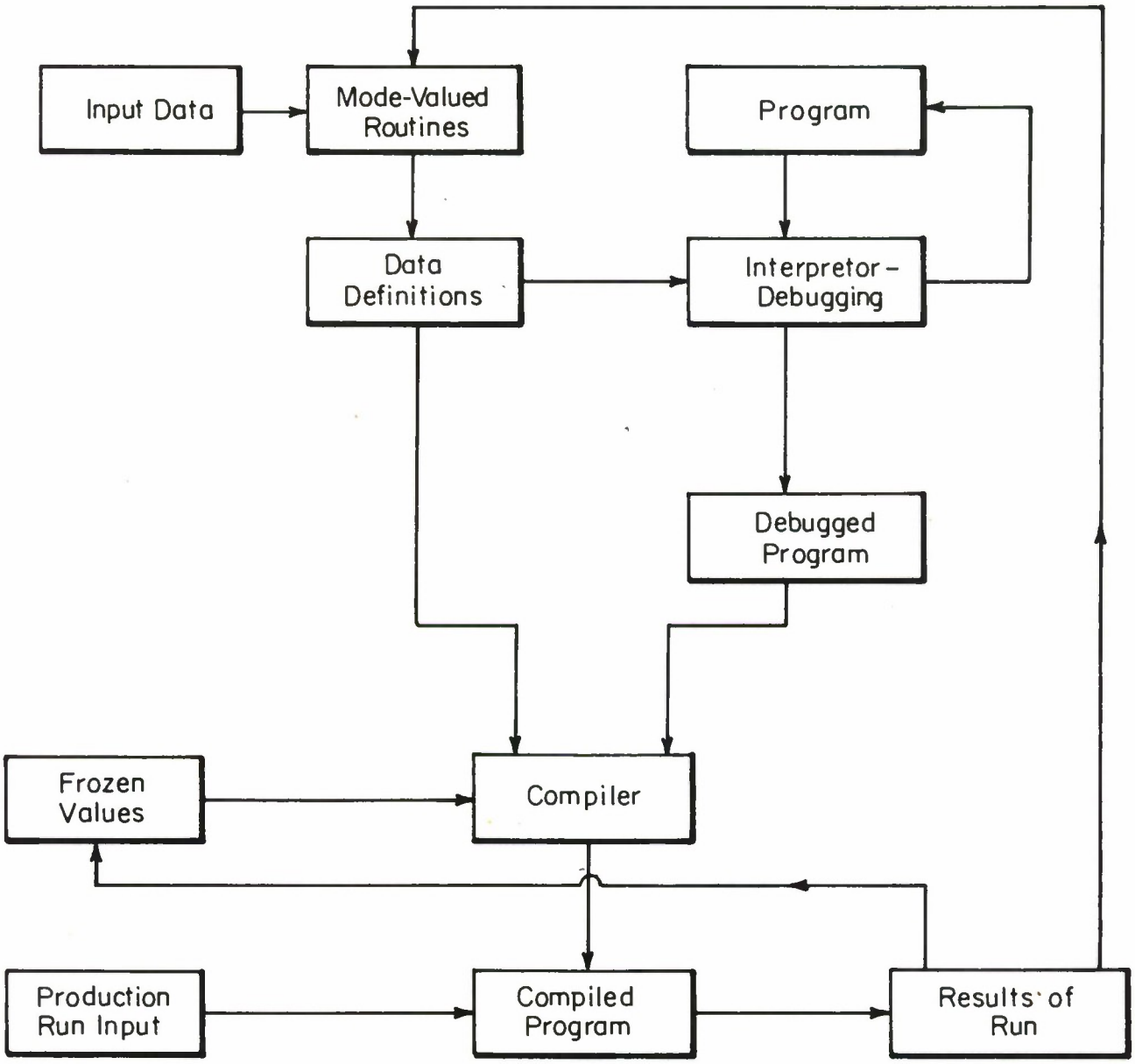

FIG. 2 PROGRAM DEVELOPMENT IN ECL

programming project might include all of the following (c. f. Figure 2):

1. Defining a set of routines which compute modes.

2. Writing a program which uses variables whose modes are of the class generated by 1.

3. Running the program defined in step 2 interpretively, halting, modifying and debugging it.

4. Running the routines of step 1 on input data to compute a set of modes.

5. Compiling the program of step 2 to get object code tailored to the data types computed in step 4.

6. Running the object program of step 4 on a data set.

Conceivably, this could be done in a single console session. Alternatively, these steps might be carried out over the course of several months as a large programming effort goes through the process of defining its data formats, coding and checking out its routines, metering the input profile, compiling and tuning code, and finally running. The key point is that all these steps can be carried out in a single system using a common language to describe their actions.

SECTION III

SYSTEM FACILITIES

In this section we discuss the key facilities seen by the programmer using ECL. In the interest of brevity, we concentrate on innovative features and treat lightly those which are straightforward. In discussing the language component, we will ignore all but its extension mechanisms; in particular, we do not give its syntax or programming example in this paper. Suffice it to say that the language is ALGOL-like in syntax, ALGOL/LISP-like in semantics and that a formal description of both syntax and semantics exists [3]. Built-in data types of the language include characters, integers, reals, and Booleans; built-in operations include the usual operations on these types. A system-provided extension package adds to this the data types symbol, list and arrays of reals, integers, and Booleans along with appropriate operations.

3.1 Syntax Extension

A number of proposals for syntax extension have appeared during the past few years, proposals ranging from simple macro extension schemes requiring prefix macro name triggers, to recognition of arbitrary context-free languages with complex parse-tree manipulation facilities. The technique used in ECL has two key properties: (1) it is very efficient in both parse speed and storage required, (2) it includes specific provision for simple common additions as well as complex comprehensive changes.

The parser is a deterministic pushdown store analyzer. It scans the input stream from left to right, recording the progress of the parse in state information. At each step, the parser either reads the next lexeme and adds it to the pushdown store or it reduces the top elements of the pushdown store. In either case, it goes into a new state. In the case of a reduction, employed whenever a complete syntactic phrase has been found, a semantic action associated with the phrase class is executed. The choice of read or reduce, the reduction to be made, and the next state to be entered are recorded in a syntax table as a function of the current state, next lexeme, and top elements of the pushdown store. This table is computed by a parse table generator using a technique developed by F. DeRemer [4], from a syntax specification in BNF. Semantic actions augmented to each syntax rule specify the desired mapping from the parse tree into the intermediary list structure representation—IL. Each syntactic form of the source text is therefore represented by some IL list.

The interpreter and compiler treat certain IL lists (e.g., those representing a <block>) specially; all others are taken as procedure or operator calls where the head of the list is the function name and the rest of the list is the set of arguments. Therefore, most augments

simply map the syntactic construction into prefix form. The final element of the language specification is the definition of the function names used as prefix operators in IL.

The language may be extended by (1) adding to the syntax specification new syntax rules with augments, (2) defining the function names used as prefix operators in the new IL forms, thereby defining the semantic specification, (3) calling the parse table generator on the new syntax specification, and (4) switching the parser to be driven by the resulting new parse tables. In subsequent input any command, in particular any program, containing the new constructs will be analyzed employing the new syntax rules, mapped by the augments into prefix form, and executed by the associated function in the semantic specification. Compiling the program and the semantic specification functions will yield acceptable although not specially optimized code for the new construct.

The most common additions to the language will surely be new operators. For example, much of APL [5] can be obtained simply by defining the appropriate array operators. While new operators could be added by using the above technique, this is needlessly complex for such a simple addition. Hence, ECL provides a special facility to handle this, making the definition of a new operator no more difficult than the definition of a new procedure. An identifier in the language can be written either like a PL/I identifier (e.g., X, TEMP, FOO, COEFFICIENT) or as a sequence of special symbols (e.g., +, -, **, +←, =#>). Any identifier can be declared to be a prefix operator, an infix operator, or both (e.g., the minus sign denotes negation as a prefix operator and subtraction as an infix operator.). An infix operator can be given an integer index from 1 to 7 specifying its binding strength.

The mechanism used to implement this facility is a simple extension of the basic analyzer; hence, operator and other extensions mesh together smoothly. The initial syntax specification includes the syntactic categories <prefix operator> and <infix operator$_i$> for $i = 1, \ldots, 7$. All operators are recognized as <identifiers> by the lexical analyzer and are handed to the parser with syntactic category <identifier>. The parser changes the syntactic category to <prefix operator> or <infix operator$_i$> under "appropriate conditions" (e.g., for the second identifier in X**I). The parser recognizes the possibility of such an appropriate condition by means of a second set of parse tables (actually part of the symbol table) which specifies which identifiers may be used as operators and in what roles (i.e., prefix, infix$_i$, or prefix and infix$_i$). The tricky point here is distinguishing between different uses of an identifier symbol; e.g., if #@ has been declared to be both a prefix and infix operator then it may appear in:

#@ B as a prefix operator acting on B,

A #@ B as an infix operator acting on A and B,

#@ ←... as an identifier being assigned a new (operator) value.

The parser distinguishes between these three uses in the same way as the human reader — by local context. The read routine of the parser examines each <identifier> that can be used as an operator, checks its local context and decides how it is being used in the context, and possibly changes its syntactic type to <prefix-operator> or <infix-operator$_i$>. The rest of the parser, in particular the part that performs reductions, is oblivious to this local transformation; it sees either an <identifier>, a <prefix-operator>, or an <infix-operator$_i$> and regards these as disjoint terminal categories.

3.2 Storage Management

There are two classes of storage provided by the ECL system: (1) storage automatically allocated and freed at block entry and exit (on the stack) and (2) storage dynamically allocated by the program (in the heap, using ALGOL 68 [6] terminology). The former is handled by well-known stack implementation techniques and requires little discussion. In providing dynamic storage allocation, however, there is a critical design decision — whether to provide automatic storage reclamation or whether to require explicit return of unused storage, e.g., by a free command.

A common characteristic of allocated storage is that the programmer does not, in general, know when it is becoming unused. Typically, a block is pointed to from many places, most of which are in other allocated blocks. Deciding when the last reachable pointer ceases to reference a block is therefore no simple matter. Keeping track of this at all times places a burden upon the programmer, one that may significantly complicate a program. Hence, ECL provides automatic reclamation[1]. Garbage collection was chosen as the implementation technique since this requires the least housekeeping storage and is guaranteed to find all unused storage. The programmer sees only a system-provided allocation function — ALLOC. Specifically, ALLOC(M) allocates an object of mode M and returns a pointer to this object. When available storage is exhausted, the allocator invokes a garbage collection.

The garbage collector is basically straightforward. A few subtle points are, however, worth mentioning. The trace phase traces all storage blocks referenced and marks all machine words in use using a bit map. By marking machine words, not objects, it is possible to mark only part of a block in a compound object. Garbage collection leaves untouched these parts actually referenced and reclaims the rest. The difficult point in the trace phase is the possibility, indeed almost certainty,

[1] Dynamic storage management in ECL therefore differs from that of PL/I [7]. The latter provides dynamic storage allocation but no automatic reclamation.

of tracing through objects having programmer-defined mode. Given an object, the trace routine must be able to determine how big it is (so as to mark all of its words), whether or not it has pointers within it and, if so, where they are (so they can be traced). This information is calculated by internal system routines whenever a new mode is defined and is entered into tables associated with the mode. Once marking is complete, the garbage collector sweeps linearly through storage, collects all unmarked work into maximal contiguous blocks, and sorts these blocks by size into a set of linked lists forming the free storage pool. Keeping different lists for various sized blocks (currently, one list for each power of 2) speeds up subsequent allocations.

Clearly, it is best to avoid garbage collection entirely if possible. We therefore stress that ECL also provides automatic, block-structured storage. This behaves like a normal ALGOL 60 stack, holding variables declared to reside on the stack as well as arguments to routines, and temporary results. Hence, all computation concerned with ALGOL-like objects (e. g., scalars and arrays of fixed-point and floating-point numbers) can be carried out on the stack and requires no use of the free storage mechanism.

3.3 Data Type Extensions

Perhaps the chief requirement of a programming language intended to serve a wide range of application areas is an equally wide range of data types or modes. Clearly, a language must include integers and reals for numerical computation, Booleans as the result of relational operations, and characters for headings and labels. List processing implies data objects which reference other objects, i. e., pointers. However, compiled code can be made considerably more efficient if a pointer variable may be declared as restricted in what types of objects it can point to; this introduces integer pointers, real pointers, character pointers, Boolean pointers, etc. Packed objects such as bit vectors are sometimes essential in saving core storage. A list of interesting data types could go on indefinitely.

In the face of so many diverse claimants for inclusion in a language, the only sensible solution is an extension facility: here, a mechanism for defining new modes. The language provides a few basic modes and five primitive routines for defining new modes in terms of these. The primitive mode constructors are ARRAY, PTR, STRUCT, PROC, and ONEOF; these create arrays, pointers, heterogeneous structures, procedures, and mode unions, respectively. These mode constructors are callable routines. They evaluate their arguments, perform some computation, and deliver a result having data type mode. The resulting modes are just as legitimate as the built-in modes. Objects of these types may be assigned values, passed as arguments to routines, returned as the value of routines, etc.

The key point of this facility is that the mode constructors compile modes in the same sense that a traditional compiler compiles procedures. That is, they calculate once, at the time a mode is created, all information about the mode that the system will subsequently need. One such computation is the storage layout for compound objects — how to represent objects of the constructed mode in the fewest possible machine words. The current algorithm produces optimal packing on almost all cases; e. g., a structure consisting of one 18-bit pointer, four 7-bit characters, three 5-bit fields, one 3-bit field and four 1-bit fields will be packed into two 36-bit words[1]. The result of the calculation is a structure table giving the location and mode of each component in a compound object, to be used by subsequent phases of mode compilation and by the runtime routines. Another computation is preparing the tables for the garbage collector, in particular, deciding whether an object of this mode contains a pointer to be traced. The most important computation, however, is the generation of three blocks of machine code: (1) to construct objects of this mode, (2) to perform assignments to objects of this mode and (3) (for compound objects only) to select the individual components of objects of this mode. To effect construction, assignment, and selection, the interpreter executes these code bodies so that these operations are partly compiled, even from interpreted code. The compiler may either use these bodies or compile corresponding code in-line depending on whether it is optimizing space or time.

The programmer can use these mode compilation routines to define the types he needs. For example, bit vectors are defined as ARRAYs of Booleans, multidimensional arrays of any sort are defined by composing the function ARRAY, data processing records are STRUCTs of characters and integers, and a list of reals is constructed from blocks of identical STRUCTs each containing an integer and a pointer to the next block. Further, the programmer can define new mode-valued procedures (i. e., mode generators) in terms of the primitive routines. We anticipate a library of modes and mode-valued procedures analogous to a library of numerical algorithms.

One additional facet of the mode extension facility requires discussion. When a mode is defined using the system primitives, certain behaviors are automatically assumed. For example, if BYTE names the mode ARRAY of 8 Booleans (represented as an 8-bit object), it will be assumed that an object X of mode BYTE has 8 components which

---

[1] This is, of course, entirely machine-dependent. However, the programmer never sees this packing. He deals only with objects of the language which have the right properties — e. g., access to the second 1-bit field gets the desired value. This differs from the approach taken in LISP 2 [8] where the programmer deals explicitly with the bit packing himself.

may be accessed as Boolean values by X[I] for I = 1, . . . , 8, that assignment of one BYTE to another copies all 8 bits, and that if X is to be passed as an argument to a routine then that routine must have a corresponding formal parameter of mode BYTE. If the programmer wishes, he can override these assumptions and specify the behavior he wants. He can, for example, declare that an object X of mode BYTE is to have the following behavior:

1. X can be assigned an integer value (e. g., X ← 73). If the value can be represented in 8 bits 2's complement notation, an 8-bit assignment is made; otherwise, an error procedure P is to be called with the integer value as an argument.

2. X can be used as an argument to a routine taking an integer formal parameter, in which case sign extension is used to get a full-word value to be treated as signed integer.

3. X is to be treated as <u>if it</u> had an additional 9th component recording the number of leading 0's in its bit configuration. X[9] is always interpreted as an integer count of the number of leading 0's in X[1]. . . X[8] at that point in the computation.

Using this facility, the programmer can specify exactly the properties of his data objects. Encoded representation for values, variables which monitor their values, objects with "protected" fields, and the ability to represent sparse compound objects fall out as simple applications.

3.4 Compilation

A compiler can be viewed in two distinct ways. It can be taken as a device for translating programs from source representation to one which can be executed directly by some computing machine. Alternatively, it can be seen as a means for factoring a computation into two parts: that which is invariant with respect to input data and can be performed once at compile time, and that which depends on the data and is therefore postponed until run time. The second view subsumes the first and is surely the more fundamental. Translation is only one of many computations that can be factored out. Others include: evaluation of expressions at compile time, data type checking, and generic selection. The interesting problems in compilation can be best addressed by pushing the notion of factoring to take advantage of additional invariants. It is this line of approach that characterizes the ECL compiler.

A program consists essentially of a large number of variables, a few constants, and some punctuation to paste this all together. ECL carries the notion of variable somewhat farther than most languages. For example, a program may declare X to be an object of mode TRIPLE when TRIPLE is a mode-valued variable or may apply FOO to

a set of arguments where FOO is a procedure-valued variable. This allows the programmer great flexibility, but presents the compiler with the problem of dealing with an unknown value of the variable. There are three possible routes it might take:

1. Attempt to deduce the value by examining the structure of the program, e. g., look for an initialization of or assignment to TRIPLE and verify that the value will not change.

2. Obtain explicit assistance from the programmer.

3. Wait until run time when the value will surely be known.

From a theoretical point of view, the first route has certain appeal. However, the inevitable undecidability results are assurance that in general one can deduce nothing; discovering subcases in which interesting deductions can be made is a significant research problem. Further, making such deductions is often a pointless task: the programmer usually knows far more about a program than could ever be deduced from examining it; he alone knows its intended function and the environment in which it is to run.

Hence, the second route is the mainstay of the compiler. In compiling a procedure P, the compiler is called with two arguments: P and a list L of all variables in P whose value is to be "frozen." P is then compiled with each variable on L replaced by the value of that variable at the point where the compiler is called. (It will be recalled that this point might be while executing another procedure or P itself.) For example, if X is declared in P to be a TRIPLE and TRIPLE is on the <u>frozen list</u> L, then the value of TRIPLE must be a mode and this mode is taken as the data type of X. Similarly, if FOO appears on L, then an appearance of FOO($arg_1, \ldots, arg_n$) can generate code specific to the value of FOO, e. g., by in-line expansion. To treat a related case, it may be that FOO does not appear on L but FOO is declared in P to have mode FOOMODE and FOOMODE is a variable on L. The compiler then does not have access to the value of FOO, but it does know its data type, i. e., the modes of its arguments and the mode of its result. Hence, the compiler can perform type-checking of arguments in calls on FOO and type-check the usage of its result in a larger context (e. g., A + FOO($arg_1, \ldots, arg_n$)).

Any set of variables may appear on the freeze list L. If an operator and all its arguments are frozen (e. g., by appearance on L), then the entire function application is frozen[1]. By recursive application of this rule, it is possible for arbitrary complex expressions to be

[1] Assuming that the operator definition contains no free variables.

frozen. These can and will be evaluated during compilation. For example, if X, Y, FOO, and FUM are all on L, then

FUM(Y, FOO(Y), FOO(FOO(X)))

will be evaluated, the result replacing that expression in the code generated.

For those variables not in L, the third route remains open: wait until run time to obtain its value. This includes "ordinary" variables as well as mode identifiers and procedure names. For example, if TRIPLE is not on L, then in a procedure with formal parameter declared to be a TRIPLE the data type is left open until the procedure is called. The compiler is governed by a consistent rule: it will compile the best code it can with the amount of information (i. e., set of invariants) given to it. This code can be anything from a single call on the interpreter (in those cases where nothing useful is frozen) to the value of the program (in those cases where everything is frozen). The interesting cases fall somewhere in between.

It is possible to compile a procedure dynamically during the course of some computation as values are calculated and frozen. Hence, a computation may involve reading part of the input data, compiling a program specific to that data, and running the compiled routine on the remainder of the data. Programs which periodically recompile themselves based on statistics gathered during the course of a run are an obvious application.

3.5 Errors and Interrupt Handling

It should go without saying that a modern programming language needs a facility for handling errors and interrupts. That is, a means for accepting asynchronous external interrupts and dealing with internal error conditions. ECL takes care of both by means of the procedure call mechanism. Every error or interrupt may be treated as if the program had explicitly called an error handling routine of its choice from the point where the error or interrupt occurred. Associated with each[1] error or interrupt is a procedure name (e. g., ENDOFILE, FLOATOVF, FIXOVF, etc.). When an error occurs or an interrupt comes in, the normal computation sequence is suspended at that point and a system routine ERR is entered. ERR finds the symbolic name associated with that error/interrupt condition and then checks whether there is a variable of type procedure valid at that point in the suspended computation. If no such variable exists, ERR types out an error message and goes into a break routine which preserves the state of the

[1] These include: end of file, fixed point overflow, floating point overflow, taking the value of a null pointer, the completion of certain I/O transactions, subscript index out of range, and timer interrupt.

computation and accepts further commands from the console. If, however, there is an appropriate variable, then the associated procedure is called; so far as ECL is concerned, that is the end of the matter—any further action is the responsibility of the called routine.

In the case of an external interrupt, it may be possible to handle the interrupt without regard to the suspended environment. Such an interrupt processor may perform some computation on the interrupt message, change global flags, variables, and queues, then continue with the suspended computation. However, to handle most errors and internal interrupts, it is necessary to access the environment in which the condition occurred. For example, it will frequently be useful to examine the call structure (the sequence of function calls that lead to this point) and to examine and change the values of variables in the suspended environment. ECL allows access to this information, not as a special feature offered to the error handling routine, but rather as a system facility available at all times. A stack of return points is used by ECL so as to allow recursive procedures; it is a simple matter to also stack the symbolic name of the called routine. Hence, any procedure, whether called to process an interrupt or otherwise, can obtain the symbolic name of the Ith dynamically preceding routine (CALLER(I)) and can access the value of any variable in that environment (DYB(<variable name>,I)).

An error or interrupt routine can exit in a number of ways, depending on the cause of the interruption. GOTO L transfers control to the nearest enclosing label L; this, however, may be arbitrary far back in the chain of calls. Since the argument to GOTO is evaluated, it is possible to use DYB to get to an arbitrary level, even one "masked" by another label of the same name; e.g., GOTO DYB(L,I) transfers control to the label L defined in the Ith enclosing environment. Two other routines allow returning a computed value. For errors, CONTWITH(<expression>) continues computation with the value of <expression> used in place of the expression which caused the error. RETURN(<expression>,I) acts as if the Ith routine back on the call chain had suddenly returned to its caller with the value of <expression>.

In summary, this scheme provides a powerful, inexpensive mechanism giving the programmer fine control over errors and interrupts. The program is armed for a specific error or interrupt in any scope where a procedure-valued variable of the appropriate name is defined. Errors or interrupts for which the program is so armed are handled by the specified routine. Control and environmental inquiry facilities of the system provide the linguistic power needed by the routine to handle such conditions intelligently.

3.6 Control Structures: Paths and Multiprogramming

The error/interrupt facility allows the mainline of computation to be suspended so that a subsidiary computation can be performed to process the cause of interruption. However, this is strictly a priority situation: the interrupt routine must complete and exit before the main computation can continue. It is frequently useful to deal with subsidiary computations going on whenever there is any work to be performed, in parallel with the main computation.

ECL provides such parallel computation. In general, a job consists of some dynamically varying number of independent processes (called paths in ECL). What has been described thus far is the behavior of one such path. Indeed, when ECL is started, there is but one path. However, that path may create new paths and start computations on these paths, computations which in general proceed asynchronously with respect to computation on the starting path. Each path is an independent computational entity consisting of an environment (the call structure and variables created during this call sequence) and an activation record which, among other things, records the state of the path. States include suspended, waiting for some resource (e.g., I/O), and runnable. All runnable paths are parallel processes. The state of a path may be changed by a number of commands; these include SUSPEND some path, WAIT some period of time, and the Dijkstra P and V semaphores [9] for synchronization among paths. All paths have a certain portion of their environment in common — potentially, any allocated storage. Hence, it is possible for two or more paths to reference common data, e.g., a buffer, a set of flags, or a message queue. This, coupled with the P and V semaphores, allows the conventional sort of cooperating sequential processes to be established.

The really interesting aspects of the ECL path facility lie, however, in its ability to host nonconventional multiprogramming, in particular, control regimes not explicitly anticipated by its designers. That is, like many other facilities in ECL, the multiprogramming mechanism is extensible. As with other extension facilities, that for multiprogramming consists of a set of primitives and a framework for combining them. Primitive operations include creating a path, setting up a function to be executed in a created path, running a path, deleting a path, accessing and changing the value of a variable in some other path, and making a copy of a path. The basic framework is provided by a distinguished path — the control interpreter. This is unique in two respects: (1) timer interrupts pass directly to it; (2) there is a control primitive — CIA — by which other paths can call for the execution of an arbitrary procedure in the environment of the control interpreter and wait for the result.

There is a program which runs in the control interpreter path and acts as the central control of ECL. Basically, its functions are to

handle I/O requests, arrange for running the other paths, and handle coordination between paths. This program is written in the language using the primitives mentioned above. For example, to perform path scheduling, a queue of runnable paths is maintained; when the timer interrupt comes into the control interpreter, the path that was running is put at the end of the queue, a new path is chosen from the runnable queue by the scheduler, and the start-path primitive is executed to run the new path. The scheduler is also a routine written in the language. Currently, it simply chooses paths in FIFO order. However, the programmer may redefine the scheduler by substituting his own routine for the system-provided one. Hence, such refinements as a priority system, either simple or with dynamically changing priorities, can be readily added.

Other control activities are equally easy to program. For example, a Dijkstra semaphore is a language-defined data structure consisting of an integer count and a queue of paths (also a defined data type) waiting on this semaphore. The P and V operations are implemented by using CIA primitive to transfer into the environment of the control interpreter where the necessary queues can be safely modified.

With the framework provided, it is straightforward to implement most of the known control structures, e.g.,co-routines, multiple parallel returns, cooperating sequential processes and fork/join structures. Further, since ECL leaves its control structures open to change, it will be possible to develop, as needed, a variety of other control regimes.

SECTION IV

SUMMARY

The ECL programming system has been designed to provide an environment conducive to effective programming. To this end, it contains a language with comprehensive data types, operators, control structures, and storage management facilities. It allows interactive program composition and debugging with smooth transition to efficient compiled code. Most important, it allows the programmer to tailor this environment to suit his needs.

REFERENCES

[1] B. Wegbreit, The Treatment of Data Types in EL1, Technical Report 4-71, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, May 1971.

[2] B. Wegbreit, Compactifying Garbage Collection in the Heap, Technical Report 5-71, Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, June 1971.

[3] B. Wegbreit, Studies in Extensible Programming Languages, ESD-TR-70-297, Harvard University, Cambridge, Massachusetts, May 1970.

[4] F. L. de Remer, Practical Translators for LR(k) Languages, Ph. D. thesis, Electrical Engineering Department MIT, Cambridge, Massachusetts, October 1969.

[5] IBM, APL/360 User's Manual, GH 20-0683-1.

[6] A. van Wijngaarden, et al., Report on the Algorithmic Language ALGOL 68, Mathematisch Centrum Amsterdam MR 101, February 1969.

[7] G. Radin and H. P. Rogway, Highlights of a New Programming Language, Communications of the ACM, Vol. 3, January 1965.

[8] P. S. Abrahams, et al., The LISP 2 Programming Language and System, FJCC, Vol. 29, 1966.

[9] E. W. Dijkstra, Co-operating Sequential Processes, in Programming Languages, edited by Genuys Academic Press, New York, 1968.

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 ECL PROGRAMMING SYSTEM

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 29	7b. NO. OF REFS 9
6a. CONTRACT OR GRANT NO. F19628-68-C-0379 b. PROJECT NO. c. ARPA Order No. 952 d.	9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-71-343 9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*	

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

ECL is a programming language system designed to provide an environment conducive to effective programming. It consists of a programming language and a system built around that language to serve as a tool and vehicle for program manufacture. The language contains comprehensive data types, operators, control structures and storage management. The system is oriented to interactive program composition and debugging with smooth transition to efficient compiled code. Most important, the system allows the programmer to tailor the environment to suit his needs.

DD FORM 1 NOV 65 1473

Unclassified
Security Classification

14. KEY WORDS	LINK A ROLE	LINK A WT	LINK B ROLE	LINK B WT	LINK C ROLE	LINK C WT
Programming language						
Programming system						
Extensible language						
Compilation						
Data type definition						
Control structures						
Syntax extension						
Storage management facilities						
Interactive programming						
Debugging						