ETC—An extendible macro-based compiler

by B. N. DICKMAN

Bell Telephone Laboratories
Whippany, New Jersey

INTRODUCTION

ETC (ExTendible Compiler) is a high level language compiler that allows the programmer to produce very efficient code when necessary, getting as close to the machine as he desires, and yet to write in machine independent statements when the production of optimized code is not necessary. The programmer may also easily extend ETC, creating new data types and operations either from previous extensions or from the machine operations (or both).

Extendible languages come in several forms. One type allows extensions only in terms of a relatively high level and complex base language. Another extendible language may try for a ‘minimal’ base language which is still machine independent, with the idea that if the base language is minimal, the definitions of the extensions become more comprehensible.

Many extendible languages depend on some macro facility for text substitution, but usually the facility is primitive, rigid, or not implemented. One language even has two separate macro languages. An extendible language could be extendible only in terms of machine independent statements, or it could be extendible ‘downward’, allowing the programmer to define new operations in terms of machine operations as well.

One test of the downward extendibility of a language is the degree to which the base language was implemented or could easily have been implemented (back to the machine language) using the primitives intended for extension of the language. This is not the same as bootstrapping, since it implies accessibility to the in- nards of the compiler by the user at any level of extension of the language. Furthermore, if one makes extensions carefully, the programmer who wishes to make changes to the language need not know anything about the implementation of the syntax ‘below him’ or how the syntax is described.

If one proceeds to extend the machine language using only the assembler, macro facilities, and a judiciously chosen set of primitives, then downward extendibility is ensured.

ETC is such a language. It is an extendible infix language compiler designed to produce very efficient code. ETC gives the programmer the advantages of using a high level language (e.g., relative ease of programming, documentation, and debugging) without the usual drawbacks (e.g., the production of inefficient code). One of the major aims of the language is to do as much ‘bookkeeping’ for the programmer as possible commensurate with the production of efficient code.

A second goal is to provide polymorphic operators over data types, that is, have the same operator operate on variables of different data types. Thirdly, it is desired to provide a language that can be incrementally implemented with relative ease.

An important bookkeeping feature of ETC is the limited automatic register allocation feature. A way is provided to divide the registers between explicit use by the programmer and implicit use by the compiler, without reserving certain registers for exclusive use by the compiler. Except for subroutine interfaces, there is virtually no need for the programmer to know the number of the hardware register being used. All that is important is the type of register. Thus ETC keeps a record of which registers are free and allocates them as appropriate. For most translators for high level languages implemented for machines with more than one register in which to do computations there is a problem of efficient allocation of the registers. Usually, memory-to-register and register-to-memory machine operations consume much more time than do register-to-register operations. Thus it is advantageous to retain the intermediate results of computations in registers as much as possible. If the number of intermediate results exceeds the number of registers, those results to be used most often should be given priority for remaining in registers.

The key phrase here is ‘most often.’ Even if a translator can tell which results occur the greatest number
of times in a program, the translator cannot tell which results will be used most often in the execution of that program. Even the programmer does not always know which paths will be executed most often. Of course, certain heuristics can be used, but efficient register allocation, and for that matter, generation of efficient code, is not a trivial matter. It is difficult to tell a language translator what space-time trade-off one wants to make. Of course, this does not mean that optimizing translators are impossible.

The solution here has been to give the programmer the ability to produce less-than-optimal code when programming constraints allow programming in a high level language, and yet give the programmer the means to produce code as optimized as he wishes without programming in machine language. The programmer may name and use the thirty-two machine registers just as memory locations, except for subscripting.

If the programmer runs out of registers, the translator outputs an error message and lets the programmer decide which results are most important. A programmer rarely runs out of registers. In practice, the programmer who knows nothing about the registers and does not explicitly use them never runs out of registers.

THE ENVIRONMENT

The SWAP assembler

ETC, the base language and the extended language, exists as a macro library written in the SWAP\textsuperscript{\textregistered} macro language. While the language extension primitives could be abstracted from the context of the SWAP assembler, many of the problems of other extendible languages do not exist in ETC, because ETC is based on SWAP. SWAP has assembly-time arrays and pushdown lists and allows the programmer easily to extend the assembler symbol table. The programmer can store and retrieve macro, symbol, and attribute libraries and can change the input stream while compiling. The macro facilities constitute a powerful set of text substitution functions.

Since ETC is implemented using the SWAP macro facilities, the label, commenting, and continuation conventions of ETC are identical to those of SWAP. The conventions are as follows:

Labels must start in column one. Machine operations, pseudo-operations, macro calls, or ETC statements start anywhere after column one, with one or more blanks or exactly one comma used to delimit arguments. Everything occurring after a "sharp sign", \#, is considered commentary. The "commercial at" sign, @, is used to denote continuation.

If @ is the last non-blank character on a card, the next card(s) starting with column one, is concatenated with the character preceding the @.

The machine

The machine for which the language was implemented is similar to the IBM 360. The differences are that there are no memory-to-memory machine operations; the memory-to-register and register-to-memory instructions only move data; the machine has a splendid set of true (immediate) instructions.

The term "labeled common" is used here to refer to read-write memory. Programs reside in read-only memory. Read-write memory is based, word addressed. A labeled common region may not overlap a physical unit of read-write memory, but several labeled common regions may be defined within one physical unit. Read-only memory is not based.

There are sixteen each of A-registers and B-registers. A-registers are used primarily for arithmetic operations; B-registers for indexing, basing, and logical operations.

THE EXTENDED LANGUAGE

General discussion

The application-oriented features will be described only insofar as necessary to give a basis for understanding the base language.

The extended language looks somewhat like FORTRAN except for the following: ETC statements may be freely mixed with pseudo-operations, machine operations, and macro calls; arrays may be only singly subscripted; and there is a facility for symbolically referring to parts of words.

Features include a DECLARE statement; DO and IF statements; limited automatic register allocation; and predefined functions such as FLOAT and SQRT. Other operations include plus, minus, divide, multiply, exponentiation, logical 'or', logical 'and'.

The metalanguage used to describe the syntax has the following conventions. Square brackets denote optional syntactic entities. In a definition the words in lower case or with embedded underscores are to be treated as metasymbols. That is, they represent the form of what is to replace them in an actual statement. All other words and special characters are to be coded as specified.

ETC uses the lexical token break-out of SWAP. This allows one to distinguish between assembly time
and run time operations where the same character is used for both, but it means that blanks have syntactic significance. In general, no blanks are allowed on both sides of assembly time operations, but at least one blank must be on either side of an operator generating machine code (operators defined by the operation defining attributes, OPERATION, BINOP and UNOP).

In ETC all registers which are explicitly referenced by the programmer must be declared symbolically. All labeled common regions must be based. This means that the base address of the labeled common region which contains the symbol must be in a register, and the labeled common region name and the register name must be tied together via a USING statement. The ETC BASE statement does all of this for the programmer.

The "type" and "value" of a symbol locate the contents of the symbol in the machine or denote it as a constant. For example, B-register one has type B and value one. Attributes, as specified in the DECLARE statement, may be either locative, as type and value, or descriptive, as signed and floating.

Example of the extended language

Problem: To search the ten word array LIST and put the maximum value in MAX.

Solution 1 (Register independent):

MAX = 0
DO I = 0 TO 9
   IF LIST(I) > MAX THEN MAX = LIST(I)
DOEND

CONCEPTS

The primitives, constituting the base language, are those features of ETC which are intended expressly for extending the language. They were used to implement ETC and are available to any programmer.

Extensions to the language are always made by the compiler to the "beginning" of the language, so that old definitions of operations, functions, or attributes may be modified, extended, or replaced completely.

Extensions to the language are generally local to the program in which they occur. To make extensions permanent, it is necessary to make them in the program defining ETC.

MAIN CONTROL

One might wonder how the assembler and ETC interact in terms of gaining control from each other. Essentially, if SWAP does not recognize a statement as a prefix operator with arguments, it is passed on to ETC via a redefinition of NONOP (the pseudo-operation called by SWAP for undefined operations) as a macro. This macro is the main driving routine. It does print-formatting and calls the scanner. The scanner is a macro whose definition is extended (always at the beginning) by OPERATION, BINOP (binary operation), UNOP (unary operation), and FUNCTION attribute declarations (and, of course, by any attribute defined by the programmer in terms of these attributes). Actually, BINOP, UNOP, and FUNCTION are all defined in terms of the OPERATION attribute. BINOP, UNOP, and FUNCTION also output information for use by the general expression parser, which OPERATION does not. Syntax determination of whether to call the parser was itself done as an OPERATION declaration.

The default on the data type of the result of an operation or function evaluation is the data type of the first operand or argument. Explicit specification to the parser of the resultant data type may be made in the implementation of the operation or function by an appropriate declaration of the ETC system symbol SYS_R_TYPE.

Defining operations

Several attributes may be used in a DECLARE statement to define operations.

OPERATION is the general operation defining attribute. Its form is

DCL operationname OPERATION(booleanexpression)

where

operationname is the name of the macro to be called if booleanexpression is true. The boolean expression may combine as much syntax and semantic analysis as the programmer desires. booleanexpression can be any SWAP macro language boolean expression but will usually be in terms of SYSLST(1) to SYSLST(m), where SYSLST(i) is the ith token in the source language statement.

OPERATION is generally used to define operations which are not binary or unary operations, to extend the definition of an existing operation to handle new data types, to improve the code generated from an existing definition, and to correct bugs in the code generation for an existing operation.

For instance, one might have the '+' operation implemented for fixed and floating point quantities and want to extend '+' to handle matrices. One would use the ATTRIBUTE attribute to introduce a MATRIX attribute and then specify an OPERATION
declaration which detects not only the syntax involved in the addition operation, but also the fact that matrices are being added. Then a macro is defined to do code generation for this specialized form of ‘+’.

Also, a programmer might discover a ‘trick’ by means of which a specific combination of operands and operations may be made to produce more efficient code than for the general case. He might specify an OPERATION declaration to detect this special case and write a macro to generate the code. No knowledge about the implementation of the operation, other than that of the specific case detected, would be required.

Finally, an OPERATION declaration may be used to detect syntax and semantics of the special case of an operation for which the code generation is incorrect and to extend the compiler to generate the correct code.

The UNOP and BINOP attributes are used specifically to define unary and binary operations.

Used in conjunction with UNOP and BINOP is the PRECEDENCE attribute. Its argument is the relative or absolute precedence to be given the operator.

If not explicitly specified, the precedence of an operator defaults to the currently highest precedence plus one.

The RIGHT_TO_LEFTSCAN attribute is used in conjunction with the BINOP attribute to indicate that the direction of scan of operators of equal precedence is to be from right to left. Default scan for BINOP is from left to right.

**Defining functions**

The form for the function attribute is

```
DCL name FUNCTION[(macroname)]
```

where `name` is the name of the function and also the name of the macro to be called for code generation, unless `macroname` is specified. The code generation macro is passed an argument list of form

```
variable = functionname(arguments)
```

and must perform the assignment to `variable` of the result of evaluating the function.

Example: A SIN function which calls a SIN subroutine expecting its argument and returning its result in A-register 0 might be coded as follows:

```
DCL SIN FUNCTION
MACRO
SIN
# SYSLST(3,1) IS THE ARGUMENT OF THE THIRD TOKEN, I.E.,
# THE ARGUMENT OF SIN.
A0 = SYSLST(3,1)
CALL SIN
# SYSLST(1) IS THE FIRST TOKEN, I.E., THE VARIABLE ON
# THE LEFT HAND SIDE OF THE EQUAL SIGN.
SYSLST(1) = A0
MEND
```

In general SYSLST(i) is a way of referencing parameters to a macro call. SYSLST(i) is local to the macro in which it appears. (Nested macro calls “push” SYSLST.)

The function may now be used in any expression.

For example:

```
X = Y + SIN(Z(1)) * 2.0
```

**Compile Time Bookkeeping**

**Attributes of Symbols**

**Symbol Table Equivalence: EQV**

Whereas the SET pseudo-operation will set only the value and type of one symbol to that of another, EQV may be used to copy the whole symbol table entry. This is especially useful when implementing an operation recursively, or when bootstrapping. Often one wants to give a register variable the same attributes as a memory variable.

Or suppose one has implemented an IF statement in a straightforward manner. Consider the comparison to less-than-zero of a signed variable fewer than 32 bits long whose high order bit occupies the high order bit of a word. (In ETC variables are allowed to be parts of words.) Our present implementation would probably unpack this variable from the word, propagate the sign, compare it to zero, and branch accordingly. In
most two's complement machines it would be simpler to test the whole word as being negative and branch accordingly without unpacking the variable from the word.

We might extend IF to detect this particular comparison; use EQV to define a new variable the same as the old, except make the new variable a full word variable; and recurse, expanding the IF macro with a 'less than zero' comparison.

The syntax of EQV is as follows:

```plaintext
sym1  EQV  sym2
```

where sym1 and sym2 are symbols, will set the value, type, and all attributes of sym1 to those of sym2. If sym2 is a constant, defaults for the attributes are used. Particular attributes of sym2 may be overridden by the use of a number of optional keyword parameters:

- \text{VALUE} = n \quad \text{sets the value of sym1 to } n.
- \text{STYP} = \text{char} \quad \text{sets the type of sym1 to char.}
- \text{SIZE} = n \quad \text{sets the size of sym1 to } n.

Example:

The following example, the implementation of the FUNCTION attribute, illustrates the use of the ATTRIBUTE attribute:

```plaintext
DCL FUNCTION ATTRIBUTE(SYSDFNFN)
MACRO
FN_NAME SYSDFNFN MACRO_NAME
  DCL IS('MACRO_NAME'--, =,MACRO_NAME)IFNOT(FN_NAME) OPERATION@
  (I.N.I.SYSLST =3&'I.SYSLST(2) , =' = '&'I.O.I.SYSLST(3) , = 'FN_NAME')
MEND
```

Explanation: If the optional macro name (MACRO_NAME) is supplied, it is to be used as the name of the code generation macro, otherwise, the function name is assumed to be the name of the code generation macro. In any case, FUNCTION is defined as an operation with exactly three tokens, where the second token is '=', and the 'operand' part of the third token is the name of the function. (The 'operand' part consists of everything in the token up to a left parenthesis.)

The argument of the DEFAULTS attribute is the name of a macro to be called to handle defaults. When used in conjunction with the ATTRIBUTE attribute, DEFAULTS associates the attribute with the name of a macro to be called by DECLARE after the last explicit attribute has been processed. The macro to be called is the one associated with the last explicitly specified attribute. Usually, the same macro will be specified for each attribute in a set of consistent attributes.

Register allocation and deallocation

ACTIVATE and DEACTIVATE are the problem programmer oriented register allocation pseudo-operations. ALLOC and DEALLOC are primitives essentially for allocating and deallocating "rock bottom" scratch registers. There is limited automatic register allocation. No "flow analysis" is done, so that the automatic register allocation mechanism generally is valid only for straight line segments of code i.e., where the only entry points or exit points are at the beginning or end of the segment.

Three attributes are necessary for data declaration purposes: REG where the specific numerical register...
designation is left to the automatic register allocation mechanism, and AQU and BQU for static allocation of registers.

The syntax for the register allocation attributes is as follows:

DCL name AQU(number)
DCL name BQU(number)
DCL name REG(type)

The third form would declare name to be a particular type (A or B) of register. It would not allocate a register. The first two forms specify both type and number of register.

The form for the allocation pseudo-operations is:

ACTIVATE list
DEACTIVATE list

The elements of list are register names, that is, they have been declared to have the AQU, BQU, or REG attribute. If the register to which the symbol refers previously has been statically allocated (i.e., through an AQU or BQU attribute), no allocation is done; the automatic allocation mechanism is merely notified that this register is free (DEACTIVATE) or that it is in use (ACTIVATE). The abbreviations ACT and DEACT are allowed.

If the register to which the symbol refers has been statically allocated and is being deactivated, the value of the symbol is "taken away". If the symbol has no value and is being activated, it is given a value by the automatic register allocation mechanism.

If the symbol has been declared using REG(A), ACTIVATE finds a free register, gives its value and type to the symbol, and marks the appropriate register busy. In this context a register is considered free if it is not marked busy (whether statically allocated or not). ACTIVATE assumes that its argument has a register type defined before ACTIVATE is called. ACTIVATE may be used in preference to ACTIVATE when defining an operation, since one is assured that no ACTIVATE of statically allocated registers will occur during the compilation of one statement.

DEACTIVATE is the inverse of ACTIVATE. It only erases the busy mark for the appropriate register.

The philosophy of register allocation

We make it necessary for the programmer to activate each register explicitly for several reasons: (1) At any point in the program one can easily see what registers are in use; (2) a statement is automatically flagged if a register not containing an expected value is used; (3) while the register is inactive, ETC can use it internally as a scratch register for one ETC statement; (4) subroutine interfacing, via passing arguments in registers, is made easier.

Notice that the use of many statically allocated
registers nullifies the usefulness of the automatic allocation feature. No register that has been statically allocated (i.e., through use of AQU or BQU) can also be dynamically allocated (i.e., through use of REG and ACTIVATE). This is true for the same reason that memory that has been statically allocated cannot also be dynamically allocated. It is advantageous to leave the specific register undesignated by using REG wherever possible.

The capability of automatic register allocation buys several advantages, but it also has one penalty: if it is to work effectively, it must have registers to allocate. Thus sparing use must be made of the static allocation attribute.

Generating diagnostics: DIAG

DIAG is a ETC primitive which is used to generate the diagnostics printed by ETC. DIAG will check the diagnostic level then in effect and, if the level of the diagnostic to be printed is not less than this diagnostic level, the diagnostic will be printed. DIAG also allows the user to prefix and suffix his message with standard phrases. Output of error flags is accomplished by specifying the flag in column one of the DIAG call. A typical DIAG call is

```
DIAG LEVNO=4,PREFIX =8,SUFFIX =1, SUBTRACTION
```

which will cause the following diagnostics to be printed if the diagnostic level is less than or equal to 4.

```
****MIXED MODE SUBTRACTION NOT IMPLEMENTED
```

The problem programmer may use the pseudo-operation DIAGNOSTIC_LEVEL to specify that no diagnostic less serious than that of a specified level be printed.

USING THE PRIMITIVES

An inner product operator

Suppose one wanted to define the inner product of two three-dimensional vectors:

\[ V_3 = V_1 \cdot V_2 \]

where \( V_1 \) and \( V_2 \) are each the first word of blocks of three words. One could write

```
DCL BINOP(DOT)
MACRO
  DOT V3 EQ V1 D V2
  V3 = V1:(0) * V2:(0) + V1:(1) * V2:(1) + V1:(2) * V2:(2)
#THE COLONS ABOVE INDICATE CONCATENATION. WITHOUT THE COLONS, #THE SUBSCRIPTED SYMBOLS WOULD BE TAKEN TO BE SUBPARAMETER #REFERENCING.
MEND
```

A square root function

```
#THIS IS THE DEFINITION OF A FUNCTION TO TAKE THE SQUARE #ROOT OF A VARIABLE
DCL SQRTF FUNCTION
MACRO
  SQRTF X EQ Y
#CALL A SUBROUTINE TO EVALUATE THE SQUARE ROOT #ASSUME IT TAKES TWO ARGUMENTS: LOCATION TO STORE RESULT #AND INPUT VALUE.
```
SQRT_SUB X,Y(1)
MEND

#NOW SUPPOSE WE HAVE A MACHINE
#OPERATION, FSQRT, WHICH TAKES TWO A-REGISTERS AS ARGUMENTS
# BUT WORKS ONLY FOR FLOATING VARIABLES.

MACRO
  SQRTF X EQ Y
  #IF FIXED, CALL THE OLD SUBROUTINE
  IS(FIXED(Y(1)), SQRT_SUB X,Y(1) ; JUMP .OUT)
  #IF X AND Y(1) ARE BOTH A-REGISTERS, GENERATE THE MACHINE OPERATION
  IS(`MTYP(X)`='A'='MTYP(Y(1))', FSQRT X,Y(1) ; JUMP .OUT)
  #IF X IS AN A-REGISTER, USE IT AS A SCRATCH REGISTER AND RECURSE
  IS(`MTYP(X)`='A', X = Y(1) ; X = SQRTF(X) ; JUMP .OUT)
  #OTHERWISE, ALLOCATE A SCRATCH REGISTER AND RECURSE
  TMP EQV O,STYP=A ; ALLOC TMP
  TMP = SQRTF(Y(1))
  X = TMP
  DEALLOC TMP
MEND

Complex arithmetic

We will define the attribute COMPLEX and extend the operations “+” and “-” to handle unsubscripted floating variables of type COMPLEX. The functions REALPART and IMAGPART will also be defined.

#THIS STATEMENT DEFINES THE ATTRIBUTE “COMPLEX” SYNTACTICALLY. IT SAYS “CALL THE MACRO “CMPLX” TO EVALUATE THE ATTRIBUTE “COMPLEX””.

DCL COMPLEX ATTRIBUTE(CMPLX)

#THIS MACRO PROVIDES THE SEMANTICS FOR THE “COMPLEX” ATTRIBUTE.

MACRO
  VARCOMPLX
  DCL VAR BSS(2) FLOAT
  #THE FOLLOWING STATEMENT MAKES AN ENTRY INTO THE SYMBOL TABLE DEFINING THE “COMPLEX” ATTRIBUTE OF VAR TO HAVE VALUE ONE.
  #THIS KIND OF STATEMENT ALSO DEFINES THE ATTRIBUTE FOR ALL SYMBOLS IN THE SYMBOL TABLE. UP TO 255 ATTRIBUTES ARE POSSIBLE (WITHOUT PACKING).
  COMPLEX(VAR) SET 1
MEND

#THIS STATEMENT EXTENDS THE COMPILER TO RECOGNIZE THE ADDITION OR SUBTRACTION OF TWO COMPLEX VARIABLES.

DCL CMPLXOP OPERATION(N,SYSLST = 5&ANY(`SYSLST(4)’,”+”,”-”)& &COMPLEX(SYSLST(1))=COMPLEX(SYSLST(3))=COMPLEX(SYSLST(5))=1)

#THIS MACRO DEFINES THE SEMANTICS OF ADDITION AND SUBTRACTION OF COMPLEX VARIABLES.

MACRO
  CMPLXOP X EQ Y OP Z
# THESE THREE COMPILE TIME STATEMENTS DEFINE THREE REAL VARIABLES OCCUPYING THE FIRST WORDS OF X, Y, AND Z
TMPX EQV X, COMPLEX = 0; TMPY EQV Y, COMPLEX = 0; TMPZ EQV Z, COMPLEX = 0

# TMPX = TMPY OP TMPZ

# THESE THREE STATEMENTS DEFINE THREE REAL VARIABLES (REDEFINE TMPX, TMPY, AND TMPZ) OCCUPYING THE SECOND WORDS OF X, Y, AND Z.
TMPX EQV TMPX, VALUE = TMPX + 1
TMPY EQV TMPY, VALUE = TMPY + 1
TMPZ EQV TMPZ, VALUE = TMPX + 1

TMPX = TMPY OP TMPZ
MEND
DCL REALPART FUNCTION
DCL IMAGPART FUNCTION
MACRO
REALPART A EQ FNCALL
# FNCALL(l) IS THE FIRST SUBARGUMENT OF FNCALL. HERE FNCALL(1) IS THE ARGUMENT OF THE FUNCTION REALPART, SINCE THE REALPART MACRO IS FED THE FUNCTION CALL AS PART OF AN ASSIGNMENT STATEMENT.
TMPX EQV FNCALL(1), COMPLEX = 0

A = TMPX
MEND
MACRO
IMAGPART A EQ FNCALL

TMPX EQV FNCALL(1), COMPLEX = 0, VALUE = FNCALL(1) + 1

A = TMPX
MEND
DCL CMPLXASSIGN OPERATION(N.SYSLST = 3& COMPLEX(SYSLST(1)) = 1&@ COMPLEX(SYSLST(3)) = 1)
MACRO
CMPLXASSIGN X EQ Y

TMPX EQV A, COMPLEX = 0
TMPY EQV Y, COMPLEX = 0

TMPX = TMPY
TMPX EQV TMPX, VALUE = TMPX + 1
TMPY EQV TMPY, VALUE = TMPY + 1

TMPX = TMPY
MEND

DISCUSSION

Downward extendibility probably limits the resultant language to the same structure as that of the assembly language (if the assembler does not have block structure, it will be very difficult to graft this onto the extended language). But perhaps this is a relationship shared by any base language and its extensions.

Code optimization on a global (i.e., more than one statement) scale is very difficult, if not impossible. However, on a statement-by-statement level one can produce very efficient code relatively easily. Allowing the programmer access to machine registers and machine operations facilitates further optimization.
In test cases coded both in ETC and machine language, ETC generated less than ten percent more code than the machine language programs with substantially less coding time for the ETC programs.

The extendibility of the language has proved to be especially useful on a large project with stringent programming schedules where it was not clear in advance what features the particular language should have. Programming began in assembly language and parts of the extended language were used as implemented.

Finally, extendibility makes maintenance of the compiler simpler. It is not necessary to know anything about the implementation of an operation in order to correct a bug in code generation. The language may easily be extended to detect the particular case for which wrong code is generated and to generate the correct code.

ACKNOWLEDGMENTS

Mr. C. B. Hedrick, Miss V. L. Bishop, Mr. D. J. Desmond, Mrs. I. T. Pelak, and Mr. D. J. Sidor have contributed to the implementation of the language. The author would like to thank Messrs. D. Bzowy, R. R. Conners, P. D. Jensen, and J. S. Walden for many helpful discussions.

REFERENCES

1 J NICHOLLS
   PL/1 compile time extensibility
   SIGPLAN Notices Boston Mass Vol 4 No 8 pp 40-44 1969
2 B J MAILLOUX  J E L PECK
   Algol 68 as an extensible language
3 W M WAITE
   A Language independent macro processor
   Comm ACM Vol 10 No 9 pp 433-440 1967
4 A L SPRINGER
   Extensible languages
   Notes for Advanced Topics in Systems Programming
   University of Michigan Ann Arbor Michigan 1970
5 M E BARTON
   The macro assembler, SWAP
   Proceedings of the Fall Joint Computer Conference AFIPS
   Press Montvale New Jersey Volume 37 pp 1-8 1970