Toward the development of machine-independent systems programming languages

by KEN MAGEL, ANDRIES VAN DAM and MARTIN MICHEL

Brown University
Providence, Rhode Island

INTRODUCTION

One of the reasons for developing high level languages has been the desire for program portability from one type of machine to another. To achieve the high degree of machine independence necessary for program portability, these languages have included general features such as arithmetic expressions, arrays, and subroutine calls which can be implemented on many machines. Facilities such as the interrupt mechanism, program status word, and device dependent input/output which are available to the assembly language programmer are hidden from the high level programmer. Unfortunately, the code generated by compilers for these high level languages is often very inefficient compared to that produced by experienced assembly language programmers. However, the added expressiveness and the ability to leave details to the compiler usually offset the inefficiency of generated code, particularly for non-systems applications. For such applications, the facilities which the high level programmer cannot use are not needed anyway.

Systems programming, on the other hand, is one area in which the inefficiency of generated code and the inability to access some features of the machine or operating system make the use of high level languages for entire systems currently extremely difficult. Yet, if a machine and operating system independent systems programming language could be developed which permitted the user to access all the facilities of each machine in a natural manner, systems programs could be transported easily from one environment to another. For example, dispatchers, scheduling algorithms, and resource allocation routines could be developed on one machine and then moved to others instead of having to be redeveloped for each environment. Furthermore, systems written in such a language could be studied, and perhaps even debugged without knowing the intricacies and idiosyncrasies of the hardware. Previously developed high level languages especially designed for systems programming such as ECPL and BLISS only partially remove these liabilities. The user still cannot access all the facilities of the hardware in a natural, machine independent manner. Lower level languages such as PL/360 do not solve the problem either because they are too tied to the architecture of one machine. Macro generators such as STAGE2 do solve the problem, but require a complete rewrite of each macro when moving from one environment to another. Also optimization across macros which might be possible in some environments cannot be performed. This paper describes various approaches which have been taken to developing machine independent systems programming languages. It then explains why none of these provides the full answer. As an example of each approach, certain features of the language for Systems Development, a PL/I derivative with improved control and data structures, basing modifiers, and string operations, are described. LSD is expressly designed for implementing operating systems, compilers, and other system programs, as well as for large general purpose applications. A prototype compiler for a large subset of the language has been in use for more than two years and generates very efficient code. Presently the full language is being implemented with code generation facilities for the IBM /360-67, /370-168, and two user microprogrammed minicomputers with very different instruction sets (DSC Meta 4's—one programmed as a general purpose machine and the other as a graphics processor). All four machines have specialized features which must be utilized in writing efficient operating systems and in certain other applications.

* The research described in this paper is supported by the National Science Foundation, Grant GJ-28401X, the Office of Naval Research, Contract N00014-67-A-0181-0025, and the Brown University Division of Applied Mathematics.
* Very sophisticated and complex optimizing compilers can sometimes partially improve the quality of generated code; in addition, machines designed specifically for high level languages can somewhat reduce the disparity.

† Very sophisticated and costly macro generators can do some cross-macro optimizations by saving needed information in global variables. ‡ Between 10 percent and 140 percent worse than very carefully hand-coded assembly language depending on the application. These results are from a benchmark study done for the Safeguard Systems Command in 1971. Even the prototype implementation without any sophisticated optimization performed considerably better than either IBM's PL/I-F or Fortran IV H.
PREVIOUS APPROACHES

Traditionally, two general approaches have been taken to the problem of providing access to all of the facilities available in assembly language, but within a high level language: (1) the inclusion of assembly language code for a specific machine within the source program, and (2) the design of high level facilities which are sufficiently general to be meaningful on many machines, but which can be efficiently implemented on each target machine. Although both methods were provided in the first version of LSD, each had serious liabilities.

The ability to include assembly language code within source programs is provided in LSD by two constructs: Code/Endcode for the inclusion of instructions, and Data/Enddata for the inclusion of static data and variables. Items within Data/Enddata are allocated by the compiler in the area generated for static variables of the enclosing block (Begin or Proc) whereas statements within Code/Endcode are placed in the compiler output at a point corresponding to where they appear in the source program. A number of aids are provided by the compiler as illustrated in the example below. For instance, the user may ask the compiler to supply registers in instructions, and he may use names longer than those permitted by the assembler. Qualified names may be used within the assembly code when referring to variables within structures, and array subscripts (constants only) may be used to refer to specific elements of arrays. Code placed within Code/Endcode goes through the full compiler optimization phase* while Data/Enddata is not optimized. Variables used within either of these constructs can be local to the constructs or the same as those referenced in normal LSD statements. An example of the assembler code a programmer could write would be:**

```
CODE:
  L   R7!, T(2)  place T(2) in a register
  A   R7!, A . B add B to T(2)
ENDCODE:
```

where T is an array of full words, and B is a full word subelement of a structure named A. A . B is a qualified reference to B, indicating that B is a part of a structure A. The ! is used before the subscript (2) to distinguish the subscript from an index or basename part of the instruction’s second operand. The R7! indicates that the compiler should supply a general purpose register. The number after the question mark is used in determining which uses of question mark registers are meant to be the same register. Given the availability of general register 3, the compiler would treat the above code as if it were:

```
L   R3, T + 4  place T(2) in a register
A   R3, A + 12 add B to T(2)
```

if twelve were the displacement into the structure A of the substructure B.

* Unless the user specifies that this is not desirable.
** Those unfamiliar with 08/360 assembly language should consult the Appendix.

The inclusion of (even enhanced) assembly language for each machine is tantamount to admitting the impossibility of using high level constructs for all programming. In certain critical portions of the code, the user is forced to program and to think in terms of the low level facilities typical of assembly languages. If the program or the system of which it is a part is to be moved to another machine, the portions of programs which consist of embedded assembly language must be located and rewritten. Often these sections cannot be translated efficiently from one machine’s assembly language to another. What is really needed is a rewrite of larger portions of each program.

If there were not some uses which seem to require the Code/Endcode and Data/Enddata constructs, we would eliminate them. However, as will be seen from the discussion of the Convention’s statement below, now we do restrict assembly language to macros whose definitions are separate from the program itself.

The other approach is to design general facilities which can be mapped into the special features of the various machines. The design of high level facilities which can incorporate special features of arbitrary machines in a natural manner is extremely difficult, if not impossible. A very sophisticated compiler must be used to try to map general constructs into specific instructions of the target machine and/or specific features of the operating system. For example, IBM System/360 machines have a number of privileged instructions to permit the authorized assembly language programmer to manipulate the program status word, the storage protect keys, and the hardware I/O mechanisms. The first version of LSD included fourteen statements, one for each of these special instructions. These statements could not be implemented for other machines. There seemed no way to provide general high level constructs which captured the power of these instructions in an easily understood manner. In addition, the general constructs should not provide facilities which cannot be provided easily on a System/360. The fourteen statements have been removed from LSD.

THE CURRENT LSD APPROACH

Since neither of the traditional approaches seemed to satisfy our need for an environment independent systems programming language, a different approach seemed necessary. If the code containing environment dependent features could be easily isolated, it could be included within macros. The macros would have to be rewritten for each environment, but if their use were suitably constrained, movement from one environment to another would involve little rewriting and would not affect the appearance and logic of any of the programs. To guarantee that all the environment dependent code was in the macros, the user would have to be able to prevent the compiler from generating any environment dependent code for other constructs. A general macro replacement facility was not developed because it left the decision as to what code would be environment dependent to the individual programmer when he was writing the individual
routine. These decisions should be controlled by the project manager or the project design (see the discussion of advantages below).

Accordingly a small number of high level constructs which are present in almost all programming languages were selected. Knowledgeable users can specify alternative implementations for each of the selected constructs. Users can then indicate where each alternative implementation should be used.

The user supplies these alternatives by writing macros which expand into either LSD statements, the assembly language of the machine for which implementation is intended, or a combination of the two. The compiler provides defaults for all of the constructs.*

There are three major areas in the implementation of high level languages where the compiler-generated code must interface with the machine and the operating system. These are input/output, module linkages, and dynamic storage allocation. LSD provides only the most rudimentary input/output.** The high level constructs involved in module linkage are procedure entering, exiting, and returning. Those involved in dynamic storage allocation and release are block (PROC and BEGIN) entry and exit, and the management of based variables (the Allocate and Free statements). Procedure entering, exiting, and calling and the allocation and release of based variables were selected as the subset for which the knowledgeable user will be able to supply alternative implementations. A major advantage of including procedure exit and entry is that anything which could conceivably involve special features of a machine or operating system can be logically programmed as a procedure call. Note that the alternative code which the user supplies can do the function inline, if desired.

THE LSD IMPLEMENTATION

The Conventions statement is used to specify alternative implementations of the linkage and storage management constructs. The syntax is basically that of a macro. The traditional macro facility has two parts: a definition of the macro, and an invocation. Our method involves three parts: (1) the Conventions statement which defines the macro, (2) declarations of the procedures or variables involved which indicate which macros, if any, are to be applied, and (3) the Call or Allocate and Free statements which actually invoke the relevant macro.

The Conventions statement has a labeled header giving the symbolic parameters, followed by a list of parameterized LSD or assembly statements and an End statement. The label on the header is the name of the Conventions statement. As many Conventions statements as desired may be included in a program.

The user specifies where he wants the statement list to be used instead of code normally generated by the compiler by using the Conventions option on Procedure or Declaration statement.‡ For example, to have the Conventions code generated instead of that normally generated for Procedure entry, a Procedure statement of the following form would be used:

\[
\text{ADDTC: PROC(LIST, ELEMENT) CONV(TRANS)};
\]

Here ADDTO is the name of the procedure, LIST and ELEMENT are parameters, and TRANS is the name of a Conventions statement.

To also have different Conventions code generated instead of the normal returning code from this procedure, the following statement would be used:

\[
\text{ADDTC: PROC(LIST, ELEMENT) CONV(TRANS) RETURN(CONV(BACK))};
\]

TRANS is the name of a Conventions statement which would be used in generating the entrance code for ADDTO. BACK is the name of a Conventions statement which would be used in generating the exit code for ADDTO. In a procedure calling ADDTO a declaration of ADDTO could be used to specify Conventions for calling and after the return from ADDTO:

\[
\text{DCL ADDTO ENTRY CONV(GOIN) RETURN (CCNV(CUT))};
\]

The keyword ENTRY indicates that ADDTO is the name of a procedure or entry point. When ADDTO is called, the statement list in GOIN will be used instead of normal calling code. Immediately following this code will appear the code for the statement list in OUT. As an example consider the following:

\[
\text{RAND: PROC;}
\]
\[
\text{DCI Z ENTRY CCNV(CONE)};
\]
\[
\text{CALL Z(B, C, D)};
\]
\[
\text{CONV P(Q, R, S, T)};
\]
\[
\text{IF Q > R}
\]
\[
\text{THEN R = R + S;}
\]
\[
\text{IF S > T}
\]
\[
\text{THEN T = T + S;}
\]

* We could have used an extensibility mechanism to provide "new" constructs which would be defined differently in each environment. This is the approach taken by SPECIL. Our macro facility is almost identical to Udin's. In our case, however, the decision on when to invoke the macro is not made at the point of invocation.

** All I/O will probably be done in other languages with the LSD routines calling these other routines. I/O is not included in LSD because it is so closely tied to the environment in which a program runs.

‡ The statement list is substituted during the syntactic phase of the compiler. The substituted code thus goes through all the compiler optimization phases.

§ Conventions invocations cannot be nested so use of calls or allocates and frees of based variables within the statement list generates the default compiler code in all cases.

¶ The syntax of the LSD Procedure statement is almost identical to the corresponding Pl/I syntax. The Appendix gives a brief discussion of the portion of LSD used in this paper.
From the collection of the Computer History Museum (www.computerhistory.org)

The call of Z above
CALL Z(B, C, D);
would be replaced by
IF B>C
    THEN C = C + D;
CODE;
    L R3, B
    L R4, C
    L R5, D
    LR R2, R13
    LA R13, SAVM
    L R1, = V(P)
    BALR R14, R1
ENDDCODE;
END; "END of CONVENTIONS"

The call of Z above
CALL Z(B, C, D);
would be replaced by
IF B>C
    THEN C = C + D;
CODE;
    L R3, B
    L R4, C
    L R5, D
    LR R2, R13
    LA R13, SAVM
    L R1, = V(Z)
    BALR R14, R1
ENDDCODE;

The call of Z above
CALL Z(B, C, D);
would be replaced by
IF B>C
    THEN C = C + D;
CODE;
    L R3, B
    L R4, C
    L R5, D
    LR R2, R13
    LA R13, SAVM
    L R1, = V(Z)
    BALR R14, R1
ENDDCODE;

LSD provides a number of compile-time facilities to aid in
writing Conventions statements. The omission of any state­
ments in the statement list which involve parameters not
present in the invocation of the Conventions statement (the
Call statement) generalizes to the omission of an entire
IF-THEN-ELSE, if any part of it uses a parameter which is
not present, and to the elimination of entire Dc loops if an
absent parameter is mentioned in the Do header. Compiler
generated temporaries may be explicitly referenced by &Tn
where n is a positive integer. The compiler will generate a
temporary for each &Tn appearing in the expansion of the
Conventions statement. &Tn's using the same n will refer to
the same temporary. Special compile-time functions are
available for using a variable's length, data type and storage
alignment to control what code is generated. For example a
compile-time IF, written %IF, can be used to indicate that
the code within the THEN clause should be generated only
if the length of the variable is a full word and the alignment is
at least a half word boundary.

The Conventions construct can also be used for allocating
and releasing based variables. For this purpose, any of four
options may be specified on the declaration statement for a
variable. The ALLOC and FREE options specify Conven­
tions statements which are to be used instead of normal
allocating or releasing code when the variable is used in an
Allocate or Free statement. The ALLCCIN and FREEIN
options can be used on declarations of area variables. These
indicate Conventions statements which are to be used
whenever a variable is allocated or released within the area.
Whenever more than one of these options applies (for example
when allocating a variable with an ALLOC option in an area
with an ALLOCIN), the ALLOCIN or FREEIN is used. If a
list of variables is allocated or released in a single statement,
the Conventions code is used once for each variable to which
it is applicable. Normal compiler allocating or releasing code
is used for other variables in the list. The parameters on
Conventions statements used for based variables are the area
and its basing parameters followed by the variable and its
basing parameters.

Conventions statements for based variable management
are used primarily in interfacing with the dynamic storage
management routines. The next section deals only with the
Conventions statements used with procedures since the use of
Conventions statements with based variables is very similar.

THE ADVANTAGES

Using the Conventions statement the programmer is able to
to completely specify the environment in which his program
operates. If he wishes, he may leave some of the environment
to the compiler and specify only what is necessary for his
program to interface with either the operating system or
other portions of the system he is implementing. A system
manager can, by placing Conventions statements and declara­
tions in Include files* syntax insure that system conventions
are observed. This is one of the advantages of having the
decision when and which Conventions to apply at a different
place (declarations of variables or procedures) than the
actual use.

Conventions statements can be used to supply debugging
information and parameter checking which can easily be
removed when no longer needed. Coroutines, multitasking,
backtracking, and even more general control paths can easily
be implemented in an operating system independent manner.
Top down implementations in the Dijkstra sense can use
Conventions statements to have calls to lower level routines
or functions initially generate messages and parameter
checks. When the lower level routines become available, the
Conventions statements can be removed from the declarations
of the routines (in which case the Call statement actually does
a call) or changed to macros which perform the functions
inline. Functions can be selected logically and a decision to
implement them as subroutines or as inline code can be
delayed until later. The statement lists within the Conven­
tions statements can be tailored to idiosyncrasies of the
machine or operating system.

Using these facilities, some degree of portability can be
achieved. Those functions which on a particular machine or

* Include files are external program-segments which the compiler will
insert (logically) at requested points in a program before syntax analysis.
under a particular operating system use special facilities do so only through Conventions statements. Experienced programmers can design, code, and debug Conventions statements for each such function. The programs being written for the system then use Calls to "routines" to perform the functions (thereby invoking the associated Convention statements instead of the normal calling code). The Conventions statement either links to (or calls) a routine to perform the function or does the function inline.

**SOME CONCLUSIONS**

The Conventions facility in LSD isolates the environment dependent code. When movement to another environment is considered, the Conventions statements and the declarations indicating where they are used can be examined to determine if the movement is possible and how difficult it will be. The user, if he wishes, can, by specifying Conventions, prevent the compiler from generating any environment dependent code of its own.

We have not been able to eliminate assembly language altogether, but we have moved it out of the programs and into macros within Include files. The logic and flow of a program can be studied and understood without recourse to any environment dependent code.

It is still something of an art to determine which features should be implemented by Conventions statements. The main advantage of the LSD approach is that decisions on what these features should be need not be made until system implementation time and can be made differently for each system. A user wanting to use a program or system implemented for one machine or operating system in a different environment can examine the Conventions associated with the system to determine if transport to his environment is feasible.

The multi-machine implementation of LSD should be available sometime in 1974. We shall then test the practicality of system portability, possibly by trying to move an interactive graphics system implemented under CP-67/CMS on the /360-67 to one of the minicomputers.

**APPENDIX—OS/360 ASSEMBLY LANGUAGE**

The general form of a statement is

\[(\text{text})\](opt label) (opcode) (first operand), (second operand) (opt comment)\]

Opt indicates that the field is optional. The label, if one is present, must begin in column one. One or more blanks must separate the label, opcode, operands, and comment from each other, and no blanks can appear within the operands except in character string literals. Each operand has the general form:

\[(\text{name})\](opt base register), (opt index register)\]

If the base and index registers are not present, the parentheses and the comma do not appear. When an operand is a register, neither the base nor the index can be specified.

The instructions used in the examples are:

\[A\ (\text{first operand}), (\text{second operand})\]

The first operand must be a register and the second a main storage location. The second operand's contents are added to the first operand.

\[L\ (\text{register}), (\text{second operand})\]

The second operand must be a storage location. Its contents are placed in the register.

\[LR\ (\text{register}), (\text{register})\]

The second operand's contents are placed in the first operand.

\[LA\ (\text{register}), (\text{second operand})\]

The address of the second operand is placed in the first operand.

\[BALR\ (\text{register}), (\text{register})\]

The first register is loaded with the address of the next instruction following the BALR, and control is transferred to the address in the second register.

**LSD**

The procedure statement has the following general form:

\[\langle\text{proc name}\rangle: \text{PROC} (\text{opt parameter list}) (\text{opt modifiers})\]

where the BNF for the portions used in this paper is:

\[\langle\text{parameter list}\rangle : : = (\text{parm list})\]

\[\langle\text{parm list}\rangle : : = \langle\text{variable}\rangle, \langle\text{variable}\rangle, (\text{parm list})\]

\[\langle\text{convention modifier}\rangle : : = \text{CONV} (\text{name of Conventions})\]

\[\langle\text{return modifier}\rangle : : = \text{RETURN} (\text{CONV}(\text{name of Conventions}))\]

The Declare statement has the following form:

\[\langle\text{DCL}\ (\text{name}) (\text{opt attribute list})\rangle;\]

The BNF for the only option used in this paper is:

\[\langle\text{entry attribute}\rangle : : = \text{ENTRY} (\text{opt parameter list})\]

The call statement has the following form:

\[\langle\text{CALL}\ (\text{procedure or entry point name}) (\text{opt call list})\rangle;\]

The (call list) is the same as a parameter list except that each parameter can be an arbitrary expression.

The Conventions statement has a header of the form:

\[\langle\text{conventions name}\rangle: \text{CCNV} (\text{routine parm}) (\text{opt argument parms})\]
where (routine parm) is a simple variable, and (argument parms) is a parameter list.

For a description of the entire language, see Reference 3.

REFERENCES