A programming language for high-level architecture

by YAOHAN CHU
University of Maryland
College Park, Maryland

and

EDWARD RAY CANNON
International Computing Company
Bethesda, Maryland

The machine language of a computer is the programming language that the bare hardware can accept and interpret. In a von Neumann architecture, it is essentially a set of machine instructions and data formats. In a high-level computer architecture, the machine language is a high-level programming language since the hardware high-level architecture accepts and interprets this high-level language. Therefore, the programming language for a high-level computer architecture is the high-level machine language. For this reason, the programming language to be presented in this paper is called the HLM language.

Since the constructs of a high-level machine language affect intimately the constructs of the interpreting high-level computer architecture, the design of a high-level architecture begins with the design of a high-level machine language. This paper presents the design of a high-level machine language, while a separate paper describes the high-level architecture which implements this high-level machine language.

DESIGN CONSIDERATIONS

Before the HLM language is described, major design considerations that result in the choices of the particular data types, structures, operations and constructs are presented. These considerations serve as guidelines for the language design and are as follows:

a. The overall consideration is understandability and simplicity of the language.
b. The HLM language should have adequate language constructs for writing programs in applications of system programming, data processing, scientific computing and process control, since it is a programming language.
c. The HLM language should have language constructs that typical programmers can understand well so that they can effectively use the language.
d. The HLM language should have language constructs that facilitate the writing of reliable programs.
e. The HLM language should have language constructs that help in creating a simple and clean interactive direct-execution architecture for its implementation.
f. The HLM language should not be over-designed merely for the sake of seeking power and elegance of the language.
g. The HLM language should have adequate language constructs for writing high-level software, since the high-level architecture may have software. For example, the HLM language may be used to write an interpreter for a high-level or a very-high-level language.
h. The HLM language is designed with a particular regard to microprocessor implementation.
i. The HLM language may become a family of programming languages. It will be a family because member languages will have a uniform syntax and a similar structure, but with a different choice of data types. Each member language will be simpler to implement since it will meet a limited need, yet it will be adequate for its programming purpose.

In case of a conflict among the previous considerations, a compromise is guided by the overall consideration.

PROGRAM

A program declares data and specifies data operations for data processing or computing. To specify complex operations, sequences of data operations are needed. Control operations are needed to sequence the data operations. These data operations and their sequencing in a program create the data flow and the control flow, respectively, of the program. This section introduces the concept of data flow and control flow of a program, and then describes the program elements and program structure of the HLM language.

Data flow and control flow

In any program, there are two flows—the data flow and the control flow. The data flow is the flow of the data
changes in the program as a result of data operations; it is described by data flow statements. The control flow is the flow of data operations sequences when the program is being executed; it is described by control flow statements. Examples of a data flow statement and a control flow statement are the assignment statement and the “if” statement, respectively.

In the HLM language, the data flow and control flow are organized so that all data flow statements are embedded in control flow statements. In this way, the order of executing the data flow statements is entirely directed by the control flow statements. Visibility of both the data flow and control flow to the HLM language programmer is one of the unique language features of the HLM language; this helps the programmer to understand program execution better.

Program elements

The program elements of a high-level language are those constituents which make up a viable program. The program elements of the HLM language are:

a. Macro definition
b. Data declaration
c. Procedure definition
d. Control flow statement
e. Data flow statement
f. Comment statement

Macro definitions describe the macros of a HLM program. Each macro definition specifies some program text for substitution when the macro name is later called. An example is shown in Figure 1 where macro x denotes "123," and macro exchange(y,z) denotes "y := y + z."

Data declarations declare the names of the data storages together with their types and structures. An example is shown in Figure 1 where buffer i of data-type number of two digits, buffer j of data-type string of 10 characters, buffer m of data-type status of 'TRUE,' 'FALSE,' and 'don't know,' and array k with 10 array elements of data-type number of two digits are declared.

Procedure definitions describe the procedures of a HLM language program. Each procedure consists of control flow statements and optionally local data declarations. An example is shown in Figure 1 where Procedures a and b are defined. Procedure a has three control flow statements but no parameters. Procedure b has buffer x of data-type number of two digits as the parameter which is called by reference. It has only one control flow statement.

Data flow statements specify data operations. Control flow statements specify the data flow statements to be next executed. Two types of control flow statements are shown in Figure 1—BLOCK and CALL. There are three BLOCK statements, each specifying one or more assignment statements to be executed. There is one CALL statement which calls Procedure b.

The comment statement provides an explanatory remark for improving readability and documentation. It is a string of characters from the character set enclosed by a pair of symbols, /* and */, and may appear anywhere a space character can except inside of a character string. An example of the HLM program is also shown in Figure 1. A more detailed example is shown in Reference 9.

Program structure

The program structure of the HLM language is chosen to be simple because simplicity is an overall design consideration. As illustrated in Figure 2, a HLM language program consists of macro definitions, data declarations, procedure definitions, and a statement to indicate the program end. An example of program structure is also shown in Figure 1. The order to the appearance of these program elements may be changed as long as a name is declared before it is referenced. The first procedure definition is the main procedure where program execution begins. The other procedures are to be called by the main procedure directly or indirectly. (A procedure is called indirectly if it is called through more than one procedure calls.)

Data

In designing a programming language, there are several aspects of data that need be considered—data types, data structures, data operations and data flow. In addition, there are control data types, control data structures and control data operations.

---

/*This is an example of the HLM language program*/
MACRO x=123 ENDM;
MACRO exchange(y,z)=y:=y+z ENDM;
BUFFER i OF NUMBER[2],
    j OF STRING[10],
    m OF STATUS[TRUE,FALSE,don't know];
ARRAY k[10] OF NUMBER[2];
PROCEDURE a:
    BLOCK i:=INPUT(0); ENDB;
    CALL b(i);
    BLOCK OUTPUT(0):=i; ENDB;
ENDP a;
PROCEDURE b(BUFFER x OF NUMBER[2]):
    BLOCK x:=x*8; ENDB;
ENDP b;
ENDPROGRAM;

Figure 1—An example showing program elements and program structure.

Figure 2—Program structure for a HLM language program.
Data types are the primitive elements which may be numerical, physical, or others that are chosen to represent the data. They determine the scope of the language to describe the data. Data structures are the program elements which actually contain the data; they offer the facility in the language to access the data. Data operations are those operations which operate on the data types and data structures. Merely providing a data type or a data structure in a programming language without an adequate provision of data operations does not make that data type or that data structure particularly useful. The "programmability" of programming language is often limited by the data operations available for the data types and data structures. Data flow refers to the changes of the values in the data structures. Control data types and control data operations refer to those data types and data operations that affect the control flow during the program execution. These aspects of the HLM language are discussed next.

Data types and operations

There are many data types in current programming languages. Examples are data types of real and label in Algol 60, fixed and floating-point numbers in Fortran, record in Cobol, printers and files in PL/I, integer and string in SIMPL-T and byte and address in PL/M. In Pascal, data types may also be defined by the programmer. In the HLM language, five data types are chosen—decimal number, character string, byte string, time and status.

Decimal numbers represent numerical objects. Character strings represent symbolic objects. A byte string represents a bit string; however, a byte string is restricted to bit strings of multiples of eight-bit lengths. Time represents real time from which relative time, incremental time and delay can be derived. Status represents the condition after a test. A special case for status is the boolean status, which has the values of TRUE and FALSE.

In the HLM language, data operations for the five data types are shown in Table I. The five arithmetic operations are +, -, x, /, and modulo; they are provided for decimal numbers, byte strings and time units. The four logical operations are .A., .0., .E., and .C. which represent logical and, or, exclusive-or, and complement, respectively. They are provided for byte strings. The six shift operations are SHL, SHR, ROL, ROR, RCL and RCR which represent logical shift left and right operations, rotate left and right operations, rotate-with-carry left and right operations, respectively; they are available only for the data type of byte string.

Concatenation, finding-length, finding-substring and conversion operations are provided for both character strings and byte strings. In addition, delay and conversion operations are provided for data type of time units, and the set operation is provided for data type of status.

Data structures and operations

There are many data structures commonly used in programming. Examples are arrays, queues, stacks, tables, trees, files and list structures. In the HLM language, data structures are declared in data declarations. Examples of data declarations are shown in Figure 3. The available data structure types are buffers, one-dimensional arrays, stacks, ports, clock, and files.

A buffer is a data storage which may store any one of the five data types (it is equivalent to the simple variable in a programming language where the variable stores a mathematical object). An array is a linear list of a fixed number of array elements which have the same data type. Each array element can be referenced by a subscript. Only one-dimensional arrays are adopted to keep the hardware array structure simple. A stack is a linear list of a varying number of stack elements of the same data type. It is a first-in-last-out structure. Only one end of the stack is accessible for insertion and deletion of stack elements.

A port is a "data storage" through which input data or output data flow. A clock is a special type which generates the real time, and a timer is a special type where time units are being accumulated. Both accept only the data type of time units. A file is a list of related file elements commonly called records. (The records physically reside on an external storage device.) Each record may be structured or unstructured. A structured record consists of fields and subfields and it can be of fixed or variable length. An unstructured record can be interpreted by means of a structured template.

The data operations for the data structures are shown in Table II. In brief, there are two operations for buffer, four for array, seven for stack, two for port, eight for file and one for clock.

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Data Operations</th>
<th>Data Flow Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>decimal number</td>
<td>5 decimal arithmetic operations</td>
<td></td>
</tr>
<tr>
<td>character string</td>
<td>Concatenate strings x and y</td>
<td>a := b + c/d;</td>
</tr>
<tr>
<td></td>
<td>Find length of string x</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Find sub-char string in string x,</td>
<td></td>
</tr>
<tr>
<td></td>
<td>starting at 4th char for j chars</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Convert char string to byte string</td>
<td></td>
</tr>
<tr>
<td>byte string</td>
<td>4 logical operations</td>
<td>i := j; k;</td>
</tr>
<tr>
<td></td>
<td>5 byte arithmetic operations</td>
<td>k := j + k;</td>
</tr>
<tr>
<td></td>
<td>4 shift operations</td>
<td>i := SHR(j, a);</td>
</tr>
<tr>
<td></td>
<td>Find subbyte string in string f,</td>
<td>f := SUBBYTE(f, i, j);</td>
</tr>
<tr>
<td></td>
<td>starting at 4th byte for j bytes</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Convert byte string to char string</td>
<td>x := CONVCHAR(a);</td>
</tr>
<tr>
<td>time units</td>
<td>Convert x msec into date-and-time</td>
<td>dat time := CONVTIME(x);</td>
</tr>
<tr>
<td></td>
<td>Delay n time units</td>
<td>a := n DELAY n;</td>
</tr>
<tr>
<td></td>
<td>5 integer arithmetic operations</td>
<td>t1 := t2 + 3;</td>
</tr>
<tr>
<td></td>
<td>increment/decrement buffer x by 5</td>
<td>X := INC 5;</td>
</tr>
<tr>
<td>status</td>
<td>set buffer x to status on</td>
<td>X := DEC 5;</td>
</tr>
</tbody>
</table>

Note: This is a control operation which is to be specified by a control flow statement as shown in Table IV.
Any data structure may be subdivided into any number of fields or sub-fields which are organized by level numbers. Examples of data structures with fields are shown in Figure 3.

Following conventional usage, the names of data structures in a data declaration may sometimes be referred to as variables.

Control data types

A control data type is the data type which affects the flow of control in program execution. For example, the value of a test in an if-statement is a control data type since this value causes a change of control flow. In a conventional programming language, the control data types are not specially recognized; they are not considered as separate data types.

In the HLM language, there are two control data types—status and time units; they have been shown in Table I. A status is a datum stored in a data storage specifically for control flow modification. The datum can be numeric, boolean (i.e. TRUE or FALSE), or symbolic (such as “on” or “complete”). The time unit represents such units as hours, minutes, seconds and microseconds of time. For convenience, it may be abbreviated as tut.

Data flow statements

A data flow statement specifies one or more data operations. It consists of a sequence of operands and operators. The syntax of a data flow statement requires it to begin with a data-operator keyword such as PUSH and SIZE. Examples of data flow statements are shown in Tables I and II, where each data flow statement begins with a data storing operation indicated by the data operator “:=”. The data storing and data reference operations together with the input and output data flows, the timing reference and the operator precedence are subsequently described.

Data reference

Data reference is the appearance of a declared name in a data flow statement. It is a data operation since the appearance means the access of the value of the name. It is the most common data operation. If an operator were chosen to represent this operation, this operation could be “fetch a” or “access a” where a is a declared name. No operator is used to represent this data operation in programming languages. This practice is followed in the HLM language.

Data storing

Another often-used data operation is data storing which stores an operand into a data storage. The operand is the value of a declared name or the value of an expression; the data storage is a declared data structure. If an operator is to be chosen to represent the data storing operation, this data operation could be: “store a into b” or “store a + b into c,” or “assign a into c.” Common practice in programming language uses an assignment operator such as “:=” to represent it. In the HLM language, this practice is followed. However, the assignment statement or statements are enclosed by the reserved control flow delimiters, BLOCK and ENDB to enhance the visibility of the data flow statements and allow a simpler implementation.
TABLE II—Data Structure Operations and Data Flow Statements

<table>
<thead>
<tr>
<th>Data Structure Type</th>
<th>Data Operations</th>
<th>Data Flow Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td>buffer</td>
<td>reference buffer x</td>
<td>y := x;</td>
</tr>
<tr>
<td></td>
<td>store into buffer x</td>
<td>x := y;</td>
</tr>
<tr>
<td>array (one-dim.)</td>
<td>reference array element x[n]</td>
<td>y := x[n];</td>
</tr>
<tr>
<td></td>
<td>store into array element x[n]</td>
<td>x[n] := y;</td>
</tr>
<tr>
<td></td>
<td>find number of elements of array x</td>
<td>y := SIZE(x);</td>
</tr>
<tr>
<td></td>
<td>search array x for argument y</td>
<td>y := SEARCH(x) FOR (y);</td>
</tr>
<tr>
<td>stack</td>
<td>reference nth element from top of stack x</td>
<td>y := x[n];</td>
</tr>
<tr>
<td></td>
<td>push down y into stack x</td>
<td>x := PUSH(y);</td>
</tr>
<tr>
<td></td>
<td>pop up from stack x into buffer y</td>
<td>y := POP(x);</td>
</tr>
<tr>
<td></td>
<td>test stack x for empty</td>
<td>x := EMPTY;</td>
</tr>
<tr>
<td></td>
<td>find number of elements of stack x</td>
<td>y := SIZE(x);</td>
</tr>
<tr>
<td></td>
<td>initialize stack x to empty</td>
<td>x := EMPTY;</td>
</tr>
<tr>
<td></td>
<td>search stack x for argument y</td>
<td>z := SEARCH(x) FOR (y);</td>
</tr>
<tr>
<td></td>
<td>and return index from stack top for the first occurrence of y. (return -1 if none found)</td>
<td></td>
</tr>
<tr>
<td>port</td>
<td>input from port 1 into buffer cmd cmd: =INPUT(1);</td>
<td>OUTPUT(0) := '@';</td>
</tr>
<tr>
<td></td>
<td>output '@' to port 0</td>
<td></td>
</tr>
<tr>
<td>file</td>
<td>open input file x</td>
<td>OPENIN(x);</td>
</tr>
<tr>
<td></td>
<td>open output file x</td>
<td>OPENOUT(x);</td>
</tr>
<tr>
<td></td>
<td>close and save file x</td>
<td>CLOSESAVE(x);</td>
</tr>
<tr>
<td></td>
<td>close and delete file x</td>
<td>CLOSEDELETE(x);</td>
</tr>
<tr>
<td></td>
<td>read DIRECT file x record n</td>
<td>READDIR(x[n]);</td>
</tr>
<tr>
<td></td>
<td>write DIRECT file x record n</td>
<td>WRITEDIR(x[n]);</td>
</tr>
<tr>
<td></td>
<td>write SEQUENTIAL file x</td>
<td>WRITESEQ(x);</td>
</tr>
<tr>
<td>clock</td>
<td>read clock x in declared time units</td>
<td>READ(x);</td>
</tr>
</tbody>
</table>

Input and output

In the HLM language, the input data or the output data flow through a pseudo-data storage called "port." An I/O device is connected to a particular port; this connection is determined by the system configuration. There are a maximum of 128 input ports and 128 output ports; the port is represented by the reserved words INPUT or OUTPUT, and the port number is indicated by a numerical subscript. Examples are shown in Table II.

Clock

A clock is a data source which generates the data type of time. It generates real time for time referencing and for synchronizing hardware processors. Reserved word CLOCK is used to represent the real time clock. A data declaration is needed to declare the names and time_units desired for one or more clocks, though all the clocks give the same real time. A reading operation to CLOCK gives a reading in its declared time_units.

Expressions

An expression is concatenation of operands and operators to specify a sequence of operations. A numerical expression is a sequence of numerical operands and arithmetic operators. In the HLM language, there are five types of expressions corresponding to the five data types. The order of evaluating the operations in an expression follows the precedence of the operators. In the HLM language, the precedence follows common convention.

CONTROL

Control refers to those language constructs that may change the path of program execution. There are several aspects of control that need to be considered—control types, control operations, control flow and control structures. These aspects for the HLM language are discussed next.

Control types

The control type refers to the manner in which the control flows during program execution. In a conventional programming language, the control flow follows the written order of the statements of the program; this type of control is sequential. There are other types of control. For example, PL/1 has the concurrent type of control in the name of multiple-tasking. PL/M has the interrupt type of control to meet the needs of a microprocessor system. CDL permits non-sequential execution of statements.

In the HLM language, two control types are selected—sequential and interrupt. The sequential type of control permits the execution of control flow statements according to the order of the control flow statements in a procedure. The execution begins with the main procedure which calls directly or indirectly the other procedures.

The interrupt type of control permits the suspension of the control flow and transfers the control flow to a predefined interrupt procedure. This interrupt procedure is then executed and returns to where the control flow was interrupted. The interrupt type of control also permits interruption by software.

Boolean expression

The application of a relational operator (i.e. =, ≠, >, <, ≥, ≤) to two data storages of the same type yields an expression of the control type boolean. All six relational operations are available to four data types, and only two (=, ≠) are available to the remaining data type STATUS. (See Table III.)
The value of a boolean expression can be negated by prefixing it with the unary boolean operator NOT. Two boolean expressions can be joined by one of the three binary boolean operators (AND, OR, or XOR) to form a more complex boolean expression.

Note that all boolean expressions must contain at least one relational operator with one exception. If a data storage of type STATUS's value is either the status constant TRUE or FALSE then it is considered to be a boolean expression.

**Control flow statements**

Control flow refers to the order in which the control flow statements are executed. Since the data flow statements are embedded in the control flow statements, control flow also refers to the order in which the data flow statements are executed.

In the HLM language, there are two control types—sequential and interrupt. Skeleton control flow statements are shown in Table IV. The control flow statements for the sequential control type are:

a. Block statement
b. If statement
c. While statement
d. Call statement
e. Return statement
f. Macro statement
g. Procedure statement
h. Set statement

Each of these control flow statements is single-in-single-out. These simple control constructs make the HLM language a structured programming language.

The control flow statements for the interrupt control type are:

a. Interrupt statement
b. Disable statement
c. Enable statement

The interrupt statement suspends the current control flow, and transfers it to a specially defined procedure. The enable and disable statements enable and disable the ability of the interrupt to occur, respectively.

**Control structure**

Control structure refers to the manner in which the declarations and statements are organized to steer the control flow. For modularity, declarations and statements are allowed to be grouped together in a certain manner. For example, the compound statement in Algol-60 and the do-end group in PL/I allow the grouping of statements. In Algol-60 and PL/I, both statements and declarations may be grouped into a so-called block structure; the block structure may permit a complex interaction between the data flow and control flow of a program.

In the HLM language, simple control structure is sought-after. There is no ALGOL-like block structure. Only three control structures are permitted—nesting structure, macro substitution and procedure structure. The nesting structure allows the nesting of two or more control flow statements (e.g. if statement enclosing a while statement). The macro and procedure structure are described in the next sections.

**Macro**

As mentioned before, macro definitions in a HLM language program describe the macros of the program. Each macro definition specifies some program text for substitution. The program text is substituted at where it is called. Syntax of this program text is not recognized until after the macro is called and the textual substitution is made.

In the HLM language, macro definitions within a macro definition are not allowed for simplicity in implementation. Nevertheless, they serve the useful functions of text substitution and text compacting.
Procedure

As mentioned before, procedure definitions in a HLM language program describe the procedures of the program, and the first procedure is the main procedure.

Procedure structure

There are three relations among the defined procedures, which are the procedure defining, calling and returning relations. These relations are called collectively the procedure structure.\(^5,6\)

In the HLM language, simplicity of the procedure structure is sought-after. As a result, a “simple” procedure structure is chosen which has the following characteristics:

a. No procedure may be defined within another procedure (i.e. no nested procedure definitions).
b. No procedure may be defined or called recursively.
c. Each procedure always returns to where it is called.

The simplicity of the previous procedure structure contributes toward simpler program writing and in turn more reliable programs.

Procedure parameters

A procedure may need no parameters if all variables are declared globally. However, in the HLM language, there can be local data declarations and parameters are permitted in a procedure definition. The parameter can be the name of any data structure of any data type. However, it can be neither a procedure name, nor a macro name. All parameters are called by reference or by value.

It should be noted that procedure parameters contribute to the interaction between the data flow and the control flow of the program. This interaction is needed, but it should be minimized and made visible for the purpose of contributing toward reliable programs.

Function

It is common to have functions in a programming language. For example, there are function subprograms in Fortran and typed procedures in Algol-60. In the HLM language, no specific designation is provided for functions. For most commonly-used functions, they may be pre-declared as data operators. Otherwise, a function is regarded as a special case of procedure where the procedure returns with a single value for any data type and a call statement is still required.

LEXICALITY

Lexicality deals with the symbols and codes of the language. It plays an important role since it greatly influences the appeal of the language to the potential users as well as the implementability of the language on a computer system. In this section, we choose the character set and code. We describe how constants are written. We specify how names are spelled, and what the operators and reserved words are.

Character set and code

The seven-bit ASCII X3.4-1968 code consists of the following two types of symbols:

a. Single character symbols which consist of 52 letters, 10 digits and 33 special characters.
b. Control symbols which consist of 33 reserved words.

An escape function represented by symbol ESC is provided in order to allow the ASCII code to be expanded beyond the 128 characters.

In order to guarantee that the character set is capable for information interchange between computer systems and communication systems, the 95 single character symbols of the seven-bit ASCII code are chosen as the character set of the HLM language. From this character set, the constants, the identifiers and the operators are formed.

Constants

Constants are data values. The constants for the data types of the language are described below.

Numerical constants

A number is a numerical constant. For the data type of decimal numbers, a numerical constant is written as a decimal number, which can be positive or negative. A decimal point may exist if it is needed. Examples of numerical constants are:

\[-579, 579.0 \text{ and } +57.95\]

The permissible range of decimal numbers is to be chosen during implementation; the default range is chosen to be 15 digits including a sign.

Character-string constants

A character-string constant is a string of characters enclosed by a pair of apostrophes. In case an apostrophe is a character of the string, this apostrophe is written as a double-apostrophe. Examples of character-string constants are:

`'ABC5*+=/?'` and `'ABC''DEF''GHI'`.

The size of a string is limited by the available memory space.
Byte-string constants

A byte-string constant consists of one or more bytes in concatenation (i.e., 8, 16, 24, etc. bits). It can be written in hexadecimal, octal, or binary form. The constant must be preceded by letter H, Q, or B and followed by a hexadecimal, octal or binary number in a single-quote pair. Examples are:

H'A1B2C3', Q'234567', Q'255', and B'10101010'.

Status constants

A status constant is an identifier or a decimal number. It first appears when the status is defined in a data declaration. It represents a condition to be used as an operand in a boolean expression in either the SET, IF or WHILE control statements. Examples of status constants are: 321, on, off, TRUE, FALSE, and ready_queue_empty.

Time constants

A time constant is a numerical value in time units. The time units can be microseconds, seconds, minutes, hours, days, months, years or combinations thereof. Examples are 10 microseconds, 15-hour/20-minute/25-second, 1977/april/15. The time units of a clock are declared in a clock declaration. The generic term for a time-unit is coined to be a “tut.”

Identifiers

Identifiers are the names for data types, data structures, procedures, macros, statuses, subscripts, fields, etc. Examples have been shown in Figure 3. An identifier name is a character string consisting of letters, digits and character underscore; with the first character being a letter. Although there is no limit to the length of the identifier name, the identifiers in a program are distinguished by the first eight characters. This choice makes the size of the hardware’s symbol table reasonable.

Operators

Operators are the terminals of the language. They consist of single-character operators, two-character operators, and reserved words. There are 20 single-character operators, 7 double-character operators, and 84 reserved words. Note that a blank is also a delimiter which is required for separating symbols.

A FAMILY OF LANGUAGES

The aforementioned programming language can be organized into a family of member languages. In this way, the programmer needs to learn only a member language and the compiler/interpreter can be simpler. It is a family because all the member languages have the same program structure, control structure and data structure. Their differences are in the choice of the data types and control types and their associated data and control operations.

A member language for software implementation may choose all the five data types and both control types since it needs all of the descriptive power of the language. A member language for computation needs only the data types of number and status and the sequential control type. A member language for data-processing needs to choose the data types of number, character string, file and status and the sequential control type. A member language for real-time control needs the data types of number, byte-string, status, time and both control types. A member language for microprocessors needs the data types of byte-string, status and time and both control types; this choice of data and control types is made to match the microprocessor capability.

For each of the member languages of the family, a compiler or an interpreter or both may be constructed. The implementation can be made simpler for the whole family by providing a family of software modules so that the compiler or interpreter of a member language could be built from a subset of these software modules.

CONCLUDING REMARKS

A programming language serves as a bridge between the programmer and the computer hardware. In the past, a programming language may have been designed for its power and elegance. The language as a result may have become hard to understand, long to learn and costly to implement. The language may well have been over-designed. The design of the HLM language has been an engineering undertaking with both the programmer’s viewpoint and the language implementation taken into consideration.

To the programmers, the language is designed with a primary consideration of simplicity and understandability. The following language concepts and constructs have been adopted.

1. The concept of the data flow and control flow in a program and visibility of their interaction.
2. Inclusion of the data types of status and time and their data operations.
3. Data structures can all be declared with fields and subfields.
4. Data operations are provided for all data types and all data structures.
5. Treatment of inputs and outputs as data sources and sinks, respectively. Input and output data transfers are treated as input and output data operations, respectively.
6. Provision of a real-time clock is provided together with data operations.
7. Simplicity of control structure. There are no block structures, no recursive procedures, no nested procedure definitions and no nested macro definitions.
9. Provision of both sequential and interrupt control types.

For implementation considerations, the language is designed to give a simple interpretation model. Costly and unnecessary constructs are avoided or eliminated. Implementation by both compilation and interpretation is considered. The following are the specific considerations:

1. Separation of the data flow and control flow interpretation.
2. Simplified model as a result of no block structures, no recursive procedures, no nested procedure definitions and no nested macro definitions.
3. Presence of reserved words at the beginning of each declaration or statement.
4. Declaration of the number of digits, characters and bytes for the data types of number, character string and byte string, respectively.

The following language constructs should be excluded for the sake of simpler implementation, but they have not because of more effectiveness in programming:

1. Provision of field and subfield definition for any of the data structures.
2. Provision of data operations for all of the data types and data structures.
3. Provision of procedure parameters.

The HLM language is now under an experimental and evaluation phase. During this phase, the language is used for writing programs. An interactive interpreter for a member language is being designed and implemented for experimental and evaluation use. The HLM language will be revised and refined as a result of this phase.

ACKNOWLEDGMENT

This research is supported by Grant MCS-75-05505-A01 from the National Science Foundation to the Department of Computer Science of University of Maryland.

REFERENCES
