# DEC/3000

# AN EXAMPLE OF SPECIAL-PURPOSE LANGUAGE DESIGN AND IMPLEMENTATION

Matthew J. Balander
The B & B Computer Company

ABSTRACT: Special-purpose programming languages are often an excellent method of reducing costs and improving productivity. This paper discusses some of the factors influencing the decision to develop and employ a special-purpose language, and presents an example of such a language. The language presented, DEC/3000, provides powerful data-entry capabilities for programmers producing applications in host languages. In tests, the use of this special-purpose language has indeed reduced costs and improved programmer productivity.

## 1. Why another language?

Despite the (quite proper) effort to develop good general purpose programming languages in the discipline of computer science, there are nonetheless still problems for which special-purpose languages should be emphasized regardless of considerations of universal applicability or portability. Factors which motivate the effort to design and implement such languages, to adopt them for use in applications, and to educate programmers in their use include cost effectiveness, programmer productivity, program reliability, and maintainability of the final product.

Languages should not of course be implemented to meet every whim of each programmer. Problem areas for which languages should be provided must be very carefully defined, if the language to be provided is to fulfill its goals of lowered software production costs and improved programmer productivity. An appropriate problem area may be identified as follows:

- o It must be possible to clearly define the problem area. It is, for example, an inadequate isolation of the problem area to state, "A language is needed to help with data entry tasks." The problem area must be carefully, clearly, and fully specified. The construction of a "language" is under consideration. It is critical to understand fully exactly what the proposed language is supposed to be able to "talk" about.

- o The programming of solutions in the problem area, using available languages, is tedious, complex, and error-prone. Existing languages, in other words, are not well suited to the expression of solutions to tasks in the problem area. The degree of difficulty with which solutions are provided in existing languages is a measure of the cost of solving problems in those languages, and, in general, an indication of the future costs of reliability and maintainability of the software. (The cost of the proposed language may thus be measured relative to the cost of existing languages by noting the reduction in complexity of solutions provided in the special-purpose language.)

- o An area is a candidate for a special-purpose language if programs are frequently produced to meet needs in the area, or such programs are heavily used. The effort involved in designing and implementing a special-purpose language cannot be justified if the language produced will rarely be used.

The decision to produce a special-purpose language is thus largely governed by economic factors. If a special language will reduce costs sufficiently, it will provide a cost-effective means of providing solutions to data processing problems. Languages achieve cost reductions in the software development cycle in several ways. Some of the more straight-forward are as follows:

- o By increasing the power of the language being used by programmers (defining "power" loosely as the amount of work done for the programmer by a line of code) the size of programs, measured in lines of code, is reduced. Given a fixed cost per line of finished code regardless of language, there is a direct savings in program development cost in using a powerful special-purpose language.

- o A well-designed language is easy to use. Its idioms and methods of expression are "natural" for expressing solutions to the category of problems for which it was designed. This reduces the complexity of solutions to problems. Complexity of a program varies inversely with reliability of the program. The less complex a program is, the more reliable it is. Greater reliability means lower costs for software development. As a result of reduced complexity, testing is simplified, and debugging is both easier and less time consuming.

- o The increased power and reduced complexity of a special-purpose language serve to reduce software maintenance costs. Adjustments and modifications of programs will require less programmer time, affect fewer lines of code, and may with greater confidence be undertaken by programmers who did not originally write the software being modified. Program modification in addition enjoys the same benefits from increased power and reduced complexity that are experienced during original software development.

From all these remarks it is plain that the use of special-purpose programming languages can be very beneficial. It is also plain that the benefits received depend heavily on the language design and implementation.

## 2. How is a special-purpose language designed?

Once a problem area has been defined for which a special-purpose language is being considered, the language must be designed. The design of a language is an art, not a technology. It will always remain so, since language is an intensely human activity, and is fundamentally alien to mechanical processing. It is far beyond the scope of this paper to discuss language design in detail, but some central considerations should be mentioned.

The language designer must know the problem area. This involves not just a passing acquaintance, but an intimate familiarity. Such knowledge usually is a result only of much work in the field. The language designer must be familiar with the problems of the problem area, and with the sorts of solutions with which they are best met. He must be aware of the limitations of available languages when applied to problems in the field, and have some experience with trying to "make do" with these languages.

The language designed must be powerful. If it is not sufficiently powerful, many of the benefits of implementing the language will be lost. Language constructs and idioms must be provided which allow the programmer to express as concisely and unambiguously as possible his intentions. Brief constructs are preferred to wordy ones, and one line of code to two or more. On the other hand, the APL-ish, mystical, "does-it-all" one-liner type of language is most certainly not desirable. Readability should always be maintained.

When providing power, it is necessary to be cautious that flexibility is not unduly limited. Regardless of the power of the language, it will not be used by programmers if they are unable to express solutions to all relevant problems with it. In zeal for making the programmer's life easier, by "doing all the work for him," care must be taken not to make it difficult or impossible for him to solve atypical or unanticipated problems using the language.

The language designed must be "natural." That is, it must express programs in much the same way that the programmer thinks, or should think, about them. A judicious compromise between machine efficiency and programmer comfort must be found. It is here that the art of the language designer will be most needed.

3. How is the language to be implemented?

While language design is the most critical phase of the production of a language, implementation is by no means less important. A good, reliable, easy-to-use implementation of the language must be provided to programmers. It must take into account their needs and work. An implementation which is difficult to use, or which is unpredictable or unreliable in its operation, will not be used. The documentation, not only of the language, but also of the implementation of the language, must be comprehensive, clear and concise. It is an integral part of the implementation as a whole.

There are three very viable options for implementation of a language: a pre-processor, an interpreter, and a compiler. The simplest of these is to implement a pre-processor for some existing language. The special-purpose language then takes the form of statements embedded in a program text ultimately intended for some existing processor. This text is first fed through the pre-processor, which converts the special-purpose language statements into source statements of the language in which the special-purpose statements are embedded, and outputs a program text which may be submitted to the existing processor. Such a concept is familiar and has been used successfully in such systems as RATFOR, a pre-processor for FORTRAN programs providing while, do-until, and similar constructs.

If all existing languages are so unsuited to the problem area that it is not wise to pre-process special-purpose language statements into one of them, then the implementation must provide all the functions of a source-language processor. One way in which this is often accomplished is to implement an interpreter for the language. Interpreters have many familiar advantages, and are not as difficult to implement as most other options. In addition, the technology of interpreters is well documented in the literature, and good advice may be found there to assist in writing the interpreter itself.

Although interpreters have many well known advantages, they also have many well know disadvantages. They are slow, often bulky, and usually have no convenient mechanism for accessing the full capabilities of the operating system and allied subsystems in which they are used. It the disadvantages of an interpreter weigh heavily in a particular situation against its use, the next alternative is to implement a full compiler. Implementing a compiler is undoubtedly the most difficult and time-consuming of the options, but possesses most of the advantages of the other options, as well as some unique to the use of a compiler. A well written compiler provides access to all system capabilities (assuming the language design has been done well enough to allow the language itself to express access to these capabilities). It provides much more efficiency than an interpreter, and is more congenial to a variety of environments, both batch and interactive.

The type of implementation to be used depends on the nature of the problem area itself, on the time and resources available for implementation, and the ways in which it is desired to use the finished processor. Pre-processors are the easiest to write, port, and maintain, compilers the most difficult, and interpreters are somewhere in between. In some cases hybrid implementations are most appropriate. Here the craftsmanship of the programmer implementing the language is of great importance. If the available programmers are not skilled in compiler writing, or have no experience in compiler writing, it may be unwise to ask them to implement a production compiler. On the other hand, the concepts and engineering of pre-processors are not of great difficulty, and most programmers with some experience in text processing can implement pre-processors in a reasonable amount of time. Interpreters, as compilers, require some special skills, and should not be undertaken by a staff without some prior experience in the area.

## 4. Can you give me an example?

The author has written a number of data entry programs, each of which was a portion of a larger data processing system. In each case, interactive terminals with forms and block mode capabilities were used. These programs were implemented in FORTRAN, COBOL, and SPL, as appropriate to the data processing system for which the data entry program was being written. The specifications for the programs indicated that the forms/block mode capabilities of the terminals were to be used whenever possible to assist terminal operators in entering data correctly.

In preparing to write these programs, it was necessary to learn a great deal about the terminals to be used (HP264X terminals were used for these particular projects). Not only was it incumbent on the programmer to be fully conversant with the application, but also to be an expert on the terminals themselves. Programming terminals such as HP264X terminals is in fact a form of low-level machine programming, and as such is quite prone to errors. No special software tools were available to assist in managing the terminals; all functions had to be explicitly provided by the applications programs.

In one case, eighty lines of SPL code were needed for the "DEFINEs" used to code the declaration of a single form in a fashion which was at least partially readable. In one FORTRAN program approximately two hundred lines of code were devoted exclusively to terminal management. In general, it was found that approximately one third of each of these programs were devoted to terminal management, with approximately one hundred lines per program devoted to defining and managing forms for display on the terminals.

Not only were terminal and form management lengthy, they were also difficult to code. The escape sequences used with HP264X terminals are meaningless in and of themselves. They are simply details which must be coded precisely in order for the terminal to behave as desired. Programmers who are not the original author of these programs will find it difficult to modify the forms involved when data needs change; the original author himself will find changes difficult.

A search was of course made for alternative methods of coping with the problem of managing these types of terminals during data entry. The only software available at the time was Hewlett Packard's DEL/3000. It was evaluated, and rejected for several reasons. First, it did not relieve the programmer of the necessity of being an expert on the terminals. To declare a form in DEL/3000 the programmer must actually produce the form from the keyboard of an HP264X terminal. He must still be familiar with all appropriate escape sequences. Second, DEL/3000 had to be used from an HP264X terminal. Not all terminals used for program development at the installation where this work was being accomplished were HP terminals. Also, because of this restriction, work on the forms and terminal management had to be done on-line instead of in (less expensive) batch. Third, DEL/3000 held out no hope of ever supporting any but Hewlett Packard terminals. Other forms/block mode capability terminals are available, and it is not wise to write one's software so that one is locked into a single vendor. Other objections, such as the "un-aesthetical" nature of DEL/3000, were raised, but could not be related to clearly definable, measurable problems.

The rejection of DEL/3000 was the rejection of the only available software tool for forms/block mode data entry with sufficient power to be considered. The remainder of this paper discusses the alternatives which we considered, and describes the solution we reached. We turned to consider what sort of tools we should implement ourselves. (In the meanwhile, the writing of data entry programs was undertaken using the various available languages, since it was not feasible to delay their production till more powerful tools were available.) What ever we came up with had to meet the following goals:

- o The tool produced must perform virtually all terminal and form management during data entry.

- o The tool must relieve the applications programmer of the necessity of understanding the programming of the terminal being used. The programmer should never need to know what escape sequences are used to perform terminal functions.

- o It must not be necessary to have one of the data entry terminals available for the programmer's use during coding of the application.

- o The tool must provide for adequate documentation of forms. It is not considered sufficient for forms to be simply entered at the keyboard; there must be a permanent, off-line record of each form clearly showing all protected/unprotected fields, display enhancements, and so on.

- o The tool must allow for future use of non-HP terminals with a minimal conversion effort. It is acceptable to need to rework the tool itself, but minimal changes should be necessary in applications programs.

- o The tool must dovetail with existing applications, and existing application languages, so that a conversion process may be undertaken, and so that programmers and analysts are not restricted in their choice of language.

Clearly a new programming language was called for. A carefully designed language, together with a comprehensive run-time support library, is capable of meeting all these goals. Implementation method was decided immediately. A pre-processor is not feasible because of the last goal. The new language must dovetail with a number of existing languages (at the least with COBOL, SPL, and FORTRAN), but pre-processors are oriented to a single application language. The difficulty of implementing several pre-processors, and the maintenance nightmares arising from having three or more programs that do the "same" thing, caused the firm rejection of a pre-processor. An interpreter was rejected for the same reasons that the pre-processor was rejected. (It is possible to construct an interpreter which is invoked by a program when needed, but such constructs are difficult on the HP3000 under MPE. Also, the use of such a system is burdensome on the programmer, and is not conceptually straight-forward.) we settled on implementing a compiler.

Since it is desirable to use the data entry tool with a variety of applications coded in a variety of languages, a "host language" concept was adopted. That is, program units coded in the new, special-purpose language would be used together with program units coded in some "host language." It was agreed that only the following restrictions should be placed on the choice of host language: the host must be able to call external program units, and to pass simple integer and integer array parameters by reference to these external program units. In this way, the special-purpose language program units would be accessable from at least COBOL, BASIC, SPL, and FORTRAN. Since no other potential hosts were in use at the time, this was considered to be sufficient.

The name "DEC" was adopted for the compiler and associated language. It is an acronym for "Data Entry Compiler." Since all DEC programs were to be used together with a host program, DEC could be devoted to solely the data entry/terminal management tasks which motivated it in the first place. Access to data bases and other files, complex data checking, and complex program logic could be left to the host. The following scope for the DEC language was defined:

> DEC is a special-purpose programming language in which data entry forms and activities utilizing forms/block mode terminals may be expressed. This is to include the definition and documentation of forms, the specification of elementary data editing and checking, the specification of type conversions, and the specification of the correspondence of data record fields with form fields.

This definition guided language design. The language designed, which is described in separate documentation, is primarily declarative in nature; it includes no explicit verbs. Actions are implied by the declarations (for example, type conversion actions are implied by the declaration of data types), but no actions are explicitly coded by the programmer. A sample DEC program is included at the end of this paper. It is heavily commented to explain the features of the language.

Terminal management tasks are shared between the compiler and the run-time support library, "DECLIB". All terminal manipulation may be performed via library procedures, and

DEC-generated procedures. There is a single entry point to DECLIB for all functions, regardless of terminal type. The DECLIB procedures are internally structured in such a way that additional types of terminals may be easily accomodated, while requiring normally only a single line in each application program to be changed to take advantage of these additional types.

Since DEC is a compiler, it inputs a source language file, and outputs RBM's in a USL file, and thus may be used during coding and program entry from any sort of terminal. It may be used either interactively or in batch.

The DEC language has been designed so that it documents forms very well, as may be seen by examining the sample at the end of this paper. All features of the display are evident from even a rapid examination of a source program. In addition, the programmer has great flexibility in placing comments in a DEC program. This encourages good documentation.

## 5. what have the results been?

The six goals for the DEC language have substantially been met. A six hundred line FORTRAN program has been reduced to two hundred lines of FORTRAN and about one hundred lines of DEC. Again defining "power" loosely, since one line of DEC replaced in this application four of FORTRAN, DEC may be said to be roughly four times more powerful than FORTRAN. This implies that DEC programs cost about one fourth as much to write as equivalent FORTRAN programs, and this has indeed been our experience. All programs converted to use DEC, regardless of language, were substantially reduced in size.

As an additional benefit, unintended but very welcome, all converted data entry programs now work on all our HP264X terminals. This is despite the fact that individual programs were originally implemented for a particular model, such as the HP2645A, or the HP2640B. In addition, there is now complete freedom to use page or line mode. The programs were formerly hard-coded to use one mode or the other.

Programmers have learned DEC in a very short time. A single afternoon is sufficient to read the manual and begin applying DEC to actual problems. Since DEC is straignt-forward, concise, and "natural," programmers find it easy to use.

Changes to a form, and to its associated data entry program, can now be made in a single afternoon, and in many cases in much less that an afternoon. Formerly one or two days or more could be consumed in such changes, depending on the extent of the change. Using DEC, a form can be entirely reorganized, and be back in service the same day.

6. Can you summarize all this?

In general, there are circumstances in which special-purpose programming languages can be used to good advantage. The design, implementation, and adoption of such a language can often be justified on a cost basis. While language design is a difficult art, analysts and many programmers with appropriate experience should be able to construct a language which would provide the desired cost benefits. Several options are available for implementing the language. The cost effectiveness of these options depends in part on the experience of the programming staff, and in part on the expected cost savings from use of the language.

An example has been presented for which available software products were unable to fill a well-defined need. Goals were established which upon examination indicated that a special-purpose programming language would be a viable method of obtaining a satisfactory software product to fill the identified need. An appropriate language was designed and implemented using a compiler and host language arrangement. The result was that the goals which had been established were substantially met, and in fact the cost of producing software using the new language is significantly lower than the cost of similar software produced without this tool.

```
* <--- LINES BEGINNING WITH "*" ARE COMMENTS
<< COMMENTS MAY ALSO BE ENCLOSED IN ANGULAR BRACKETS >>

$CONTROL USLINIT,LIST,SOURCE,MAP,CODE
* Note that compiler control is similar to HP compilers.  This
* specifies that a listing is to be produced, that it should
* include the source images, a symbol map, and code.  Code output
* is in an assembly-like format for readability.

*******************************************************************
* A DEC program consists of a declaration of a form, followed    *
* by one or more "data entry," or "DE," sections which declare   *
* data entry information associated with the most recently       *
* declared form.                                                  *
*******************************************************************

-FORM  FORMPROC  << name of procedure will be FORMPROC >>
  << You could call this procedure in COBOL with 'CALL FORMPROC >>
  << USING . . . or in FORTRAN with CALL FORMPROC(. . .)       >>

  0,0,"THIS IS A SAMPLE FORM"
  * This statment says, beginning in row 0, column 0, place the
  * following data: "THIS IS A SAMPLE FORM"

  2,0,"'  '"
  * This statement says, beginning in row 2, column 0, place a
  * two character unprotected field.  The initial contents of
  * this field are to be "  ".

  3,0,"'XX'"
  * Same as last statement, except row 3, and initial contents "XX".

  4,0,"!B'XX'"
  * Same as last, except row 4, and the "XX" is to be in inverse
  * video (display enhancement B).

  5,4,"\C!DTHIS WILL BE IN ALTERNATE CHAR SET C, ENHANCEMENT D"
  * At row 5, column 4, place the specified data using alternate
  * character set C, display enhancement D.

  LABEL:  12,12,"DATE:  !B'  '/'  '/'  '"
  * The terminal operator will see "DATE:    /  /  ", the last eight
  * characters of which will be in inverse video (display enhancement
  * B).  Note that there are only six unprotected characters.  Even
  * though these six unprotected characters are in three separate
  * unprotected fields, the fact that the field has a label ("LABEL")
  * indicates to DEC that all six characters should be considered
  * a single data field.  The label also allows reference to this
  * field in a later data entry section.

-MROF  << end of declaration of this form >>
```

```
-DE IN=INPUTPROC, OK=OKPROC
  << Declare data entry information associated with FORMPROC >>
  << Can later in host program CALL INPUTPROC(. . .), etc.   >>

  << We can reference fields defined in the form section FORMPROC >>
  << only by referring to labels declared there.  "LABEL" is the  >>
  << only such label in this example.                             >>

  0,4,P,LABEL; VERIFY NUMERIC-NO-BLANKS; SAVE
  << Beginning in byte zero of a data record, there is a 4-byte   >>
  << long, packed decimal field.  Its data source is the form     >>
  << field labeled LABEL.  Verify that the data input by the user >>
  << on the form is  all numeric, with no blanks.  After data has >>
  << been successfully input, and you go back to erase all the    >>
  << unprotected fields in the form, do not erase this one, SAVE  >>
  << it.                                                          >>

  * In addition, data editing (including justifying, blank and zero
  * filling, and the like) can all be specified.  Constants may also
  * be specified as a data source, instead of a field declared in a
  * form section.  Totaling of fields and auto incrementing or
  * decrementing of field contents may all be specified.

-ED   << ends the data entry section >>

*******************************************************************
* The host language program would be similar to the following:    *
*                                                                 *
* OPENDETERM(CBUF,. . .);  << OPEN THE TERMINAL FILE (DECLIB) >>  *
* FORMPROC(CBUF);    << DISPLAY THE FORM >>                       *
* INPUTPROC(CBUF,. . .);  << INPUT DATA FROM THE FORM >>          *
* << USER MAY HERE DO ANY DATA MANIPULATION DESIRED.  IF NOT      *
*    SATISFIED WITH DATA ENTERED, RE-EXECUTE THE INPUTPROC        *
*    STATEMENT AFTER GIVING USER A DIAGNOSTIC.  IF ALL IS OK, THEN *
*    CONTINUE. >>                                                 *
* OKPROC(CBUF,. . .);  << ERASE SCREEN EXCEPT "SAVE" FIELDS >>    *
* << REPEAT THE INPUTPROC/OKPROC SEQUENCE UNTIL ALL DATA IS IN >> *
* CLOSEDETERM(CBUF,. . .); << WHEN ALL DONE CLOSE FILE (DECLIB) >> *
*******************************************************************

-END  << ENDS DEC PROGRAM >>
```