Some practical experiences with the Pascal language

by G. G. GUSTAFSON, T. A. JOHNSON and G. S. KEY

Computer Sciences Corporation
San Diego, California

INTRODUCTION

In 1976 the Naval Ocean Systems Center (NOSC) initiated the BIODAB project to determine if a high performance relational DBMS could be developed to support scientific applications.

The BIODAB prototype was implemented in FORTRAN V and small percentage of Assembly Language (less than 10 percent). The modifications needed to enhance the static prototype for general purpose use involved the addition of several major processors, including: Data Base Definition, Update (Add, Change, and Delete), and Unload. In addition, the Report Writer had to be extensively modified for greater efficiency and flexibility.

The results of that study encouraged NOSC to develop RELATABASE as a general-purpose enhancement of the BIODAB prototype. The design constraints for RELATABASE were threefold. First, retain the high performance data access technology developed during the BIODAB project. This constraint required a hardware architecture in which sequential searching of a highly compressed data base was practical. An access method employing double-buffered, asynchronous I/O with a hardware masked search instruction was developed for this purpose. Second, RELATABASE should not use any more main memory than necessary for each of its many functions. Finally, RELATABASE should be an interactive system capable of supporting ad hoc queries of a data base. All of these criteria were satisfied using the UNIVAC 1110 at NOSC.

We define the "production environment" to be that environment in which programs are developed as deliverable products under constraints of both time and money. It is our objective to distinguish organized programming of this type from systems, research, hobbyist, or other types of programming.

THE DEVELOPMENT EFFORT

Implementation language choice

The choice of the implementation language was carefully considered. Many software tools were available in FORTRAN from the BIODAB project, and all team members were competent FORTRAN programmers. A preprocessor for writing structured FORTRAN programs was also available. In spite of these tools, Pascal was seriously considered as the new applications language for the following reasons: (1) as a block-structured language, Pascal did not require a preprocessor; (2) most of the tools which had been built for FORTRAN, and particularly for character manipulation, were either unnecessary or easily rebuilt in Pascal; (3) because Pascal-1100 allows direct access to the UNIVAC-1100 Executive Request mechanism, reliance on special Assembly Language routines could be significantly reduced; as an example, Figure 1 lists the RELATABASE external module DRUM10 which interfaces directly with the executive and performs random access I/O; the replacement of Assembly Language routines with ones written in a high level language is usually desirable because the code is easier to read and maintain, and because the maintenance function can usually be performed by less senior programmers; (4) although only one of the team members had previous experience with Pascal, learning the language was not expected to be a significant problem; (5) because the new software would rarely need to communicate with the existing FORTRAN code, the interface between the two languages would not be a problem; (6) the control structures, data structures, and data type security features of Pascal were definite advantages.

In order to promote rapid learning of Pascal and uniform coding practices, a set of programming standards was chosen and imposed on all Pascal code. A similar set of standards was provided for all new FORTRAN code which was needed to support existing BIODAB code. The Pascal standards included: descriptive preamble comments for each independently compiled procedure or function; the capitalization of all Pascal reserved words; a few indentation rules to make data and control structures more visible.

Project experience with Pascal

The Pascal programming team had three members. Two full-time project members had experience in the production environment but did not know Pascal. The other programmer was a part-time member who had used Pascal extensively as a student.

The inexperienced Pascal programmers found that mastery of the language came slowly at first for a number of reasons. The most common problems were remembering to use the semicolon as a statement separator and the lack of
const
low = 3B;

Type
ascidentype = packed array [1 .. 12] of CHAR;
fieldtype = packed array [1 .. 12] of CHAR;
packertype = packed record
    filename : packed array [1 .. 12] of CHAR ;
    wordtwo : INTEGER ;
    status : 0 .. 778 ;
    finalcount : halfword ;
    bufferaddress : halfword ;
diskaddress : INTEGER ;
end ;
(* ASCII to FIELDATA string conversion *)
procedure asc2fd ( inputstring: ascidentype ) ; EXTERN ;

value outputstring: fieldtype ;

procedure ENTRY drumio ( filename: ascidentype ;
    iopacket: packertype ;
end ;

var
iopacket : packertype ;

begin
    if nwords <= 0 then begin
        nwords := 0 ;
    end else begin
        status := iopacket.status ;
        nwords := iopacket.finalcount
        iopacket.diskaddress := halfword ;
        iopacket.iofunction := iofunction ;
        iopacket.nwords := nwords ;
        diskaddress := INTEGER ;
end ;

Figure 1—Pascal-1100 executive request (ER).

a construct-closing keyword which identified which structure was being closed (i.e., something like the END IF in ANSI X3.9-1978 FORTRAN). The latter problem was solved with some satisfaction through indentation of nested constructs. However, once the learning pains were past, the power and flexibility of Pascal began to be appreciated. Code generally was written faster and easier than would have been possible in FORTRAN. Data type security proved to be a most helpful feature in reducing run time errors. The record and set (Figure 2) data structures were well received and saw frequent use. A record type is a most convenient way to group related data in a logical and clear manner. The ability to build and manipulate sets offers the programmer many new and powerful approaches to algorithm development.

Because the implementation of RELATBASE followed the top-down concept with functional independence of modules, we encountered no problems of module interfacing. Most of the debugging process was limited to minor logic errors in newly introduced modules. The modification of the standard procedure WRITELN in Pascal-1100 to allow numeric variables to be displayed in octal was a great help because several of the data types we used were defined as packed, partial computer words and could not be meaningfully displayed otherwise.

Project productivity

The productivity of the project can be divided between the efforts of the FORTRAN team and the Pascal team. In all, RELATBASE contains 53,600 lines of source code. Of that total, 9,100 lines were retained from the BIODAB program, 23,000 lines were new FORTRAN code (generated by the FORMEL ‘structured FORTRAN’ preprocessor), 3,200 lines were Assembly Language code to support Pascal programs, and 18,300 lines were written in Pascal. The implementation of RELATBASE began in mid-January with a five-man team, and continued through September of the same year. During that time the staff size varied, decreasing until only one member remained at the project’s conclusion. The total number of man-hours given to the entire project was 4,574.

The production rates for Pascal and FORMEL code have been adjusted to account for source lines added because of the coding conventions (Pascal) or preprocessor (FORMEL) used. The adjustment factors are 90 percent of total lines for Pascal and 75 percent of total lines for FORMEL. On this basis, Pascal production rates were 50 lines per programmer-day. FORMEL production is computed at 86 lines per programmer-day. The BIODAB project, where 90 percent of the code was written in FORTRAN, without a preprocessor, produced 45 lines of source code per programmer-day. These data are summarized in Table I.

In order to compare the Pascal rate with FORMEL’s, one must keep in mind that the FORMEL staff was composed of expert FORTRAN programmers, while two of the members of the Pascal staff had never coded in Pascal be-

From the collection of the Computer History Museum (www.computerhistory.org)
DIFFICULTIES WITH PASCAL

Pascal offers significant advantages in terms of data and control structures, but exhibits some disadvantages in the production environment. These disadvantages may be due in large part to the design goals which Wirth chose for the language. The current standardization effort for Pascal may eliminate most of these disadvantages. The production programmer deciding upon an implementation language should, however, be familiar with Pascal's disadvantages as well as its advantages.

We encountered two broad categories of problems using Pascal in a production environment. The first set of problems was associated with the language definition itself. The second set was associated with the specific language implementation that we used. In fairness, we hasten to add that a number of the difficulties may have arisen from the manner in which RELATA BASE was developed using multiple source languages (i.e., FORTRAN, Pascal and Assembly).

The difficulties with Pascal as a production language can be further subdivided into those whose repair we consider to be mandatory and those whose repair we consider to be desirable. "Mandatory" implies that without some solution to the specified problem, preferably a new standard for the language, Pascal cannot provide the production programmer with needed facilities. "Desirable" implies that a solution should be sought within the language standard, although a solution, albeit awkward, can be devised by the programmer. Desirable also includes those facilities which are primarily a convenience or could be included within a particular implementation.

Language definition

Standard types

Failure to provide standard types equivalent to the FORTRAN types DOUBLE PRECISION and COMPLEX causes considerable extra programming with Pascal. Had Pascal remained a teaching language, then the need for these data types would be reduced. It can be argued that Pascal supports the programming of double precision (or any precision) arithmetic operations. However, the need to develop specialized arithmetic routines is not in keeping with an environment in which a customer is paying for a product. The same argument applies to the complex arithmetic operations. It may be easy to define complex variables through appropriate type statements, but coding and recoding the complex arithmetic operations through functions and procedures is undesirable in a production environment.

Because Pascal supports strong type checking, it is necessary to include a type-less operator akin to the intrinsic function BOOL in UNIVAC's FORTRAN V:

\[
\begin{align*}
\text{(bool-stmt)} & := \text{BOOL}((\text{one-word-var}) \\
\text{(one-word-var)} & := (\text{integer-variable}) \\
& := (\text{real-variable}) \\
& := (\text{logical-variable}) \\
& := (\text{typeless-variable})
\end{align*}
\]

The result of the BOOL function is to wholly ignore any type incompatibilities which might arise during arithmetic or Boolean operations upon variables. The power of this function must be limited to logical expressions. Its use would be inappropriate in an assignment statement, for example.

External compilation

Two associated problems exist within a multi-language environment. Pascal does not define the mechanism needed for independent external compilation of modules. In large-scale software development, where more than one programmer is involved in development, it is mandatory that modules be developed independently and tied together through mechanisms offered by the operating system or system processors. Requiring five programmers to simultaneously edit the same workspace is justification enough for external compilation.

In complex systems it is frequently necessary to collect programs into separately addressable banks. This technique, known as "bank-named collection," minimizes core-second charges by switching out banks which are not currently required. In RELATA BASE, up to four active banks were resident in core, and occasionally the need arose to communicate between them. Pascal does not support this requirement, particularly if one of the two communicating banks is switched out.

FORTRAN solved the problem posed by switched bank communication through the COMMON statement. A named COMMON can be placed in a control bank (i.e., one that would not be switched out), and any FORTRAN module can make updates to it. This feature was particularly useful in maintaining the status of a bank that was executed earlier but is no longer active. Although RELATA BASE designers were able to work around this problem, we believe the need for a Pascal equivalent to FORTRAN's COMMON is mandatory.

---

TABLE I.—Summary of Language Productivity

<table>
<thead>
<tr>
<th>Source</th>
<th>Source Lines</th>
<th>Productivity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unweighted</td>
<td>Weighted</td>
</tr>
<tr>
<td></td>
<td>Unweighted</td>
<td>Weighted</td>
</tr>
<tr>
<td>PASCAL</td>
<td>2774</td>
<td>21500</td>
</tr>
<tr>
<td>FORNEL</td>
<td>1800</td>
<td>23500</td>
</tr>
<tr>
<td>ALL</td>
<td>4574</td>
<td>44500</td>
</tr>
</tbody>
</table>
Dynamic arrays

The bounds of an array are sometimes ill-defined in the production environment. Scientific programs in particular often use core as a workspace rather than as a specific pre-defined entity. Matrix manipulations may be vector oriented, but type incompatibilities arise when attempting to reference a portion of a matrix rather than the whole matrix. Likewise, it is sometimes necessary in systems-level programming to discard a number of words which are not meaningful to a particular process. Because Pascal does not allow direct referencing of partial arrays, the programmer is faced with the necessity of a type statement of the form:

```
type
  area = packed record
    case BOOLEAN of
      TRUE : ( record1 : ... );
      FALSE : ( record2 : ... );
    end;
```

This construct affects the results of the FORTRAN EQUIVALENCE statement. Unless dynamic arrays are included in the language standard, we urge caution to those who advocate that the tag field be required in variant records. Modifying Pascal to allow dynamic, or adjustable, arrays meets most of the production environment needs and therefore becomes mandatory.

Parameterization of constants

Advocates of Pascal, ourselves included, frequently assert that Pascal programs are portable. This statement is not generally true because of the inability of the programmer to define constants, particularly parameterizing constants, in terms of previously declared constants. Constant definition parts often include annoyances such as:

```
const
  pi = 3.15159265 ;
  two-pi = 6.28318531 ;
```

rather than the statements:

```
const
  pi = 3.15159265 ;
  two-pi = 2.0 * pi ;
```

Nor is the code:

```
const
  high = 256 ;
  highm1 = 255 ;
  highp1 = 257 ;
```

conducive to either portability or reliability. The need to declare, within the constant part only, constants in terms of previously declared constants is mandatory. We believe that the form of constant declaration should allow any type of constant expression—arithmetic, relational and multiplicative operations—upon previously declared constants.

Data initialization

The failure to provide compile-time data initialization is a source of both increased costs and decreased reliability. This problem is particularly important when more than one programmer is involved in development of a system which requires large amounts of initialized data. Both factors make some form of compile-time initialization mandatory.

Declare before use

The enforcement of the "declare before use" rule for procedure and function declarations, together with the requirement for the FORWARD directive, appears to be a design flaw. Even single-pass compilers should be able to recognize the failure to declare a module before the termination of compilation. We believe it desirable to remove the restrictions imposed by this rule with respect to procedure and function declarations.

Termination of comments

One of the more annoying problems associated with debugging a Pascal program is caused by the failure to include (or the accidental removal of) the comment terminator (the Pascal symbols "") or "**/*"). The compiler ignores the balance of the code and usually produces voluminous messages at the end of the listing. There are few errors which are more difficult to diagnose than failure to close a comment because the programmer frequently "sees" comments termination, even though none exists. Although some Pascal implementations warn of a possible error, usually when a semicolon is encountered, we believe the end of source line should act as a comment terminator. We admit that this need can be repaired only by a modification to the language.

For-statement syntax

As an aid to producing structured code, the syntax of the for-statement could be modified. The recommended form would be:

```
"for" (loop-variable) '"'="' (start-value) ("to" | "downto") (stop-value) ['"by" (step-value)] ['"when" (Boolean-expression)] (statement).
```

The failure to provide a means of stepping through values (i.e., the proposed by-clause) appears to be a major cause for error in production code. The contrivances needed to account for non-step-by-one values of the loop control-variable are counter to one of the more important design goals.
of the language: "a systematic discipline based upon certain fundamental concepts clearly and naturally reflected by the language" (pg. 133). Forcing recomputation of a pseudo loop variable within the body of a loop is not consistent with a natural or clear language.

An early escape mechanism is needed in the syntax of the for-statement. The proposed when-clause is a precondition to loop execution in the same manner as the usual precondition required of the current value of the loop control-variable. Therefore, the for-statement body will be executed if, and only if, the Boolean expression yields a value of TRUE and the value of the loop control-variable is within the range of start-value to stop-value, inclusive. We would not require that the declaration of the loop control-variable be external to the loop body. Because the variable's value is usually, and we believe correctly, undefined at termination of the loop, it seems an unnecessary requirement to declare the variable externally to the for-statement.

Standard functions

The lack of standard functions in Pascal requires consideration. The functions provided for numerical analysis (e.g., ABS, ARCTAN, COS, EXP, LN, SIN, SQR, and SQRT) are simply not enough in the production environment. The FORTRAN intrinsics functions (e.g., MIN (choosing smallest value), MAX (choosing largest value), LOG10 (common logarithm), TAN, ASIN (arcsine), ACOS (arccosine), SINH (hyperbolic sine), COSH (hyperbolic cosine), and TANH (hyperbolic tangent) are important in scientific programming. Furthermore, requiring these functions to be developed during a production project raises the specter of accuracy and precision errors. Because many computer systems contain these functions as a part of their vendor-supplied libraries, their repeated recoding is even less desirable. A cogent argument has been made for inclusion of additional operators in programming languages. Additional operators in Pascal would be most welcome.

Input/Output facilities

The I/O facilities of Pascal require major revision and redesign. We will mention only two of our difficulties. Data types, other than the standard types, cannot be displayed directly. A common solution is: define a variant record type, one variant of standard type and the other of the type to be displayed; assign to the type to be displayed variant of the record the value of the variable to be displayed; display the standard type variant. For example, if the current address contained in a pointer was to be displayed, the statements of Figure 3 might be used. The second difficulty with the Pascal I/O facilities is that data cannot be displayed in other than the standard type base. For example, an integer cannot be displayed as some power of eight (i.e., octal). These difficulties have a direct effect upon the work required to perform program debugging. Any improvements would be desirable.

Language implementations

The difficulties derived from implementation are spawned in part by the valid insistence that Pascal-1100 remain as close to the standard as possible. We regard these difficulties as dependent upon the implementation and do not consider it advisable to include them within any Pascal language standard.

Source code inclusion

We found the need to copy source code into a module from another system-known entity to be a mandatory requirement for production implementations. This condition was especially true for type declarations which, in the Pascal-1100 environment, were required in both the calling and called modules (Pascal-1100 supports external compilation). The facility which meets this need is the COBOL COPY verb, although with perhaps somewhat less of a baroque form. Unless a program development group has a powerful text editor, the failure to provide a form of source code inclusion can impact the development schedule.

Identifier names

To the extent practical, it is desirable to have a Pascal implementation accept identifier names of up to the maximum length which can fit on a source line. If a compiler limits unique identifier names to, say, 12 characters, as does Pascal-1100, prefixes which are usually meaningless are bound to be attached to the name. These prefixes tend to reduce readability and increase maintenance costs.

CONCLUSIONS

Pascal is a relatively new addition to the family of production programming languages. We believe that as the language matures Pascal will become accepted as a superior programming tool. The RELATABASE project has shown that the language is easily mastered, yields productivity rates
which compare well with other "older" languages and contains powerful and easily implemented data structuring. The drawbacks to Pascal as a production language probably can be corrected through the current standardization effort.

REFERENCES