This paper discusses and provides some preliminary data on errors in APL programming. Data were obtained by analyzing listings of 148 complete and partial APL sessions collected from student terminal rooms at the University of Alberta. Frequencies of errors for the various error messages are tabulated. The data, however, are limited because they provide no detailed information on how each error type was caused and do not include logic errors. The data indicate that assignment errors are the most common type; and that syntactic and semantic errors are about equally frequent.
BEST-COPY AVAILABLE

RIR-78-1

Programming Errors in APL

Greg P. Kearsley

Division of Educational Research Services
University of Alberta
Programming Errors in APL
Greg P. Kearsley
Division of Educational Research Services
University of Alberta

In recent years there has been considerable interest in the study of computer programming by cognitive psychologists, computer scientists, and human factors specialists. Psychologists have been interested in programming as a complex problem-solving task which reveals important aspects of human information processing capability (either in individuals or groups). Computer scientists have mainly focused on those aspects which pertain to language design and implementation. Human factor specialists have been concerned with improving the productivity and quality of programming activity (e.g., training, error reduction). While each group of scientists has a somewhat different perspective due to their particular orientation, there has been considerable interaction as research interests transgress traditional disciplinary boundaries.

The study of programming errors is an example of a topic within this research area which has interested cognitive psychologists, computer scientists and human factor specialists. As a consequence, a number of different approaches have been employed. Brown & Sampson (1973) suggest some techniques for avoiding or reduce programming

1. Two pioneering works in this area were Sackman (1970) and Wienberg (1971).
errors on the basis of experience in business. Youngs (1974) collected various types of protocol data from a group of novice and experienced programmers on assigned tasks in ALGOL, BASIC, COBOL, FORTRAN and PL/1. A comparison of the relative importance of different types of errors for the different languages and in terms of programming experience was made. Miller (1974) used an experimental language and non-programmers to test the relative difficulty of certain control structures (conjunctive vs. disjunctive, affirmative vs. negative tests). Gould (1975) studied the debugging processes of experienced FORTRAN programmers and compared the debug times and errors missed for different types of errors. Mayer (1975) studied the effects of using diagrammic models during the teaching of a simplified version of FORTRAN.

These studies are representative of the current approaches to the study of programming errors in coding and debugging. A number of major variables have been identified; the two most important being programming experience and the nature of the task or problem. Different types and patterns of errors arise from inexperienced and experienced programmers, and also from different application or problem domains. Other variables such as time-sharing versus batch programming, use of programming aids, or use of CRT versus hardcopy are also known to affect the nature of programming errors. Some attempts have been made to formulate theories of programming (e.g., Gould, 1975; Shneiderman, 1977) and to relate the study of programming behavior to general
psychological research (e.g., Cooke & Bunt, 1975).

While many languages (both real and experimental) have been studied, errors in APL programming apparently have not received attention. This is somewhat surprising since APL is commonly considered one of the most powerful of the presently existing high level languages. Certainly, it is one of the most widely used languages today. Furthermore, APL errors should be particularly interesting because of the canonical nature of the error message types. Saal & Weiss (1977) provide a comprehensive and interesting study of APL usage; however, they did not report error data in this study. The present paper provides a discussion and some preliminary data on errors in APL programming.

**Error Types in APL**

Youngs (1974) classified programming errors into 4 broad categories:

- (a) **syntax errors** which result from expressions which are incorrect regardless of the context in which they appear (e.g., an unmatched parenthesis)

- (b) **semantic errors** which derive from invalid combinations of operations

- (c) **logical errors** which produce incorrect results but do not cause malfunction of the program

- (d) **clerical errors** due to oversight or carelessness such as mispunched cards, missing cards, exceeding page/line limits, etc.
In APL, syntactic, semantic, and most clerical errors occur as one of three different types:

1. Immediate execution errors which produce one of the eight explicit error messages: SYNTAX, VALUE, INDEX, RANK, LENGTH, DOMAIN, DEFN, or ENTRY ERROR, as well as an indication of their location in the expression. For example:

   \[(A-B+C)\]
   \[\text{SYNTAX ERROR}\]
   \[(A-B+C)\]

2. Errors during the execution of a defined function. These errors produce an explicit error message as in (1) and also suspend execution of the function at the line in which the error was detected. For example:

   \[\text{SYNTAX ERROR}\]
   \[\text{DEMO[1]} (A-B+C)\]

3. Errors in the use of system commands or variables. These errors are concerned with manipulating workspaces, functions or variables or resource allocation. The explicit error messages are:

   \[\text{INCORRECT COMMAND}\]
   \[\text{WS/OBJECT NOT FOUND}\]
   \[\text{NOT SAVED}\]
   \[\text{SYMBOL TABLE/WS FULL}\]
   \[\text{SV IMPLICIT ERROR}\]

   The exact form of these error messages differs between versions of APL and installations (i.e., these are the most system dependent messages).
It is interesting that a language with a relatively large number of defined primitives has a relatively small number of error messages. However, the six messages, VALUE, RANK, DOMAIN, LENGTH, INDEX, and DIPN cover the basic semantic errors possible. A major reason for this is their generality across the major data types (i.e., scalars, vectors, and matrices), as well as across constants and literals. Thus while errors due to invalid data types are possible in other languages (e.g., DECLARE or REAL statements), they are not in APL because of the dynamic allocation of variable types. Since APL has no special statements for subroutine calls, errors of this type cannot arise. If a subroutine is given incorrect arguments (the equivalent of an incorrect parameter list), a VALUE error would occur. Other errors such a bad branch (e.g., to a missing label) or failure to initialize a loop counter will also result in VALUE errors.

Because of this generality, multiple causality of error messages is common in APL. Figure 1 illustrates this problem. In Figure 1a, a student defined a monadic function A with the argument FIB (which does not appear in the function. Upon executing A, a SYNTAX error was produced since the occurrence of A in line 1 lacks an argument. The student then tries executing the argument of the function but again receives a SYNTAX error, this time because there is no function relating the two constants. Although the same error message was generated, the nature of the errors are different. Furthermore, this error message does not reveal
the real problem, namely that the student doesn't understand function headers. Figure 1(b) shows another example with system commands. Both of these errors are examples of clerical errors although they generate different error messages.

APL Error Frequencies

Data on error frequencies in APL was obtained by analyzing listings of 148 complete and partial APL sessions collected from student terminal rooms at the University of Alberta. These listings included the work of both novice and experienced programmers and a variety of applications areas. The mean duration of these sessions was 31.4 minutes (maximum: 246.6 minutes, minimum: 2.1 minutes). The mean CPU time per session was 2.6 seconds (maximum: 52.2 seconds, minimum: 0.1 second). There was an average of 8.36 errors/session.

Table 1 presents the frequencies of errors for the various error messages. First of all, it can be seen that the 8 immediate execution errors accounted for over 90% of all error messages while workspace error messages (due to system commands) accounted for less than 9% of the total. Within the first category, VALUE and SYNTAX error messages account for over half of the messages. DEFN and ENTRY error messages were also relatively common. DOMAIN, LENGTH, RANK,

---

2. The version of APL was APL/VS running on an AMDAHL V6 under the ATS operating system.
and INDEX error messages occurred relatively infrequently. As far as the workspace error messages are concerned, INCORRECT COMMAND accounts for about half of these errors with WS NOT FOUND accounting for about 25% of the total. There were no instances of system variable error messages in the sample.

Many instances of some error messages were due to the same error. For example, a large percentage of DEFN errors have to do with the closing square bracket of the function line number. It is commonly omitted or a round bracket used by mistake (this being the same key shifted). The majority of INCORRECT COMMAND errors are due spelling or spacing errors and most WS NOT FOUND errors appear to occur due to a forgotten workspace name (WS NOT FOUND errors are typically followed by )LIB). A large percentage of DOMAIN errors are attempts to divide by 0. On the other hand, VALUE and SYNTAX errors arise from a number of different problems and this probably accounts to some extent for their popularity.

The failure to assign values to variables (which result in VALUE errors) and unmatched parenthesis (SYNTAX error) were two common problems. Many errors were due to misunderstandings about the header in defined functions. These misunderstandings included (i) the unnecessary duplication of the function name in the first line or elsewhere in the function (sometimes resulting in unexpected recursion), (ii) the use of different variable names in the body of the function than those used in the header, (iii)
putting the arguments in the wrong position in the header producing a function with an unexpected name (see Figure 1a), (iv) the redundant use of quad for input when the function arguments already assigned values to those variables, (v) the output of function results when they were automatically produced due to the explicit result form of the header. These misconceptions can result in almost any of the explicit error messages; although typically they produce either a VALUE or SYNTAX error.

A number of debugging strategies were observed in the analysis of the data. The most common strategy was the systematic decomposition of expressions, i.e., testing each set of operations working from right to left. Another common technique in debugging defined functions was to rebuild new functions using working parts of earlier functions. Various types of "retry" behavior were observed quite frequently. The most common one was simply to retype exactly the expression which produced the error to see if it generates the error again. Another "retry" behavior was to CLEAR or sign-off and then start over again. This later approach was common for novice programmers.

Conclusions

The data presented in the preceding section is quite limited in what it reveals about errors in APL. It provides no detailed information on how each error type was caused, say in terms of particular operations or algorithms. More
importantly, this data does not include logic errors (which generally do not produce error messages). Because the characteristics of the programmers was not known (i.e., their experience) nor the nature of the programming problem, the effects of these variables is not known. Finally, it likely that the errors generated in a student programming environment would differ from a commercial or production environment.

The data does indicate that assignment errors (which would generate VALUE errors) are probably the most common type of error made in APL as in other languages. It also appears that syntactic and semantic errors are about equally frequent in APL. The data also reveals that DEFN errors are much more common than one would expect while RANK, DOMAIN, LENGTH, and INDEX errors are less common than anticipated, although the present data does not firmly establish this conclusion. It seems possible that the syntactic complexity of APL expressions leads to more syntactic errors than in other languages. In so far as subscripting is a frequent operation in APL, it is interesting that INDEX errors are not more common (although errors in subscripting could generate rank or length errors). The misunderstandings associated with the function header in defined functions seems a unique problem of APL without an exact parallel in other languages perhaps suggesting a need for language design changes.

As well as contributing to a better understanding of
the programming process and the design of computer languages, information about programming has two major practical uses. The first use is in the teaching of the programming language. For example, the present data on APL errors, suggests that increased attention should be given to the presentation of assignment and the form of function arguments. The second use is in the coding and debugging of APL programs. Given a knowledge of the most likely errors, increased effort can be made during the coding and checking of programs to prevent these problems. While at some point in the future we may have automatic correction of errors and program proving, at the present time, both of these uses of information on programming errors is of some importance.
REFERENCES


Figure la. An example of the same error for different reasons.

Figure lb. An example of the same error producing different error messages.
### Table 1.

APL error frequencies.

<table>
<thead>
<tr>
<th>Function Execution</th>
<th>Total Errors</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>VALUE</td>
<td>317</td>
<td>25.6</td>
</tr>
<tr>
<td>SYNTAX</td>
<td>314</td>
<td>25.4</td>
</tr>
<tr>
<td>DEPA</td>
<td>217</td>
<td>17.5</td>
</tr>
<tr>
<td>ENTRY</td>
<td>137</td>
<td>11.1</td>
</tr>
<tr>
<td>DOMAIN</td>
<td>50</td>
<td>4.0</td>
</tr>
<tr>
<td>LENGTH</td>
<td>34</td>
<td>2.7</td>
</tr>
<tr>
<td>BANK</td>
<td>34</td>
<td>2.7</td>
</tr>
<tr>
<td>INDEX</td>
<td>28</td>
<td>2.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Workspace Management</th>
<th>Total Errors</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>INCORRECT COMMAND</td>
<td>52</td>
<td>4.2</td>
</tr>
<tr>
<td>WS NOT FOUND</td>
<td>26</td>
<td>2.1</td>
</tr>
<tr>
<td>NOT-SAVED</td>
<td>9</td>
<td>0.7</td>
</tr>
<tr>
<td>NOT FOUND</td>
<td>7</td>
<td>0.5</td>
</tr>
<tr>
<td>WS FULL</td>
<td>5</td>
<td>0.4</td>
</tr>
<tr>
<td>STACK FULL</td>
<td>3</td>
<td>0.2</td>
</tr>
<tr>
<td>NOT COPIED</td>
<td>2</td>
<td>0.1</td>
</tr>
<tr>
<td>SYMBOL TABLE FULL</td>
<td>2</td>
<td>0.1</td>
</tr>
</tbody>
</table>
APPENDIX

This report is part of a study of APL programming intended to provide the foundation for a computer-based APL problem solving laboratory. Such a laboratory would permit the student to write and debug APL programs under the control of a powerful APL tutorial system. The Stanford Basic Instructional Program (SBIP) system provides one model of how such a system can be designed and what capabilities can be provided. The SBIP system features:

(a) a monitored BASIC interpreter written in SAIL which allows the instructional system complete information about student errors

(b) a Curriculum Information Network (CIN) which describes a large number of programming problems in terms of the basic skills involved in each. Problems for solution are selected from CIN using a model of the student's previously acquired skills.

(c) a hint/help system which gives graphic and textual aid during problem solving.

Another approach is the PLATO IV CAPS system which is a table-driven diagnostic compiler/interpreter. CAPS resembles batch diagnostic compilers (such as PL/C) except that instead of trying to recover from detected errors, it reports errors and attempts to help the student repair them interactively. Because CAPS is table-driven, it is possible to have CAPS work for all languages for which tables exist. At present tables exist for FORTRAN, PL/I, and COBOL. The CAPS system consists of an edit-time and run-time error analyzer, an editor, a file manager, and the error table and interpreter for each language. It also features a "common misconception table" which contains information about the language which is a potential trouble spot and templates for the help to be provided if that problem arises.

Regardless of which approach is used in the design of a problem solving laboratory, a considerable amount of detailed information about programming errors in the target language must be known. In the present case, this means a reasonably complete list of the of the skills involved in learning APL. Table 2 provides such a list. This skill list provides a basis for the APL programming problems to be developed and the concepts/procedures to be taught. It also forms the basis for the derivation of a set of errors which could arise in learning or performing these APL skills. This set of error rules will be the next step in the development of an APL programming laboratory.
### Table 2
#### Skills in APL

1. **Simple Operations**
   - Use arithmetic primitives alone (scalars)
   - Use arithmetic primitives in combinations (scalars)
   - Use arithmetic primitives alone (vectors)
   - Use arithmetic primitives in combinations (scalars)

2. **Simple Assignment** (scalars and vectors)
   - Assign numeric scalars to variables
   - Display numeric scalars
   - Reassign numeric scalars
   - Assign numeric vectors to variables
   - Display numeric vectors
   - Reassign numeric vectors
   - Assign literal strings (vectors) to variables
   - Display literal strings
   - Reassign literal strings

3. **Simple Indexing** (Vectors)
   - Display all elements of vector
   - Replace (reassign) selected elements of vector

4. **More Complicated Assignment and Indexing**
   - Assign numeric values to matrices
   - Display numeric values in matrices
   - Reassign numeric values in matrices
   - Assign literal values in matrices
   - Display literal values in matrices
   - Replace literal values in matrices

5. **More Complicated Operations**
   - **Numeric**
     - Find min/max
     - Find floor/ceiling
     - Find powers/square roots
     - Find absolute value/residue
     - Find combinations/factorials
     - Generate random numbers
     - Sort in ascending/descending order
   - **Selection**
     - Select using membership/index
     - Select using grade up/grade down
     - Select using take/drop
     - Select using compress/expand
   - **Restructuring**
     - Restructure using reshape
     - Restructure using catenate
     - Restructure using laminate
     - Restructure using transpose
     - Restructure using rotate
     - Restructure using reverse
Relations
Compare numeric arrays using equalities/inequalities
Compare literal arrays using equalities/inequalities
Compare numeric arrays using logical relations
Compare literal arrays using logical relations
Translation
Translate number bases using encode/decode
Translate data structures using execute/format

5. Algorithms
Alter execution order using parentheses
Rewrite expressions to remove parentheses
Write algorithm to compute means
Write algorithm to do sorts
Write algorithms to produce graphs
Write algorithms to do text editing
Write algorithms for statistical functions

6. Defined Functions
Function Definition
Create and execute niladic function
Create and execute monadic function without explicit result
Create and execute monadic function with explicit result
Create and execute dyadic function without explicit result
Create and execute dyadic function with explicit result
Display & Editing
Display entire function
Display selected lines
Modify lines and display
Delete lines and display
Insert lines and display
Add lines and display
Branching
Use slash for unconditional branch to line number
Use slash for conditional branch to line number
Use slash for conditional branch to label
Use arrow for conditional branch to label
Use n-way conditional branch
Use computed branch to line number
Iteration
Build single loop using line numbers
Build single loop using labels
Build nested loops
Demonstrate recursion
Eliminate branching via structured programming
Input/Output
Use quad for input
Use quad for output
Use quote-quad for input
Use quote-quad for output

18
7. Workspaces
   Save workspace
   Load workspace
   Copy variables, functions from workspace
   Copy variables, functions from public library
   Clear workspace
   Drop workspace
   Rename workspace using WS1D
   Use Lib, VAAS, PNS
   Change printing precision
   Change page width
   Change index origin