AN ANALYSIS OF RUN-TIME ERRORS IN PASCAL PROGRAMS

by

Richard J. LeBlanc
Charles N. Fischer

Computer Sciences Technical Report #384

April 1980

An Analysis of Run-Time Errors in Pascal Programs

Richard J. LeBlanc[1]

Charles N. Fischer[2,3]

Abstract

The results of an experiment in which run-time errors in Pascal programs were recorded and analyzed are reported. A surprisingly large number of run-time errors in a wide variety of categories were observed. The implications of these statistics on compiler implementation and programming language design are discussed. The utility and importance of run-time checking mechanisms are confirmed by this experiment.

Key Words and Phrases: Pascal, run-time checking, error distribution, compiler implementation, programming language design

CR Categories: 4.12, 4.22

Footnotes

[1] School of Information and Computer Science, Georgia Institute of Technology, Atlanta, Georgia 30332

[2] Computer Sciences Department, University of Wisconsin - Madison, Madison, Wisconsin 53706

[3] Research Supported in part by National Science Foundation Grant MCS78-02570.

[4] As neophyte programmers often learn to their chagrin!

[5] This is a very painful conclusion to reach in light of the amount of work that went into our implementation of set operations.

[6] Pascal 6000 [5] does this limited kind of pointer checking.

[7] Since *eof* may be false even though no more data remains to be read.

1. Introduction

The UW-Pascal compiler [1] is a diagnostic compiler for the programming language Pascal [5] developed at the Madison Academic Computing Center of the University of Wisconsin-Madison. Among the unique features of this compiler is the variety of errors it can detect at run-time. In particular, the compiler generates code to monitor the use of pointers and variant records, in addition to including the typical arithmetic and range checking. The motivation for these checks and the implementation techniques used have been previously reported [2,3]. This paper describes the results of a six-month experiment which monitored the actual distribution of run-time errors in some 24,000 Pascal program executions.

2. The Experiment

The run-time support routines for the UW-Pascal compiler were modified to make a record of all errors detected during the execution of Pascal programs. Additionally, via sampling, an estimate was made of the total number of Pascal executions during the course of the experiment. Sampling was used rather than an actual count in order to reduce costs and to avoid problems with contention for the files used to record the data. One-tenth of the total executions were actually sampled.

For each program to be counted, a record was made of its initiation and its termination. An estimate of the total number of runs is easily obtained from the initiation count and the sampling rate. Terminations were also recorded in order to detect certain errors (e.g., infinite loops) which are not trapped by the Pascal support routines. The difference in the initiation and termination counts can be used to estimate the total number of such errors.

Tables 1, 2 and 3 present the data which were collected. As the tables indicate, the UW-Pascal compiler can operate in two different modes. Its basic mode of operation is to generate a standard relocatable binary module which can be linked with support routines, library routines and separately compiled Pascal routines [6] for execution. As an alternative for those programs which only need be linked to Pascal support routines, the compiler can generate code in core and cause it to be executed directly without a linkage step. The data for the two modes are separated in the tables because it is conjectured that the counts presented, for the most part, represent two different groups of users and that significant conclusions can be drawn from the differences detected between the two groups. The data for load-and-go compilations is further subdivided in order to examine the utility of proceeding with the execution of programs known to contain source errors.

3. Explanation of Error Types

The number of distinct error messages which can be produced by the UW-Pascal compiler is considerably larger than the number of entries in the tables. The following list details the grouping of run-time errors we have chosen. Brief descriptions of unusual errors detected by the compiler are also included.

The error groups are:

(1) arithmetic error - This category includes such standard errors as zero division, real overflow, and attempts to convert a real number too large for integer representation.

(2) subrange error - These errors represent attempts to assign an illegal value to a subrange variable.

(3) subscript out of range - A reference to an array using an index outside of the declared bounds of the array occured.

(4) set range error - Two different errors are included here: assigning an out-of-range value to a set variable as well as attempting to create a set including an element beyond the limits imposed by the implementation.

(5) no label for **case** expression - This error occurs when a **case** statement has a selector which does not match any of the labels.

(6) illegal reference to variant - Pascal requires that when a reference is made to one of the fields of the variant part of a record, the tag field must have a value which matches a label of the referenced variant.

(7) pointer error - Every use of a pointer is checked to detect attempts to use unintialized pointers, pointers with a **nil** value and pointers which point to objects which have been disposed.

(8) attempt to change a locked tag - The tag field of a variant record may not be changed when a field of a variant is being used as a **var** parameter or a **with** record (see [2,3]).

(9) attempt to dispose a protected object - A dynamically allocated object may not be disposed when the entire object or

any part of it is being used as a **var** parameter or a **with** record (see [2,3]).

(10) error in standard routine parameter - These errors indicate violations of the restrictions placed on parameters to standard procedures and functions (e.g., a negative value as a parameter to chr).

(11) attempt to read past end of file - A call to read, readln or get occurred when eof was true.

(12) attempt to read an ill-formed number - An attempt to read an integer or a real found an illegal sequence of characters.

(13) illegal call to I/O routine - This group includes trying to read on a file not open for reading, trying to write on a file not open for writing, and calling reset on input or rewrite on output.

(14) reference to unassigned external file - A reference was made to a file variable for which there was no corresponding external file assigned to the run.

(15) stack or heap space overflow - A call to new or the invocation of a procedure of function discovered that needed space was unavailable.

(16) separate compilation error - Execution of a program built from incompatible separately compiled modules [6] causes this error (only possible for relocatable compilations).

(17) user limit error - This category involves a variety limitations on execution such as limits on execution time, total cost, pages printed, etc.

(18) error in source code - UW-Pascal allows execution of programs containing source code errors; execution is

terminated if an erroneous statement is reached or an incorrectly declared variable is referenced.

4. Analysis of Results

The most striking results of this experiment are the wide distribution of errors and the unexpectedly high run-time error rates which we observed. 28% of all relocatable runs, 57% of all load-and-go runs without source errors and 81% of all load-and-go runs with source errors terminated with some run-time error. Indeed, the true error rates must be higher still since programs which do not induce a run-time error can still compute an incorrect result[4]. Of course, errors are to be expected in a university computing environment. Nevertheless, the moral is clear: a programming system must anticipate user errors as the norm, not the expection. The automatic detection and isolation of run-time errors is an all but indispensible component of such a system.

The error distributions reported in the tables confirm our expectation that different groups of users tend to use the two versions of UW-Pascal. Particularly significant is the far smaller percentage of the runs using the relocatable compiler which terminate in error. Though this difference is influenced to some extent by use of the relocatable version for "production" work, the vast majority of the runs recorded were for academic class assignments. Another indication of the difference in user populations is the differing frequencies of subscript errors, variant errors, end of file errors and references to undefined external files. The significance of each of these categories of errors will be considered below. In general, however, the differing frequencies apparently reflect significantly different skill levels on the part of programmers as well as different patterns of feature use.

The data in Table 3 confirm the utility (for debugging purposes) of allowing the execution of programs with source errors. Almost 20% of such executions resulted in normal termination. This indicates that only parts of these programs not affected by the source errors were exercised by the test data. Such successful executions are certainly helpful to the testing process. Of those executions terminating in errors, slightly more than half involved errors other than the source code errors. Since the distribution of these errors is much like that for load-and-go compilations without source errors, it is apparently the case that useful information is being obtained about errors other than those detectable at compile-time.

Runs using both versions of the compiler exhibit a high frequency of subscript errors. The much higher frequency for load-and-go compilations is probably a reflection of less sophisticated programmers using arrays as their primary data structures. In any case, the sheer quantity of these errors strongly demonstrates the necessity of subscript checking. Debugging is far more difficult when erroneous subscripts can potentially have unrestricted effects, rather than resulting in an immediate error message. Of further interest is the relatively small number of subrange errors in comparison to subscript errors, especially for load-and-go executions. This comparison seems to indicate that subrange variables are not being used as effectively as possible, particularly by the less sophisticated programmers.

The extremely small number of set range errors indicates either that sets are very easy to use or that they are not used very much. The fact that none at all were made by users of the load-and-go version of the compiler tends to support the latter conclusion[5]. It seems that the primary use for sets involves set constructors (or set variables with a constant value) in boolean expressions. Such use probably does not justify the inclusion of set types in a language. The **in** operator in Ada [4] is perhaps a good alternative to provide the capabilities required by the most common use of sets.

There are a significant number of **case** statement errors, despite the availability of an **otherwise** clause in UW-Pascal. This is one kind of error which could be completely eliminated by a slight change in the design of the language. If every **case** statement were required to include either an alternative for every possible value of the expression or an **otherwise** clause, these errors could not occur. This restriction is not an undue imposition on a programmer; in fact, it is simply a prudent style of programming.

As with subscript errors, the frequency of pointer errors among all users indicates the necessity of pointer checking. Nearly half of all pointer errors involved references using a **nil pointer**, so requiring initialization and eliminating <u>dispose</u>, as Ada does, does not at all eliminate the need to check the validity of pointers. Among the pointer errors in relocatable executions, about 30% were dangling pointers, emphasizing the importance of checking for such errors rather than merely verifying that a pointer references an address somewhere in the heap area[6].

The importance of variant checking is also confirmed by the data. Variant errors were far more prevalent in programs using the relocatable compiler apparently because variant records received considerable use by the students in a compiler class, who were the dominant users of the relocatable version. In Table 1, we see that variant records were the most misused class of data structures in Pascal. During our development of this compiler, and before we had variant checking available, we found variant errors to be among the most difficult of our debugging problems. Thus, we believe that the frequency with which variant errors occur supports our beliefs in the value of variant checks, probably the most neglected of possible run-time checks in Pascal implementations.

The checks involving locked tags and protected objects were included in the compiler more for the sake of completeness than in the belief that they would catch many errors. The occurence of a few of these errors does indicate the necessity of these checks in providing complete security of access to data objects. However, as described in our previous papers, some run-time overhead and a considerable addition to the complexity of the compiler are necessary to implement these checks. The small number of errors detected indicates that such specialized checking can be removed (as an optimization) without a significant loss of run-time security. It can well be argued that the low probability of occurence of these errors makes the inclusion of such checking in a compiler uneconomical.

The large number of end-of-file errors, especially among the less experienced programmers using the load-and-go compiler, is probably indicative of basic problems with the design of the I/O features in Pascal. Experience in teaching the language has shown us that proper use of eof and eoln is one of the most difficult things for programmers to master. Particularly troublesome is the fact that eof is often misleading when reading numbers from a file of characters[7].

The frequency of errors involving references to unassigned external files points out another difficulty with the use of files in Pascal. The problem in this case seems to be that no standard mechanism is available for specifying in a program the binding of an external file to an internal file name. This design was apparently influenced by the operating system under which Pascal was first implemented, which allows executions to essentially be parameterized by a list of file names. Many other operating systems (includes ours) only provide this capability to a limited extent, resulting in the problems our users have experienced. The easiest solution seems to be to extend reset and rewrite to take an optional second parameter which is a string specifying an external file name (as has been done in several existing implementations).

The extremely large number of user limit errors certainly includes many manifestations of infinite loops, but we have no way to estimate how many. We believe this surprisingly large total mostly reflects the relatively inhospitable program development environment imposed by our operating system.

5. Summary

The data collected in this experiment provide some valuable insights about issues of compiler implementation and programming language design. The importance of subscript, pointer and variant checks is demonstrated by the frequency with which such errors were detected. Problems with the design of **case** statements and I/O features are also apparent from the results. The sheer number of run-time errors observed certainly emphasize the value of run-time checks as a "last line of a defense" between a user and his errors. The question appears not to be "Can I afford run-time checks?" but rather "Can I afford not to have them?".

Work is currently in progress involving the collection and analysis of data on the frequency of occurence of compile time errors in Pascal programs. We expect this experiment to provide equally enlightening insights into the design and usage of Pascal.

Acknowledgements

We are grateful to the Madison Academic Computing Center, and in particular Tad Pinkerton and Manley Draper, for their aid and support in conducting this research.

References

[1] Fischer, C.N. and LeBlanc, R.J. UW-Pascal reference manual. Madison Academic Computing Center, Madison, Wisconsin (1977).

[2] Fischer, C.N. and LeBlanc, R.J. Efficient implementation and optimization of run-time checking in Pascal. SIGPLAN Notices, 12, 3 (March 1977), 19-24.

[3] Fischer, C.N. and LeBlanc, R.J. The implementation of run-time diagnostics in Pascal. To appear in IEEE Transactions on Software Engineering.

[4] Ichbiah, J.D., et.al. Preliminary Ada reference manual. SIGPLAN Notices, 14, 6 (June 1979), Part A.

[5] Jensen, K. and Wirth, N. Pascal User Manual and Report, 2nd Ed. Berlin: Springer-Verlag (1976).

[6] LeBlanc, R.J. and Fischer, C.N. On implementing separate compilation in block-structured languages. SIGPLAN Notices, 14, 8 (August 1979), 139-144.

Table 1

Relocatable Compilations
Total Runs: 13936
Total Errors: 3871
% Errors: 27.78

Error Type	Count	% of Runs	% of Errors
arithmetic error	4	0.03	0.10
subrange error	141	1.01	3.64
subscript out of range	471	3.38	12.17
set range error	8	0.06	0.20
no label for CASE expression	181	1.30	4.68
pointer error	462	3.32	11.93
illegal reference to variant	612	4.39	15.81
attempt to change a locked tag	9	0.06	0.23
attempt to dispose a protected object	4	0.03	0.10
error in standard routine parameter	8	0.06	0.21
attempt to read past end of file	152	1.09	3.93
attempt to read an ill-formed number	128	0.92	3.31
illegal call to I/O routine	37	0.27	0.96
reference to unassigned external file	402	2.88	10.38
stack or heap space overflow	66	0.47	1.70
separate compilation error	78	0.56	2.01
user limit error (estimated)	1031	7.40	26.63
error in source code	77	0.55	1.99

Table 2

Load-and-go Compilations
(without source errors)
Total Runs: 7529
Total Errors: 4262
% Errors: 56.61

Error Type	Count	% of Runs	% of Errors
arithmetic error	20	0.27	0.47
subrange error	129	1.71	3.03
subscript out of range	1249	16.59	29.31
set range error	0	0.00	0.00
no label for CASE expression	244	3.24	5.73
pointer error	463	6.15	10.86
illegal reference to variant	45	0.60	1.06
attempt to change a locked tag	0	0.00	0.00
attempt to dispose a protected object	0	0.00	0.00
error in standard routine parameter	25	0.33	0.59
attempt to read past end of file	579	7.69	13.59
attempt to read an ill-formed number	135	1.79	3.17
illegal call to I/O routine	37	0.49	0.87
reference to unassigned external file	85	1.13	1.99
stack or heap space overflow	75	1.00	1.76
separate compilation error	--	--	--
user limit error (estimated)	1176	15.62	27.59
error in source code	--	--	--

Table 3

Load-and-go Compilations
(with source errors)
Total Runs: 2929
Total Errors: 2361
% Errors: 80.61

ErrorType	Count	% of Runs	% of Errors (I)	% of Errors (II)
arithmetic error	9	0.31	0.38	0.73
subrange error	28	0.96	1.19	2.29
subscript out of range	355	12.12	15.04	28.98
set range error	0	0.00	0.00	0.00
no label for CASE expression	112	3.82	4.74	9.14
pointer error	121	4.13	5.12	9.88
illegal reference to variant	10	0.34	0.42	0.82
attempt to change a locked tag	0	0.00	0.00	0.00
attempt to dispose a protected object	0	0.00	0.00	0.00
error in standard routine parameter	5	0.17	0.21	0.41
attempt to read past end of file	252	8.60	10.67	20.57
attempt to read an ill-formed number	58	1.98	2.46	4.73
illegal call to I/O routine	21	0.72	0.89	1.71
reference to unassigned external file	82	2.80	3.47	6.69
stack or heap space overflow	23	0.79	0.97	1.88
separate compilation error	--	--	--	--
user limit error (estimated)	149	5.09	6.31	12.16
error in source code	1136	38.78	48.12	--

I: "error in source code" included
II: "error in source code" excluded