TAILORING TESTING TO A SPECIFIC COMPILER — EXPERIENCES

Harlan K. Seyfer

SPERRY UNIVAC
Major Systems Software Development Center
P.O. Box 43942
St. Paul, MN 55164

ABSTRACT

The testing of the Univac UCS-Pascal compiler is described. Tests were acquired from various sources, converted from existing tests, and developed in house. Test development and execution using the Univac Test Controller System is illustrated with examples. The experiences gained from this and other compiler testing efforts are described.

1. INTRODUCTION

Since early 1981, the Product Test Development group at the Sperry Univac Major Systems Software Development Center has been involved in test development for a Pascal compiler. This compiler is being written under Univac's Universal Compiling System (UCS), described in [Gyllstrom-79]. To test this compiler, we began with a mixture of test acquisition and development methods based on three ordered goals.

1) Acquire tests wherever possible. Exhaustive testing of a compiler is out of the question, both in terms of complexity and resources. Thus the problem is how to get the most out of the resources available for testing. An obvious solution is to acquire pre-existing tests. Not only does this avoid reinventing those tests, it brings to bear differing perceptions on how to go about testing the software.

2) Convert tests from other languages, which exercise similar, perhaps vendor unique, features. Over the years we have built a store of such tests in several languages. Because of language similarities, most conversions were from preexisting Fortran tests.

Also, there is available [NESC-80] a set of elementary math function tests written in Fortran but readily convertible to other languages.

3) Develop tests to overcome the limitations of acquired and converted tests. There are restrictions on the capabilities of acquired tests. By their nature they must be general enough to run with compilers of differing origin and on a variety of systems [Oliver-75]. Also, assumptions are often made about the "typical" compiler and system [Oliver-79], which are not necessary when tailoring tests to a single compiler.

Home-grown tests bring to bear experiences with previous qualifications of the vendor's compilers and allow concentration on known or suspected weaknesses.

The Univac Series 1100 Test Controller System (TCS) [TCS-80] has helped relieve much of the tedium of test analysis and record keeping. This paper will present a walk through the execution of a sample test and describe the output and reports automatically generated.

Several practical experiences have been gained and observations made during this and other compiler testing efforts. Some of these involved issues that either had to be resolved or their effects minimized. This included such problems as testing "odd corners" of a language, revealing tests in advance, internal or external auditing, assumptions made during testing, the difference between conformance and performance testing, test sizes, and the effects of optimization.

2. THE TESTS

2.1. SOURCES OF ACQUIRED TESTS

After a literature search [Seyfer-82], two Pascal compiler test suites were identified and obtained, the Pascal Validation Test Suite [Freak-82]and a set of syntax error analysis and recovery tests [Ripley-81a]. They differ substantially in their objectives.

2.1.1. The Pascal Validation Suite

The Pascal Validation Suite [Wichmann-80]was developed at the University of Tasmania, Australia, and at the National Physical Laboratory, Teddington, England. Version 2.2 consists of 318 tests designed to support the ISO draft Standard. A part of the documentation of each test is a reference to the relevant Standard section. The tests are grouped into six classes according to what is being tested for: conformance to the Standard, deviance from the Standard, implementation definition, error handling, quality, and extensions. More on this classification scheme is given below.

2.1.2. Syntax Error Analysis and Recovery

Ripley and Druseikis [Ripley-78] analyzed errors in Pascal programs written by students in two graduate computer science classes at the University of Arizona. The object was to determine the most common types of error and the efficiency of various error recovery algorithms. The Pascal syntax error tests resulted from the accumulation of distinct errors appearing in those programs. Data supplied with each test includes documentation as to the location and type of error and the numeric error code of the corresponding error message from Appendix E of Jensen and Wirth [Jensen-74]. Ripley and Druseikis employed a simple method for classifying errors: single missing token, single extra token, single wrong token, and non-single token errors. They also devised a grading system for a compiler's accuracy of diagnosis: diagnosed accurately, diagnosed incorrectly, and diagnosed poorly, in addition to diagnosed late. Since their 127 tests were a distillation of nearly 3000 errors, a weighting factor was assigned each error according to its original frequency [Ripley-81b].

In addition, over 300 assorted Pascal programs, not written to be used as tests, were obtained. They can be considered a random selection. This approach can produce situations beyond the imagination of a test programmer. At the least, this technique produces a sampling of how a subset of users envision Pascal usage. The routines range from 10 lines up to a 4,000 line utility. There are several statistical and math programs and a metacompiler. Several programs were donated by programmers who have finally realized a valid use for those old Pascal programs (e.g. games) they have been saving. Most programs obtained in this manner need some adaptation to be useful as tests; for example, canned input is commonly needed to insure consistent behavior from execution to execution.

After the initial qualification, this accumulation may be pared down to a set of regression tests that found errors and a set of tests exercising somewhat obscure corners of the compiler. The latter can usually be identified before the qualification simply by inspection. Admittedly, obscurity is a somewhat subjective determination. An obscure feature is one that, in the test analyst's opinion, is infrequently used. The set of tests detecting errors obviously does not become apparent until the qualification nears completion. The reduction in the number of tests of this type is necessary to reduce maintenance and execution efforts. Since these tests are a random sampling from the input domain of the compiler, this reduction does serve to limit their randomness by targeting specific problem areas.

2.2. CONVERTED TESTS

In addition to tests written for Pascal, we have in our test library a large body of Fortran compiler tests with features similar to those in the Pascal compiler, e.g. the IF-THEN-ELSE clause. Consideration was given to converting these via the powerful Macro processor [Greenwood-79 and MACRO-81]. However, it was determined that manual conversion would be simpler and less costly for such a one-time endeavor.

2.2.1. Service Subroutines

A set of vendor-unique service subroutines may be provided in Univac Pascal for the programmer's convenience and information, e.g. time and date. In addition, there may be a set of calls referencing operating system functions, e.g. messages to the system log file. Again, these are implemented in Fortran where there already exist sets of tests for these features.

2.2.2. ELEFUNT Routines

ELEFUNT is a Fortran test package for elementary math functions. It was developed by William J. Cody, Jr., at the Argonne National Laboratory [Cody-80 and NESC-80]. Each ELEFUNT program is a test of one or more of the elementary function subroutines generally supplied with the support library accompanying a compiler. Functions tested are ALOG/ALOG10, ASIN/ACOS, ATAN, EXP, POWER, SIN/COS, SINH/COSH, SQRT, TAN/COTAN, and TANH. These tests are easily adapted to other languages; and so, it was a simple manner to rewrite them in Pascal.

2.3. DEVELOPED TESTS

2.3.1. To Cover Weaknesses in Acquired Tests

An examination of the acquired tests revealed several aspects of the compiler insufficiently exercised by them. For example, Version 2.2 of the Pascal Validation Suite contains no tests exercising external procedures, functions, or files. Also, a need for more extensive implementation-defined and dependent checks was perceived. These are discussed in more detail below.

In addition to the errors encompassed by the Ripley and Druseikis tests, it was necessary to create tests for each diagnostic which can be generated by the compiler at both compile and run times. Although these cannot assure that a diagnostic will be called up in all cases for which it should, these do allow a level of confidence that, in at least one instance, the diagnostic can be invoked correctly. [Eggert-81] and [Fischer-80] provided helpful guidance in developing some runtime error tests.

2.3.2. To Cover Weaknesses in the Language

The language Pascal has been widely discussed and debated [Moffat-81]. In particular the paper "Ambiguities and Insecurities in Pascal" [Welsh-77] has proven very helpful in checking such problem areas as: type equivalencing, scope rules, set constructors, variant records, functions and procedures as formal parameters, and range violations. Using the paper as a starting point, tests were created to probe the language weaknesses described. Here test development preceded compiler development such that as the compiler became available the tests were executed not to give a pass/fail indication, but to determine how the compiler would behave when presented with difficult to diagnose problems. Once this behavior is determined, appropriate steps can be taken to either document the behavior or alter the compiler. The test itself then remains as a regression test, if it compiled and executed correctly; otherwise, it becomes a test of the diagnostic and error recovery system. Care was taken to minimize redundancy between these tests and the implementation-definition section of the Validation Suite. These are good tests of compiler robustness.

Example #1 is an sample of the source for one of these tests. It is based on a comment in [Welsh-77]. (The examples can be found in the appendix.)

3. TEST EXECUTION

3.1. PHYSICAL ORGANIZATION OF THE TESTS

Tests at the Major Systems Software Development Center are organized into test packages. There is a package control element holding information applicable to every test in the package, e.g. the files the compiler and libraries are located in. Each test has a JCL element, one or more source elements, and a base element. The base element holds the correct (we hope) results of a successful execution of the test. This is dynamically compared to current output on future runs of the test.

3.2. THE TEST CONTROLLER SYSTEM

Our test execution vehicle is the Univac Series 1100 Test Controller System (TCS) [TCS-80]. TCS is a set of routines that provides the capability for administering a test or group of tests in either a dedicated or production environment.

The TCS Controller routine, interactively with the user, selects the test or tests to be executed, determines which files contain the processors and/or libraries to be tested, and chooses the desired processor options. The runstream generated by the Controller assigns the necessary files, logs the test in a status file, compiles and executes the test, compares the test output with a predetermined test result, and updates the status file as to whether the test passed or failed. The Controller can be instructed to ignore expected differences.

The TCS Status routine examines the test comparison results to determine whether differences occurred in compilation, linkage, or execution. Currently active tests are timed so that loops or unexpected terminations can be identified. All of this information, along with processor level information and the amount of time to execute the test, is recorded in the status file. The Status routine generates summary reports based on this data.

A detailed walk through a test execution and the resulting reports is presented in the appendix.

4. EXPERIENCES GAINED AND OBSERVATIONS MADE

4.1. TESTING IN ODD CORNERS OF A LANGUAGE

A limitation is imposed by the effort to exercise "odd corners" of a language [Goodenough-80b, Grune-79, and Wichmann-76]. Two comments on this follow. First, there are more such "corners" then can be seen by test developers. These tend to be "odd" only in a relative sense. In other words, what is tested is affected by the, perhaps unique, test developers's style of programming and their perceptions of what should be tested. Small test development groups with homogeneous backgrounds tend to amplify this problem.

Secondly, as a user community changes, so does its usage of a programming language. As new needs evolve, programmers are motivated to exploit features of a language in ways perhaps not foreseen by test developers (or compiler writers for that matter). The inevitability is that error reports will come in from the field. Most test authors recognize this and attempt to maintain their tests in tune with current requirements.

In general a solution to both problems is to cull a variety of programs from an existing user community. A good cross section of tests should prove invaluable as it represents how a subgroup of users envisions using the language. In lieu of a preexisting user community a test development group of heterogeneous backgrounds is helpful. It is the mixing of diverse perspectives which is necessary.

4.2. REVEALING TESTS IN ADVANCE

Because of their public nature, published test sets are known beforehand. Does their disclosure in advance have a negative effect? Grune [Grune-79], in discussing the Mathematics Center Algol-68 test set, states, "In my opinion, if a compiler processes the test set well and works well on the daily stream of average programs, it is a very good compiler. Through its unusual complexity, the test set will uncover most incorrect short-cuts, and the constant use of simple features will prevent the compiler from being too much tuned to the test set." Wichmann and Jones [Wichmann-76]somewhat ambiguously come to the conclusion that tests should not be revealed. Although, they suggest that "to disclose any tests which are only a small sample of all the possibilities would merely provide a useful tool to compiler writers."

At an early stage in UCS-Pascal testing, the development group was relying almost exclusively on the Validation Suite. This was necessary since it was immediately available while the other tests were still in development. After a small subset of the other tests became available, the compiler was achieving a 90.7% success rate (206 passes out of 227 tests, excluding tests for features unimplemented at the time) with the conformance and deviance tests of the Suite and a 45.9% success rate (45 passes out of 98) with similar tests in the other group. Judging from the nature of the tests and the types of problems detected, this was not a matter of a few bugs causing many tests to fail. This is strong evidence that at that point the compiler had become tuned to a particular set of tests.

Similar statistics for other test sets are not available. Nonetheless, that such tuning can exists with such a highly regarded set of test as the Pascal Validation Suite is highly illustrative of the problem. Obviously, the compiler upon its release was completely tuned to our tests. Furthermore, it may be interesting to observe that, viewing the user community as the ultimate set of tests, the compiler will become tuned to that group. Problems reported by new users differ from those reported by old users.

4.3. TEST SIZES

Our experience, and that of others [Goodenough-80b and Wichmann-76], indicate that the purposes of testing are in general better served by many short tests rather than a few large tests. Short tests tend to pinpoint errors more exactly. They also avoid the problem of error masking, which occurs when more than one error can be detected by a test, but, because of a severe error occurring first, succeeding errors are not detected until the first is fixed and the testing cycle begun again.

Short tests do have drawbacks. They restrict testing for undesirable interactions among language features, do not place a strain on the capacity of the compiler, and tend to make the job of examining test output laborious. The last problem can be minimized by an appropriate test audit tool on successive runs of the same tests. The first two drawbacks can be avoided by employing test routines, lengthy when necessary, directed specifically at those relatively well defined problems. Our experience is that only one to five percent of tests need be large, i.e. more than say 50 lines.

In addition, while describing the Mathematics Center Algol 68 tests, Grune [Grune-79] mentions an interesting circumstance of testing: "At least one quarter of the errors uncovered by the test set ... were accidental discoveries." A test can uncover an error other than the one it was intended to detect. This unplanned but desirable effect occurs more frequently with large tests than with short.

4.4. OPTIMIZATION

Tests mathematical in nature may behave differently when compiled with optimization than when compiled without. This, clearly, is due to the rearrangement of machine instructions producing two distinct computational sequences. It should also be obvious that the amount of difference, if it exists, will be dependent upon the nature of the function processed, the arguments fed it, and whether it is the math library or main routine which is optimized. This can cause problems if one is trying to maintain a record of correct results. Because of this, we keep tests exhibiting such behavior separate, with as many records of valid output as necessary.

4.5. INTERNAL AUDITING VERSUS EXTERNAL AUDITING

Should a test be internally or externally audited? In the former case, the test itself evaluates the results and gives a pass/fail indication. This is usually accomplished with the algorithm:

```
IF <test result> = <expected result>
THEN WRITE 'PASS'
ELSE WRITE 'FAIL'
```

An externally audited test outputs <test result> to a file. This is manually or semi-manually verified once, then verified using a file comparator on successive test executions. This brings into play two assumptions: that the first examination is correct and that the file comparator works correctly. The present Univac file comparator has been around since the early sixties and is in a language, assembler, other than the one being tested, so the latter assumption appears valid. The manual examination may appear to present problems. However, it has valuable advantages if the test is written by one person and the output examined by another. So, the former assumption too is workable, constituting a form of code review. The result is that external auditing helps minimize the problem of a test wrongly passing. A set of tests intended for portability among vendors, such as those available for checking the language conformance of a compiler, can not assume that a compiler, other than the one being checked, is present. Because the comparator would have to be written in the language tested, the tests themselves might as well be self-checking to simplify effort and avoid potential problems.

4.6. TESTING FOR CONFORMANCE OR PERFORMANCE

A distinction can be drawn between testing for conformance to a standard and testing for overall performance quality. Testing for conformance is a legitimate goal, but should not be confused with testing for quality, which requires a more rigorous approach. The primary visible quality indicators are efficiency and numeric accuracy.

Efficiency can be measured in terms of space and time requirements. Accuracy also is quantifiable, but standards rarely define it. R.S. Scowen and Z.J. Ciechanowicz in their survey of compiler testing [Scowen-80]discuss the vagueness of standards in dealing with accuracy. John V. Cugini [Cugini-81] covers the numerical accuracy problem in depth. The ELEFUNT tests [Cody-80] are an excellent example of accuracy tests where they are most needed.

4.7. BASIC ASSUMPTIONS FOR TESTING

A set of assumptions is always made about features that are required to work before testing can proceed. Refer, for example, to [Cugini-80] and [FCVS-78]. There are three approaches to this issue: 1) set up a hierarchy of features based upon those required to function properly before succeeding features can be tested, 2) set up one test checking the basic assumptions and required to be executed before the others, and 3) ignore any basic assumptions. With the last approach, several tests may fail because an implied assumption was not satisfied. Unfortunately, the cause for failure may not be easily apparent from an examination of one or a few test outputs. The first solution seems to involve more work on the part of test developers than is necessary. A hierarchy can be overly elaborate to the point where the assumptions become the feature details tested. In addition, a large number of detailed assumptions may be unnecessary. The second approach provides a reasonable compromise and a vehicle for quick-look testing.

5. ACKNOWLEDGE- MENTS

Any tester soon comes to realize that his job is made much easier and productive if he has the sincere cooperation of the development programmers whose product he is testing. I have been most fortunate in this regard. Thanks go to Scott Costello, who gathered many of the routines to be used as tests. I am obliged to my colleagues in testing, Max Feuer, Tekla Kridle and Ken Sogge for their valuable comments on this paper and their enlightening conversations. Finally, I thank Pete Klausler for his assistance in producing imaginative tests.

6. REFERENCES

[CCVS-80] COBOL COMPILER VALIDATION SYSTEM (CCVS), VERSION 4.0, USER'S GUIDE (IMPLEMENTATION DOCUMENTATION), Federal Compiler Testing Center, August 1980, available from National Technical Information Service as publication PB80-219900.

[Cody-80] W.J. Cody and William Waite, SOFTWARE MANUAL FOR THE ELEMENTARY FUNCTIONS (Englewood Cliffs: Prentice-Hall, 1980).

[Cugini-80] John V. Cugini, Joan S. Bowden, and Mark W. Skall, NBS MINIMAL BASIC TEST PROGRAMS — VERSION 2, USER'S MANUAL, VOLUME 1 — DOCUMENTATION, NBS Special Publication 500-70/1 (November 1980), pp. 15-16.

[Cugini-81] John V. Cugini, SPECIFICATIONS AND TEST METHODS FOR NUMERIC ACCURACY IN PROGRAMMING LANGUAGE STANDARDS, NBS Special Publication 500-77, June 1981.

[Eggert-81] Paul R. Eggert, "Runtime Checking for ISO Standard Pascal", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. 7, No. 4 (July 1981), pp. 447-448.

[FCVS-78] FORTRAN COMPILER VALIDATION SYSTEM (FCVS78), VERSION 1.0, DETAILED TEST SPECIFICATIONS, Federal Compiler Testing Center, November 1978, available from National Technical Information Service as publication AD-A062-038.

[Fischer-80] Charles N. Fischer and Richard J. LeBlanc, "The Implementation of Run-Time Diagnostics in Pascal", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. 6, No. 4 (July 1980), pp. 313-319.

[Freak-82] R.A. Freak and A.H.J. Sale, PASCAL VALIDATION SUITE — VERSION 3.0, Department of Information Science, University of Tasmania, GPO Box 252C, Hobart, Tasmania 7001, Australia. It can be obtained in North America from: Richard J. Cichelli, c/o ANPA Research Institute, Box 598, Easton, PA 18042. Phone (215) 253-6155. It can be obtained in Europe from: DR. B. Wichmann, National Physical Laboratory, Teddington, TW11 OLW, Middlesex, England. Phone 1-977 3222 EXT 3976.

[Goodenough-80a] John B. Goodenough, "The Ada Compiler Validation Capability", SIGPLAN NOTICES, Vol. 15, No. 11 (November 1980), pp 1-8.

[Goodenough-80b] John B. Goodenough, ADA COMPILER VALIDATION IMPLEMENTERS' GUIDE (Waltham, Mass.: SofTech, Inc., 1980), available from National Technical Information Service as publication AD-A091-760 (October, 1980).

[Greenwood-79] Stephen R. Greenwood, "Macro: A Programming Language", SIGPLAN NOTICES, Vol. 4, No. 12 (December 1979), pp. 80-91.

[Grune-79] Dick Grune (ed.), THE REVISED MC ALGOL 68 TEST SET, IW-122/79-November (Amsterdam, The Netherlands: Mathematisch Centrum {Kruislaan 413, 1098 SJ Amsterdam, The Netherlands}, 1979).

[Gyllstrom-79] H.C. Gyllstrom, R.C. Knippel, L.C. Ragland, and K.E. Spackman, "The Universal Compiling System", SIGPLAN NOTICES, V14 N12 (December 1979), pp64-70.

[Jensen-74] Kathleen Jensen and Niklaus Wirth, PASCAL USER MANUAL AND REPORT, SECOND EDITION (New York: Springer-Verlag, 1974).

[MACRO-81] UNIVAC SERIES 1100 MACRO PROGRAMMER REFERENCE MANUAL, UP-8336.1, Sperry Univac, St. Paul, Minnesota (1981).

[Moffat-81] David V. Moffat, "Index to the Periodical Literature — 1981 Pascal Bibliography (June, 1981)", SIGPLAN NOTICES, Vol. 16, No. 11 (November 1981), pp. 7-21.

[NESC-80] ELEFUNT (FORTRAN ELEMENTARY FUNCTION TESTS), NESC Abstract 881 (Argonne, Illinois: National Energy Software Center, Argonne National Laboratory, 9700 South Cass Avenue — 1980).

[Oliver-75] Paul Oliver, TRANSFERABILITY OF FORTRAN BENCHMARKS, available from National Technical Information Service as publication AD-A039-741 (January 1975).

[Oliver-79] Paul Oliver, "Experiences in Building and Using Compiler Validation Systems", PROCEEDINGS OF NATIONAL COMPUTER CONFERENCE, AFIPS (1979), pp. 1051-1057.

[Ripley-78] G. David Ripley and Frederick C. Druseikis, "A Statistical Analysis of Syntax Errors", COMPUTER LANGUAGES, Vol. 3 (1978), pp. 227-240.

[Ripley-81a] G. David Ripley, ERRONEOUS PASCAL CODE, David Sarnoff Research Center, Princeton, NJ 08540 (1981). This is a set of Pascal syntax error analysis and recovery tests.

[Ripley-81b] G. David Ripley, PASCAL SYNTAX ERROR DATA, report accompanying test programs (1981).

[Scowen-80] R.S. Scowen and Z.J. Ciechanowicz, COMPILER VALIDATION — A SURVEY, NPL CSU Technical Report No 8/81, National Physical Laboratory, Teddington, Middlesex TW11 OLW, United Kingdom (December 1980).

[Seyfer-82] Harlan K. Seyfer, "Compiler Test Sets", to appear in SIGPLAN NOTICES (1982).

[STP-80] SERIES 1100 SYSTEM TEST PACKAGE (STP) LEVEL 2R1, SRA-365. Sperry Univac Marketing, 3001 Metro Drive, Suite 300, Minneapolis, Minnesota 55420 (1980).

[TCS-80] USER GUIDE, SPERRY UNIVAC SERIES 1100 TEST CONTROLLER SYSTEM (TCS), LEVEL 2R1, RRD-A446.2, Sperry Univac Marketing, 3001 Metro Drive, Suite 300, Minneapolis, Minnesota 55420 (1980).

[Welsh-77] J. Welsh, W.J. Sneeringer and C.A.R. Hoare, "Ambiguities and Insecurities in

Pascal", SOFTWARE — PRACTICE AND EXPERIENCE, Vol. 7 (1977), pp. 685-696.

[Wichmann-76] B.A. Wichmann and B. Jones, "Testing Algol 60 Compilers", SOFTWARE PRACTICE AND EXPERIENCE, Vol. 6, No. 2 (April-June 1976), pp. 261-270.

[Wichmann-80] B.A. Wichmann and A.H.J. Sale, A PASCAL PROCESSOR VALIDATION SUITE, Report CSU 7/80, (Teddington, England: National Physical Laboratory, 1980).

7. APPENDIX TEST EXECUTION EXAMPLES

Perhaps the best way to demonstrate the mechanics of Sperry Univac's method of testing a compiler is to step through a sample test execution. The sample will be for the test using the source shown in Example #1.

Example #2 illustrates how a demand user might interact with the Test Controller. In lines 1 and 2 the user calls the Controller specifying the test package (PCRT) containing the tests.

At lines 4 and 5 the user states that he wishes to have the test run in demand mode. If he had intended to execute several tests, he could have specified that the tests be run in batch.

In lines 6 through 29 the Controller queries the user for options and system features. An element exists in the test package file containing the queries and the default values.

Line 31 warns that this is a test package still undergoing development.

At lines 32 and 33 the user specifies the test to be executed. Line 34 confirms that the test selection has proceeded successfully. The Controller could have returned with a message saying the selected test can not be run, because certain environmental conditions are not met. For example, the test could be restricted to running on certain types of machine.

At lines 35 and 36, if the default files are not desired or if he would like to see what they are, the user would type "no". The Controller would display the default files and ask which are to be changed.

At line 37 the Controller assigns the user a reference number to be used in obtaining test status reports, an example of this report will be given later.

Lines 38 through 44 inform the user of the stages of test execution as they are initiated. At lines 38 and 39 test execution begins. Lines 40 and 41 state that the statusfile for test package PCRT is being updated with an entry for the test executing. This update is for the start time of the tests. This information is valuable for tests causing the software being tested to enter an infinite loop or to hang.

Lines 43 and 44 report that the test has been executed and that the comparison with the base is proceeding. The results of the comparison are given at lines 45 and 46. In this sample there were differences in compilation, but none in execution.

In lines 47 through 51 the Controller queries the user for what information he wants on the individual test report. Line 47 asks if a report is desired. If the reply is "no", obviously no report is generated. If the answer is "yes", as here, a report is created and lines 1 through 46 in Example #3 are added to the report. When the reply to the query at line 49 (Example #2) is "yes", a second query follows, asking for the form of the compilation listing. After the reply at line 52 (Example #2), lines 47 and 48 (Example #3) are appended to the report. A hardcopy of this report is then usually obtained for further analysis of the test session.

Example #3, as stated, is the test result for a single test. Lines 1 through 20 contain the outcome of the comparison between this test execution and a previous test execution. This example shows that during the previous execution the compiler did not diagnose the discrepancy in type declarations. When the second execution occurred, the compiler did diagnose it correctly. Lines 13 through 16 indicate with a <T> the lines occurring during the second execution, but not during the first. Lines 11 and 12 are the sign-on line of the compiler. Since such information as the time and date will always be different, the Controller was instructed to ignore these lines. For the user's information, the Controller instruction accomplishing this is reproduced at line 5.

There are two comparisons indicated by a divider of double quotes and bracketed number at lines 7 and 17. The file comparator was instructed, at line 3, to compare only the results UPAS processor operation and, at line 4, the XQT processor. These instructions also state that for reporting differences UPAS is the compiler and XQT is the execution. Two facts which are immediately obvious to the reader, but not to the Test Controller.

Line 33 contains the processor (DRED) call, which inserts the features selected by the user in lines 6 through 30 in Example #2. Since a minimal compilation was requested, only diagnostic material is produced by the compiler at lines 35 through 38 in Example #3. (To shorten the example, the sign-on and sign-off lines have not been included.)

Because the user had asked for a long compilation of the source code at lines 51 and 53 in Example #2, a long compilation listing is appended following line 46 of the report.

Example #4 is a sample of a status report for an entire package. The report, for the most part, is straight forward. The status of the test used in the preceding example is given at lines 19 through 21. The test was executed at 10:25:00 a.m. on 10 June 1982. The test base was created using UPAS level OR1T1 and the test was executed with level OR1T2. The test required 000:22 SUPS (Standard Units of Processing) or about 22 seconds of CPU and I/O time.

7.1. EXAMPLE #1 — SAMPLE TEST SOURCE

```
{ The language imposes an ordering on the different classes }
{ of declaration within a block.  However, there is nothing }
{ forbidding declaration after use within  the type         }
{ definition.  This should be no problem to a two-pass      }
{ compiler.  A one-pass compiler should diagnose correctly  }
{ Refer to Welsh, Sneeringer, and Hoare, p688.              }
PROGRAM main (output) ;
  type
    matrix = array [1..10, 1..10] of complex ;
    complex = record
                realpart, imagpart: real
              end ;
  var
    M     : matrix ;
    ii,rr : integer ;
begin ;
  writeln ('For a single-pass compiler, an error should have
            occurred in the TYPE section during compilation.') ;
end .
```

7.2. EXAMPLE #2 — SAMPLE OF TEST EXECUTION IN DEMAND

```
1.  << user types "@tcs$.controller pcrt"
2.          pcrt is name of test package >>
3.  FTC 3R1.85   06/10/82 15:37:08  CREATED  01/22/82 09:27:20
4.  MODE?
5.  << user types "demand" (for this example) >>
6.           SPECIAL EDITING CONSIDERATIONS FOR  PCRT
7.           PASCAL COMPILER CALL OPTIONS MAY BE SPECIFIED.
8.           COMMA MUST PRECEDE OPTION(S); E.G., ",S"
9.           ' ' ( BLANK IN SINGLE QUOTES ) ==> NO OPTIONS.
10.          ENTER PAS OPTION(S):
11. ----- DEFAULT  IS ' '    ---->
12. << user presses XMIT key for default >>
13.          MAP OPTION MAY BE SPECIFIED.
14.          ' ' ( BLANK IN SINGLE QUOTES ) ==> NO OPTIONS.
15.          ENTER MAP OPTION(S):
16. ----- DEFAULT  IS S      ---->
17. << user presses XMIT key for default >>
18.          TYPE OF LIBRARY MAPPED MUST BE SPECIFIED:
19.                FOR                                 I ENTER
20.             -----------------------------------I------
21.             NON-REENTRANT LIBRARY               I  NR
22.             CONFIGURED COMMON BANK LIBRARY      I  CC
23.             NON-CONFIGURED COMMON BANK LIBRARY  I  NC
24.          ENTER TYPE OF LIBRARY MAPPED
25. ----- DEFAULT  IS NC     ---->
26. << user presses XMIT key for default >>
27.          OPTION KEYWORDS MAY BE SELECTED.
28.          ENTER OPTION KEYWORDS:
29. ----- DEFAULT  IS NOOPTIONS·  ---->
30. << user presses XMIT key for default >>
31. PCRT: DEVELOPMENT
32. TEST NAME?
33. << user types "ai-scop-1" >>
34. AI-SCOP-1 WILL BE RUN.
35. DEFAULT FILES OK?
36. << user types "yes" >>
37. YOUR DEMAND USER NUMBER IS: 05
38. @MSG,N COMPILE, MAP, AND XQT OF AI-SCOP-1 FOLLOW (IN BRKPT)
39. @BRKPT PRINT$/TESTRESULTS
40. @ESTAT:SYSTEM$.STATUS,L  PCRT
41. FTS 3R1.86    06/10/82 15:55:30  CREATED  02/03/82 14:11:47
42.  ***  AI-SCOP-1  FROM  PCRT  ***
43. SDFCOMP 3R1.34   06/10/82 15:55:31  CREATED  02/03/82 14:11:
44. **** END SDFCOMP ****
45. THERE WERE DIFFERENCES IN COMPILATION.
46. THERE WERE NO DIFFERENCES IN EXECUTION.
47. KEEP LISTINGS?
48. << User types "yes" >>
49. COMPILATION LISTINGS?
50. << User types "yes" >>
51. DEFAULT OPTIONS ARE L.  WHAT OPTIONS?
52. << User presses XMIT key for default >>
```

7.3. EXAMPLE #3 — SAMPLE INDIVIDUAL TEST REPORT

```
1.  SDFCOMP 3R1.34   06/10/82 11:20:58
2.  SDFCOMP   RESULTS.AI-SCOP-1/BASE,TESTRESULTS.,PTCFILE
3.   COMPILER UPAS
4.   EXECUTION XQT
5.   IGNORE UPAS 1,9 BEGIN UCS
6.  **** SDF COMPARE INITIATED **** - BY SDFCOMP -
7.  """""""""""""""""""""""""""""""""""< 1 >""""""""""""""""""""""""""
8.                              SDF COMPARE FOR PROCESSOR UPAS
9.          @PAS$.UPAS          SOURCE$.AI-SCOP-1,RB$.REL,,,NOOPTIONS
10.         @PAS$.UPAS          SOURCE$.AI-SCOP-1,RB$.REL,,,NOOPTIONS
11. <BI     BEGIN UCS PASCAL OR1T1 02/10/82 15:52:49
12. <TI     BEGIN UCS PASCAL OR1T2 06/11/82 11:20:47
13. <T>     **ERROR(MAJOR) 10  Error in type    v
14. <T>     **REMARK(CLARIFICATION) 10  Scanning resumes here
15.          after last error with this number           v
16. <T>     3  matrix = array [1..10, 1..10] of complex ;
17. """""""""""""""""""""""""""""""""""< 2 >""""""""""""""""""""""""""
18.                              SDF COMPARE FOR PROCESSOR XQT
19.         @XQT  RB$.ABS
20.         @XQT  RB$.ABS
21. @HDG ****  TEST OUTPUT FOR PCRT TEST   ****  AI-SCOP-1
22. FTS 3R1.86   06/11/82 11:20:42  CREATED  02/03/82 14:11:47
23. @ADD,LP FESTPAS*PCRTJCL.AI-SCOP-1/JCL
24. :DOCUMENTATION
25.  << documentation appearing in the test's JCL >>
26. @ASG,T  RB$.,F/10//500
27. READY
28. @ERS   RB$.
29. FURPUR 28R2T2 S74T11 06/11/82 11:20:45
30. END ERS.
31. @SYSTEM$.DRED
32. DRED 3R1.47    06/11/82 11:20:46  CREATED  07/16/81 09:01:51
33. @ADD,LP   DRED$.      .
34. @PAS$.UPAS          SOURCE$.AI-SCOP-1,RB$.REL,,,NOOPTIONS
35. **ERROR(MAJOR) 10  Error in type    v
36. **REMARK(CLARIFICATION) 10  Scanning resumes here
37.  after last error with this number           v
38. 3  matrix = array [1..10, 1..10] of complex ;
39. @MAP$.MAP,F   SOURCE$.MAPNC,RB$.ABS
40. END MAP.  ERRORS: 0  TIME: 7.081
41. @XQT  RB$.ABS
42.  << The execution of the test is performed as a check >>
43.  << on runtime error recovery and diagnostics.        >>
44. @FREE   RB$.
45. READY
46. @BRKPT PRINT$
47. @HDG ****      LISTING(S) FOR PCRT TEST ****  AI-SCOP-1
48.  << A long listing of the source element appears here >>
```

7.4. EXAMPLE #4 — SAMPLE TEST PACKAGE STATUS REPORT

```
1.  TEST CONTROLLER SYSTEM STATUS PRINTED AT 13:40:06 ON  6-10-82
2.  *** ON THE FLY STATUS -- DEMAND MODE ***
3.  *** USER NUMBER 62 - RUNNING PASCAL CRITIQUES
4.      (PCRT: DEVELOPMENT) ROUTINES ***
5.    1 TEST(S) PASSED.
6.    2 TEST(S) FAILED.
7.   73 TEST(S) WERE NOT RUN.
8.    3 TOTAL TESTS RUN.
9.   *** TESTS THAT HAVE FAILED ***
10.  AI-RCRD-EQ18  AI-SCOP-1
11.  NO TESTS ACTIVE.
12.  # FOLLOWING TESTS WERE RUN ON AN 1100/60 UNDER EXEC 38R2 #
13.   AI-NAME-EQN3 PASSED.                               10:24:15
14.      ON  6-10-82 ***  BASE UPAS  OR1T1 ; TEST UPAS  OR1T2
15.      ELAPSED SUPS = 000:21
16.  >AI-RCRD-EQ18 VARIED IN COMPILATION-EXECUTION< 10:24:30
17.      ON  6-10-82 ***  BASE UPAS  OR1T1 ; TEST UPAS  OR1T2
18.      ELAPSED SUPS = 000:25
19.  >AI-SCOP-1    VARIED IN COMPILATION               10:25:00
20.      ON  6-10-82 ***  BASE UPAS  OR1T1 ; TEST UPAS  OR1T2
21.      ELAPSED SUPS = 000:22
22.  TESTS WERE STARTED BETWEEN 10:24:15 ON  6-10-82
23.       AND 10:25:00 ON  6-10-82.
```