Technical Report CS74005-E

INTRODUCTION TO PEST CONTROL USING THE WATFIV COMPILER

David A. Ault

Department of Computer Science

College of Arts and Sciences

Virginia Polytechnic Institute & State University

Blacksburg, Virginia 24061

April 1974

As much as 50% of programming time is spent testing and debugging the program. Analyzing a problem for computer solution and developing the algorithm and program structure to be used are intellectually challenging and rewarding and so this time is well spent. The translation of this algorithm into particular statements of a programming language is often tedious, but usually a step willingly taken by one who is interested in seeing the results of his design. These two areas are the main concern of most "programming" courses. Keypunching this program onto cards is a chore which is necessary at this stage of computer related technology, but hopefully it will be removed in the future. The problem of testing and debugging a program is usually left to the student. As a result, many extra hours are spent looking for unexplained errors. Not all of the time spent debugging a program is wasted time. The imperfect human being is likely to make mistakes at all stages of the programming task: from the design logic, through the program coding to the keypunching. Therefore, it is worthwhile to obtain a certain degree of proficiency in debugging programs and this can be learned only by experience. However, obtaining experience armed only with the brief syntax error messages of most compilers and execution error messages of most operating systems condemns the new programmer (and often the experienced programmer) to spending an excessive amount of time correcting programs.

The purpose of this paper is to consolidate into a brief guide procedures to aid in debugging FORTRAN programs which are being executed under the control of a WATFIV compiler. No attempt has been made to include facilities available under other compilers, although many of the principles explained here apply when

using other languages or other compilers. It is intended to inform you of useful practices in debugging and useful facilities of WATFIV, so that your debugging experience will be more profitable and so you will more quickly become proficient at debugging your own programs. Please read it carefully and keep it handy while you are preparing your programming assignments. The procedures presented here are a combination of personal experience with my own program bugs and with those of my students, suggestions of other faculty members and ideas obtained from the references listed at the end of the paper.

The paper is titled "Introduction to Pest Control" because successful program debugging begins before the bug is found by the computer. That is, this paper stresses both the prevention of bugs and effective methods of locating them after they occur. The techniques presented here are fundamental and are important for both the casual programmer and the professional. The professional programmer will be interested in a more detailed study of programming methods (references [4], [6] and [11]) and programming testing and debugging techniques (references [2], [7] and [10]). Mathis [9] describes a course devoted to debugging techniques. The course includes specific facilities which are available now and those that he would like to see developed.

The integer and real constants used in the examples in this paper and the bounds on their ranges apply to the IBM 360 and 370 series computers.

I. Punched card preparation hints.

A. Compare each card with its corresponding FORTRAN statement to be sure you have it punched correctly.
B. Make sure that each statement is contained between columns 7 and 72, inclusive, and that statement numbers appear in columns 2 through 5.
C. Column 6 is left blank except when the card is a continuation card. It is useful to use nonblank characters which will sequence the continuation cards, i.e., place the numeral 1 in column 6 of the first continuation card, the numeral 2 in column 6 of the second continuation card, etc.
D. Be sure that all comment cards have a C punched in column 1.

II. Syntax error messages.

The WATFIV compiler provides the best syntax error messages available to the FORTRAN programmer. Follow the steps below to correct each syntax error.

A. Read the WATFIV error message carefully and determine the error it identifies.
B. Locate the error in the FORTRAN statement. In most cases the error message follows immediately after the statement in which the error was found.
C. Correct the syntax error. You may need to consult a reference manual or a textbook for the correct syntax.

III. Avoiding coding errors.

We include specific suggestions to help you avoid execution errors and/or execution interrupts arising from coding errors. Even though you have made all of the appropriate corrections in the syntax, your program may not execute successfully.

A. Check to be sure that you have spelled each variable consistently. For example, an easily made keypunch error can produce the distinct variables SUM and SUN. The compiler will not catch this error, but execution of the program will either produce faulty answers or an interrupt because of an undefined variable.
B. Check to be sure that each variable is the data type intended. A variable beginning with A through H or O through Z will be a single precision, floating point real variable unless it is declared otherwise in an explicit or an implicit type declaration statement. Similarly, a variable beginning with a letter I through N will default to a full word integer variable unless otherwise declared. If any special type declarations, like DOUBLE PRECISION or COMPLEX, are made, check all of your assignment statements to make sure that all arithmetic and all assignment statements use properly declared variables. For example, if DOUBLE is a double precision variable and SINGLE is a single precision variable, then the assignment statement

```
SINGLE = DOUBLE
```

will truncate the number in DOUBLE from a 16 decimal digit number to a 7 decimal digit number.
C. Check for the possibility of an attempt to divide by zero during execution of your program. If you are not absolutely sure that a divisor will never be zero, place a check for zero in your program and branch around the division step if the divisor should become zero.
D. Check the possible range of all subscripts. If a subscript can increase without bound (or decrease below 1), place a check on the subscript <u>before</u> it is used to identify an array element.

E. If you are using double precision or complex library functions or user written functions, they must be declared in an appropriate type declaration statement.

F. Be sure that the variables or constants which are used as arguments in a subprogram call agree in precision and type with the parameters in the parameter list of the subprogram definition. For example, if the parameter is a single precision variable and the corresponding argument is 10000000., then the WATFIV compiler will interrupt and return an error message since 10000000. is a double precision constant. It can be expressed as a single precision real constant in the form 1E7. As another example, suppose a parameter in a subprogram is declared to be an INTEGER*2 variable and the integer 5 is passed as the corresponding argument in the subprogram call. When using the WATFIV compiler an interrupt will occur, but when using a compiler which does not check for agreement of the argument and parameter, the value received by the subprogram will be zero!

G. Be careful when using a subprogram which changes the value of a parameter in the parameter list of the subprogram definition. If you use a constant as an argument in the corresponding position in the program call, the subprogram will attempt to change the constant. WATFIV will catch the error, however other compilers may not. As a result, the constant will have the new value whenever it is used in the future.

H. WATFIV checks the range of subscripts in subprograms as well as in the main program. Either the extent in each dimension of an argument in the subprogram call and the corresponding parameter in the subprogram parameter list must agree exactly (it is allowed to have the extent of the highest dimension of the array in the calling program to be less than the corresponding extent in the subprogram, but it is best to ignore this potential trouble spot) or the extent in each dimension of the parameter must be passed as an argument in the subprogram's parameter list [1, p. 162].

I. Be very careful when using COMMON. If a COMMON variable can be changed by more than one module of the program, then either an unanticipated execution time change in the variable or program modifications during program development and testing can cause unexpected side effects and bugs that are very hard to locate.

IV. Common arithmetic execution errors and their detection.

A. The DIVIDE CHECK error is caused by an attempted division by zero (see part C of section III).

B. UNDERFLOW occurs when your program generates a nonzero number of modulas less than 10**-78. The standard fixup is to set the variable to zero.

C. Exponent OVERFLOW occurs when a value over 10**75 is generated. The standard fixup is to set the variable equal to 10**75. In some cases overflow and underflow may be avoided by periodically normalizing the numbers used in the calculations.

D. Integer OVERFLOW occurs when an integer is generated of modulas greater than 2**31. Integer overflow is not detected by some WATFIV compilers. The result of this overflow is the number modulo 2**31.

While an exponent underflow is not really an error, it will cause the interruption of program execution unless the programmer takes steps to prevent this action. See the TRAPS subroutine description below.

Although a divide check and an exponent overflow condition are errors, during the debugging stage of program development it might be useful to let the program keep executing to catch other program bugs in the same run. This is made possible by the use of the subroutine TRAPS. The subprogram call is of the form

CALL TRAPS(n_1, n_2, n_3, n_4, n_5)

where the positive integer arguments n_i correspond to the following arithmetic errors:

n_1	Integer arithmetic overflow.
n_2	Exponent overflow, i.e. larger than 10**75.
n_3	Exponent underflow, i.e. less than 10**-78.
n_4	Integer division by zero.
n_5	Floating point division by zero.

Program execution will continue until one of the program errors, say the ith error in this list, occurs n_i times. The subroutine call should be placed at the beginning of the program.

Inhibiting termination of execution is not enough for proper code checking. You need a record of the error even though execution continues. To do this you may use the standard subprograms called OVERFL (for testing arithmetic overflows) and DVCHK (for testing for divide checks).

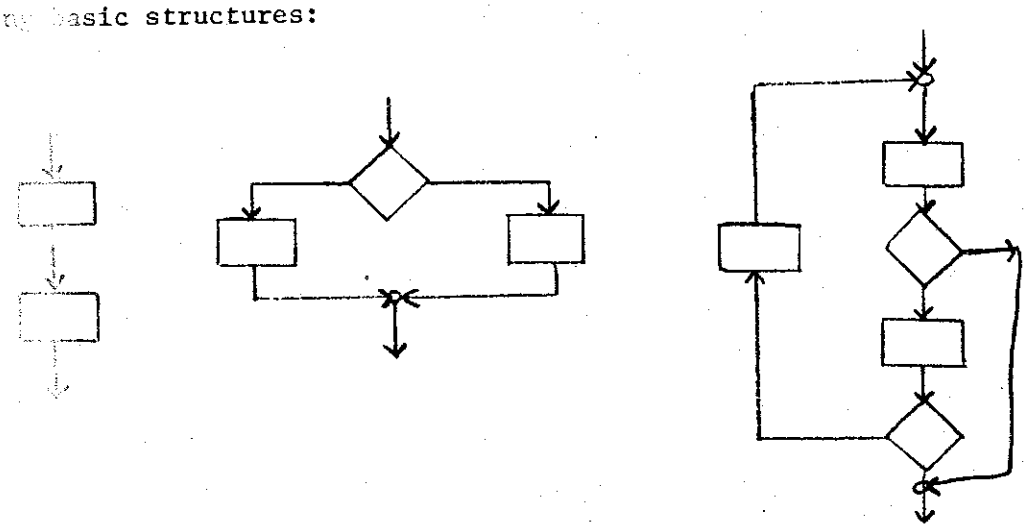
The statement CALL DVCHK(J) tests for the presence of a divide check condition: J is set to 1 if a divide check is detected, J is set to 2 otherwise. The divide check indicator is then "turned off". You can use the value of J in a GO TO (n,m), J statement, where n is the first statement number of the program segment to handle a divide check error and m is the first statement number for normal processing.

Similarly, the statement CALL OVERFL(J) tests for an arithmetic overflow or underflow condition. The value of J upon return from the call is: J is 1 if an arithmetic overflow condition is detected, J is 2 if there was no overflow and no underflow, and J is 3 if arithmetic underflow occurred. The overflow and underflow indicators are "turned off". J can be used in a similar manner as was suggested above to handle these error conditions.

V. General programming techniques to avoid errors and to locate difficult to find bugs.

A. To limit the complication of your program and thus to lower the chance for coding and logic errors, write your program in modular form. That is, isolate distinct parts of your program (even if they are executed just once) and write each part in the form of a subprogram. Use a simple main program to test each subprogram by reading in values, calling the subprogram with these values as arguments, and printing the results. When each part is thoroughly tested, combine them into one program. If each subprogram is, in fact, correct, you need only test the logic and coding of your program that connects the subprogram calls.

B. The second part of producing correct programs is clean coding. For each module, draw a flowchart which is constructed from the following basic structures:



The first two structures are basic structures of smooth [6] or structured [4] programming. The third structure is an adaptation of other smooth programming structures. The author agrees in principle with the goals of these authors, but considers their structures too restrictive. In building your program logic from these structures, a rectangular box may be empty, may contain one operation or may be replaced by any of the three structures. One of the two decision boxes in structure three may be empty. While writing each program module, limit your code to a direct translation of the resulting flowcharts.

References

1. Blatt, John M., Introduction to FORTRAN IV Programming, Goodyear Publishing Company, Pacific Palisades, Calif., 1971.

2. Brown, A. R. and Sampson, W. A., Program Debugging, American Elsevier, New York, 1973.

3. Computer Center User's Guide, Volumes 4 and 10, Virginia Polytechnic Institute and State University, Blacksburg, Va., 1973.

4. Dijkstra, E. W., Notes on Structured Programming, in Structued Programming (APIC Studies in Data Processing No. 8), Academic Press, New York, 1972.

5. FORTRAN Debugging, Computer Center Information Systems Division, Virginia Polytechnic Institute and State University, Blacksburg, Va.

6. Foulk, C. R., Smooth Programming, First Computer Science Conference (Columbus, Ohio), Feb., 1973.

7. Hetzel, William C., Editor, Program Test Methods, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973.

8. Kreitsberg, Charles B. and Ben Shneiderman, The Elements of FORTRAN Style, Harcourt Brace Jovanovich, Inc., New York, 1972.

9. Mathis, Robert F., Teaching Debugging, SIGCSE Bulletin 6, 1 (1974), 59-63.

10. Rustin, Randall, Editor, Debugging Techniques in Large Systems, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1971.

11. Wirth, Niklaus, Systematic Programming, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973.