University of Wollongong
Research Online

Department of Computing Science Working Paper Series

Faculty of Engineering and Information Sciences

1982

A pascal implementation of a display system for pascal programs

Reinhold Friedrich Hille
University of Wollongong

T. F. Higginbotham
University of Wollongong

Recommended Citation

Hille, Reinhold Friedrich and Higginbotham, T. F., A pascal implementation of a display system for pascal programs, Department of Computing Science, University of Wollongong, Working Paper 82-2, 1982, 11p.
http://ro.uow.edu.au/compsciwp/21

Research Online is the open access institutional repository for the University of Wollongong. For further information contact the UOW Library: research-pubs@uow.edu.au

A PASCAL IMPLEMENTATION OF A DISPLAY SYSTEM FOR PASCAL PROGRAMS

*R.F. Hille**
and
*T.F. Higginbotham**

ABSTRACT

A description is given of the design and implementation of a Pascal program for the stepwise visible execution of other Pascal programs. This system operates at the source code level by inserting additional statements into the user program. This additional code causes the stepwise execution of the user program, as well as the display of the statement currently executed together with variables whose values have just changed. This system is intended as both, a teaching aid and a debugging aid. It enables the user to investigate the dynamic properties of his program.

Keywords: debugging, Pascal, visible program execution.

CR Categories: 1.5, 4.4

*Department of Computing Science, The University of Wollongong, Post Office Box 1144, Wollongong, N.S.W. 2500, AUSTRALIA

A PASCAL IMPLEMENTATION OF A DISPLAY SYSTEM FOR PASCAL PRGRAMS

R.F. Hille
and
T.F. Higginbotham

University of Wollongong
Post Office Box 1144
Wollongong, N.S.W. 2500
AUSTRALIA

Keywords: debugging, Pascal, visible program execution.
CR Categories: 1.5, 4.4

1. Introduction

First year students of computing science at this university learn to program in the teaching language Pascal, using the department's computer, a Perkin-Elmer 3220, running under UNIX, version 7. Under this version of UNIX, we have a Pascal interpreter and a "pretty printer" called pxp which can be used to format Pascal programs and also to produce an execution profile. What we do not have is a Pascal debugging system. We have found ourselves spending an inordinately large amount of time helping our students to debug programs.

In order to help beginning students to understand fully the dynamic aspects of their Pascal programs, we usually encouraged them to insert additional write statements, so that they can see after execution what happened to the variables, i.e. the state of the program.

Such a procedure has some shortcomings. Often it is not immediately obvious, from which part of the program the output comes, or too much information appears on the screen in some disorganised fashion, or finally, inexperienced programmers (normally first year students come under that category) may destroy the control structure of their programs by inserting the additional statements carelessly.

We believe that beginning students would benefit greatly from a system which makes the execution of their programs visible on the terminal screen and gives them control over the speed of execution by allowing them to step through their programs. Our purpose is to provide the student user with a display system which will accept any program of any size. For this reason, our system cannot show the entire program at once, but only the line of code currently executed. Such a system deliberately ignores the static aspects of a program and concentrates on its dynamic features, thereby allowing the student to test his program while seing every detail of its execution on the screen.

We decided to develop a display system which takes the user's Pascal program as input and transforms it into another Pascal program containing additional statements, which will have the effect of diplaying the action of the original program on the terminal screen.

To be effective, our system should do the following:

1. Display on the terminal screen the current source statement if it either reflects the control structure of the user program (that is, contains one of the keywords indicating this), or assigns a new value to one of the variables, or reads the value of one of the variables.
2. Display on the screen the name and current value of any variable that has just been changed in an assignment statement.
3. Display on the screen any read statement, together with its line number, that is about to be executed, so that, if reading takes place from standard input, the user has the opportunity to decide what action to take. Output statements should also be displayed, so that the user can follow the meaning of the output by seing tha arguments of "write".
4. Halt execution after each display (except for read statements) and wait for the user to type either a newline character or a space followed by the newline character, depending on the state of the environment (see the next section about implementation).
5. Display statements together with their line numbers to enable the user to refer to a listing of his source code.

The most straightforward way to do this is to operate at the source code level, that is, develop a program which inspects the source code line by line and inserts additional statements into the user program. The code to be inserted depends on the nature of the current statement. We decided to write this system in Pascal, the source code of the user programs, because we wanted to give the interested student the opportunity to understand the system. At this stage of their education, Pascal is the only language that our first year students could be expected to know.

2. Design

Our design follows as much as possible the description of standard pascal given by Jensen and Wirth [1] . The only deviation has to do with the breaking up of programs into lines of code. The details are given in the following description It is quite likely that the user does not want to display the entire program, but only wants to trace through part of it. Therefore, we implemented the commands

{trace on} or (*trace on*)

{trace off} or (*trace off*),

which can be placed around that portion of the program whose execution is to be displayed. These commands must be inserted by the user as additional comment lines. They must be on separate lines in order to be recognised properly. The command "trace on" causes our display system to insert the required additional code until either the command "trace off" is encountered or the user program ends. When the trace flag is off, our system simply copies the user program without change, except for additional keywords "begin" and "end".

In order to avoid the overhead of a scan for tokens, one line of source code is processed at a time. When an opening comment bracket is found at the beginning of a line, a flag is set, suppressing the copying of any code to the new program file until a closing bracket is found. After any opening comment bracket a scan is made for the words "trace on" or "trace off". When the words "trace on" are seen the insertion of additional code is enabled, otherwise the input lines are simply copied, apart from the insertion of "begin" and "end" in certain places (see the description of cases below).

Because each line is processed separately, the Pascal statements of the source code must be on separate lines. For example, the guarded statement

if <condition> then <statement> ;

must be written as

```
if <condition> then
    <statement> ;
```

In the following we describe the operation of the main part of our system. The action taken depends on the presence of certain keywords at the beginning of the current line, or on the presence of ":=" in the current line.

2.1. Keyword "if"

A "write" statement is inserted into the source code which causes display of the current line together with its line number. After that we insert the statement "if not eof then readln;", which will cause execution to halt until some input has come from the terminal. This may be a newline character or any other character.

After these two lines the actual input line is copied. Because the "if" statement may be guarding another "if" or one of the loops that are possible in Pascal, the procedure must call itself recursively at this point to process the next input line. If the next line is "begin", then it must be copied and a repeated recursive call must be made at this point until the keyword "end" is found. If there was no keyword "begin" on the line following the "if", then it must be inserted, because the following single statement may be expanded into a compound statement. After the recursive call to process the single statement the keyword "end" is inserted. This will be followed by a semicolon only if the previous statement had a semicolon.

The line containing "if" is displayed before the test so that the user still sees the reference to the test even if only the alternative is executed.

2.2. Keyword "else"

This case receives the same treatment as the one above, except that the current line is displayed only when execution of the "else" block begins.

For example, the source program may contain the lines:

```
if condition then
   variable1:=expression1
else
   variable2:=expression2;
```

After processing, this would look like:

```
writeln('xxxx  if condition then');
readln;
if condition then
begin
   variable1:=expression1;
   writeln('xxxx  variable1:=expression1;');
   writeln('variable1 =',variable1);
   readln;
end
else
begin
   writeln('xxxx  else');
   readln;
   variable2:=expression2;
   writeln('xxxx  variable2:=expression2;');
   writeln('variable2 =',variable2);
   readln;
end;
```

Here, "xxxx" stands for the line number of the original source code, to be shown on the screen. Note that the keyword "else" may be followed by another "if". In this way a number of alternatives may be strung together, as is illustrated below:

```
if condition1 then
   statement1
else if condition2 then
   statement2
 .
 .
 .
else if condition i then
   statement i
 .
 .
 .
else
   statement n
```

In this case the user must ensure that "else if condition then" is written on one line and not as follows:

```
else
   if condition then.
```

Since our program inspects one line at a time without look-ahead to the following line, it would insert after the "else" the keyword "begin", thereby altering the meaning of the original construct.

2.3. "While" loop

The head of a "while" loop is displayed before its execution begins, so that the user has a reference even if the loop is not entered at all. The body of the loop is enclosed in "begin" and "end" if they are not already present and then the same recursive scheme is used as in the case of the "if" statement. The difference here is that the display code for the head of the loop is inserted again at the end of the loop body, thus ensuring a display at every turn. Also, the statement "readln;" is placed immediately after the keyword "begin", again to halt execution at that point. Care must be taken when the "while" loop tests for end of line. In that case the user must type a character other than "newline" so that eoln does not return "true", thus preventing execution of the loop.

2.4. "For" loop

This case is treated in a fashion similarly to that of the "while" loop except that the display code is inserted inside the loop at the beginning, followed by the display of the name of the loop variable together with its current value.

2.5. Keywords "repeat" and "until"

In the case of the line "repeat" the display code is inserted after the current line, whereas the line "until <condition> " must be preceded by the display code. Execution is halted after the display.

2.6. Keywords "read" and "readln"

Lines beginning with these keywords are preceded by the display code, to enable the user to decide what input to type.

2.7. Assignment statement

An assignment statement is displayed after its execution, followed by the name of the variable on the left as well as its current value. Again, execution halts after the display.

2.8. Keyword "function"

This case requires some special attention, because in Pascal the return statement takes the form of an assignment to the name of the function. If we treated this like an ordinary assignment statement, then the line

"writeln('name =',name);"

would imply a function call. To handle this case, we must use a temporary variable which will take the value to be returned. We used the name "ffff" for the temporary variable, whose form is (we hope) unlikely to appear in students' Pascal programs. This variable will then be substituted for the function name in such assignment statements and an additional statement will be inserted, assigning the value of the temporary variable to the function name. Function declarations may be nested to any depth.

Procedures do not require special treatment. Also, the fact that control is in the body of a function or procedure will be evident from the line numbers appearing in the display.

2.9. Case statement

The display code is inserted immediately before the head of the case statement. It's effect will be to send the output

```
xxxx  case <variable name> of
        variable  =  <integer>
```

to the terminal. If a label is followed by a single statement, the keywords "begin" and "end" are placed before and after it, respectively, because that statement may be expanded into a compound statement. Labels with the statements following then are processed until the keyword "end" is encountered, indicating the end of the case statement.

Any one of the cases described in the sub headings above may be nested to any depth within any other of the cases. The only limit is imposed by the size of the runtime stack provided by the operating system, which will limit the depth of recursion of our procedure "processline".

3. Implementation

A number of considerations have led us to implement the system as a UNIX shell program. Firstly, the name of the user program can then be supplied as an argument to our system, which is not possible for Pascal programs. Furthermore, it is useful to check whether the user program supplied exists at all, and if so, whether it compiles without error. Only when these conditions are met, will the user program be copied to a file named "DDsource". Our program then makes a copy of "DDsource" with the additional code inserted, and writes it to a file called "DDmonitor". Unfortunately, Pascal does not permit file names with the suffix ".p", which is required for the compiler. Therefore, we move the file "DDmonitor" to "DDmonitor.p" before invoking compilation and execution.

After that, the shell program will remove from the user's directory all files created by our system. However, it may be required to leave the Pascal object file resulting from the compilation of the program containing the display code, so that the user may execute it again if he wishes, without having to go through the entire process again.

The code of the UNIX shell program is given in the figure below:

```
if test -r obj
then
        rm obj
fi
if test -r $1
then
        pi -l $1 > DDlisting
        if test -r obj
        then
                cp $1 DDsource
                px display.o
                echo "
PASCAL DISPLAY:
new source file created
"
                mv DDmonitor DDmonitor.p
                pi DDmonitor.p
                px
        else
                echo "
PASCAL DISPLAY:
program contains errors,
listing follows.
"
                sleep 2
                p DDlisting
        fi
        rm DDlisting DDsource DDmonitor DDmonitor.p
else
        echo "
PASCAL DISPLAY:
can't find $1
"
fi
```

Figure 1: the UNIX shell file

The names of the files created by our system carry the subscript DD, so that it is highly unlikely that the user has a file with such a name in his directory. This precaution is necessary as execution takes place in the user's directory. First, any existing Pascal object file "obj" is removed to enable us to check whether the user program actually compiles, which is done by trying to produce executable P-code. There is an additional test to ensure that the file of the name supplied as an argument exists and is readable.

Because of the length of our program (some 900 lines of Pascal), we decided not to include the code here (it is available upon request from R.F. Hille, Department of Computing Science, the University of Wollongong, P.O. Box 1144, Wollongong, N.S.W. 2500). Instead, we describe the algorithm.

The main program consists of a loop of two statements only, namely:

```
while not eof(source) do
begin
        getline;
        processline
end;
```

The procedure "getline" simply loads the next input line into the buffer, whereas the recursive procedure "processline" does the checking and inserting of display code. The algorithm of "processline" is illustrated by the example of one of the cases shown in the next figure:

```
procedure processline

{The flags and the line buffer are global variables}

begin
   check for comment;
   check for keyword at the beginning of the line;
   if commentflag then
        look for "trace on/off"
           and set traceflag accordingly;
        check for end of comment;
   else
        case keyword of
        : if :
           if traceflag then
                echo the current line;
                insert readln statement;
           copy the current line and get the next;
           if that line is "begin"
                copy it;
                while the current line is not "end"
                   get next line;
                   processline;
           else
                insert "begin";
                processline;
                insert "end";
                if line ended in ";"
                   insert ";";

        : else :

{For the other cases see the section on design}
```

Figure 2: algorithm "processline".

4. Usage and Limitations

Any section of the user program that is to be displayed during execution, must be preceded by the comment line {trace on} and followed by the comment line {trace off}. Every Pascal statement must be on a separate line.

Redirection of standard input and output is not possible when the display is on, because that would interfere with the display and the stepwise execution. Execution will halt after each display until the user types either <newline> or <space> followed by <newline>. To use our display system, one types the line:

display <program name.p>

If no file of the given name exists in the user's directory, an error message will be printed. If the program supplied does not compile, its listing with error messages from the Pascal compiler will appear on the screen. Otherwise "display" will be executed and on completion an appropriate message is printed. Thereafter, the newly generated program will be compiled and executed.

The user may exit from loops by typing EOT, but a clean exit will result only from "while" loops which check for end-of-file, otherwise execution will be aborted. In the case of "while not eoln do" the user should type <space> followed by <newline> to avoid immediate exit from the loop.

Procedure names cannot be passed as parameters. This limitation concurs with that given by Joy et al. [2] in their description of the Berkeley Pascal compiler.

The user must not have files with the names "DDsource", "DDmonitor", "DDmonitor.p", "DDlisting" in his directory, as these would be overwritten. No variable with the name "ffff" must be declared in a function of the user program.

5. Conclusion

We have developed a system that enables the user to step through his Pascal program. Any statement that reflects the dynamic aspects of the program, i.e. a control statement or a statement which changes the value of some variable, or an input/output statement, is displayed on the screen together with its line number to enable the user to follow the action with the aid of a listing.

Originally our intention had been only to provide some simple debugging aid. However, as the system evolved, it became clear to us that it was necessary to go all the way and turn this into a display system which can handle all cases. Our only limitations are that we require Pascal statements to be on separate lines and that procedure names cannot be passed as parameters. We hope that our system will help students to understand the dynamic aspects of programming.

References

[1] Jensen, K., Wirth, N. (1975) "PASCAL User Manual and Report", Springer, New York, Heidelberg, Berlin

[2] Joy, W.N., Graham, S.L., and Haley, C.B. (1977) UNIX Pascal User's Manual, Version 1.0