stichting
mathematisch
centrum

MC

AFDELING INFORMATICA (DEPARTMENT OF COMPUTER SCIENCE) IW 172/81 AUGUSTUS

P.W.E. VERHELST & N.F. VERSTER

PEP: AN INTERACTIVE PROGRAMMING SYSTEM WITH AN ALGOL-LIKE PROGRAMMING LANGUAGE

Preprint

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1980 Mathematics subject classification: 68B20

ACM-Computing Reviews-category: 4.13, 4.22

PEP: an Interactive Programming System with an Algol-like Programming Language*)

by

P.W.E. Verhelst & N.F. Verster**)

ABSTRACT

PEP (Program Editor and Processor) is an interactive programming system based on an Algol-like language. It is intended to replace BASIC as a system for interactive program development on small computers (LSI-11).

The language processed by the system allows declaration of variables, constants and procedures; it has structured statements for conditional and repetitive execution of program parts.

We describe design and implementation of the system and give our impressions after one year of experience with the system.

KEY WORDS & PHRASES: Interactive programming, Interpreters, Programming languages

*) This work is supported by the Netherlands Organization for the Advancement of Pure Research (Z.W.O.). Part of this work was done during a stay of the first author at the Mathematical Centre, Amsterdam.
This paper is not for review; it is intended for publication elsewhere.

**) Eindhoven University of Technology, Department of Physics
P.O. Box 513, 5600 MB Eindhoven, Netherlands

INTRODUCTION

We use a simple computer network, consisting of a central PDP-11/23 host computer and several small LSI-11 satellite computers, for the control of physics experiments. Each LSI-11 computer is located at an experiment and is connected to it via interfaces. The satellites are connected to the host computer using serial lines running at 9600 baud. The host computer provides background storage for data and programs, and has provisions for the transfer of data to the central Burroughs B7700 computer of our university. The main tasks of the satellite computers are data acquisition, adjusting experiment parameters, and simple numerical analysis. Data collected by them is transferred to the host computer using a simple line protocol[1]. In general, the satellite computers have no local background storage.

The datacommunication lines we use have an effective transfer rate of about 500 bytes per second. This is acceptable for the transfer of small programs and small amounts of data; it is not fast enough, however, to allow swapping of large amounts of memory. The software running on the satellite computers should support interactive development and execution of programs, without causing too much delay due to slow background storage. An interpreter based system best satisfies this condition because it generally is smaller than a system based on separate edit, compile, and link programs. By keeping the major parts of such a small system resident in memory, most of the commands can be executed without access to background storage.

The only acceptable system available to us for use on the satellite computers was BASIC. Dissatisfaction with the BASIC language and the unavailability of another suitable system have led to the development of the PEP system (PEP = Program Editor and Processor). The objective was to construct an interactive programming environment which provides a better structured language than BASIC, but which retains the attractive properties of BASIC systems.

Related systems are UCSD PASCAL[2] and PASCAL-I[3,4]. These systems are based on compilers that generate code for a virtual machine. Because they rely on background storage to save compiled programs and to switch between compilation and program execution, they were not suited for our applications. Another related system is BASIS[5], an interpreter for a subset of the PASCAL language. The BASIS system comes very close to our system, but is larger and slower.

DESIGN CONSIDERATIONS

Anatomy of BASIC

BASIC systems are very widespread at the moment, especially in the area of personal computers and other small systems. Originally, BASIC was a FORTRAN-inspired programming language, which had to be sufficiently simple for use by business students [6]. We think that the popularity of BASIC is not only due to the simplicity of the language, but also to a number of other attractive properties of BASIC systems as a whole. The most attractive features of these systems are probably the following:

(1) A simple editor is integrated in the command language.

(2) System commands may be freely mixed with directly executed programming language statements.

(3) BASIC is available for small computers that do not have excessive amounts of memory and background storage.

Especially points (1) and (2) make BASIC systems easy to use. It is not necessary to learn complicated command and edit languages in addition to the programming language. The absence of complicated operating procedures will soon give the user the feeling that he has control over the system. The free interleaving of all kinds of commands makes such systems ideal for the gradual development of programs (for example, comparing the effects of simple changes to programs is very easy). The ease of trying out certain constructs will give confidence to the user that he comprehends their meaning.

Although we found the general behaviour of BASIC systems very attractive for our applications, we were less satisfied with the BASIC language. Some of the weaker points are the following:

(1) The absence of procedures makes good program structuring impossible. The GOSUB statement does not help because it does not allow parameters.

(2) The use of short identifiers and line numbers makes programs unreadable and increases the chance of making errors.

(3) The practice to indicate the data type by adding a suffix to the variable name is not reliable. Originally, this was done to distinguish between reals and strings (using a dollar sign). In the version of BASIC that is available to us, a percent sign is used to indicate integer variables. Because of the relatedness of integers and reals, this easily leads to mistakes (I and I% are different real resp. integer variables).

Interactive systems invite the user to program sitting in front of his terminal. Although this practice generally should be condemned, we cannot ignore the fact that it occurs. Programs developed in this way in an unstructured language easily become unreadable and unreliable. We think that programming in a better structured language can increase the readability and reliability of such programs; more errors will be detected because of extra restrictions imposed by the language.

Compilation versus Interpretation

Compilation and interpretation are only two extremes taken from a wide spectrum of techniques available for the implementation of language processors. These techniques differ in the distance between the interpreted code and the original program text. At one end of the spectrum we have interpretation of the program text; at the other end we have compilation of the program to machine language and interpretation of this code by the hardware. Several intermediate forms are possible, in which the program is translated to an intermediate code, which is subsequently interpreted. Examples of such implementations are UCSD PASCAL[2] and EM1 code[7].

In making a choice between the available techniques, there are several factors that must be considered.

(1) A larger distance between program text and code will result in a higher complexity of the implementation.

(2) Program execution speed is higher if more checks are done by the translator instead of by the interpreter.

(3) A lower level code means extra delay between program modification and program execution because of the compilation time in between.

(4) If the intermediate code is invertible, i.e. allows reconstruction of the original program text, storage can be saved by only storing the program code (see also Brown[8]).

We did not consider the lower execution speed of an interpreter based system to be a problem for our applications. In experiment control, speed of program execution is often limited by the data rates associated with the experiment, instead of by the speed of the computer. We were more concerned with the memory requirements and therefore chose for a system based on a high level intermediate code that is invertible and allows a compact representation of programs.

THE PEP LANGUAGE

We had to choose between inventing our own language and using an existing programming language. Arguments for taking an existing language are program portability and easier user education if the language is already known. We expected our applications to be highly nonportable because of their connection to experiments. The only generally known language at our university is a dialect of ALGOL-60 that is available on our Burroughs B7700 computer; this language is too complex to implement on a small machine. These circumstances, and the desire to improve on the languages known to us, were the main reasons for designing our own language. We had the following objectives for the PEP language:

(1) good error detection capabilities

(2) good program structuring facilities

(3) easy to learn (for users with some programming experience)

(4) easy to use in an interactive environment

(5) reasonably simple to implement

How these objectives have been realised will be discussed in the following sections.

Error Detection

Good error detection is especially important in an interactive environment where programs are often changed too easily. Errors are better detected statically (i.e. before program execution) than dynamically (i.e. while executing the program). Static detection does not give the frustration of time lost because of program aborts in the middle of execution. Static error detection is only possible if the language imposes restrictions that force the programmer to supply redundant information to make his programs acceptable. Redundancy in the language is most useful if it increases the ability to detect inconsistencies in programs. We have chosen for the usual restrictions of Algol-like languages, i.e. type checking and obligatory declarations.

Some semantic error sources that are easily avoided are automatic conversion from real to integer and use of the same operator symbol for both real and integer division. Implicit conversion from real to integer may lead to errors if a value is falsely assumed to be integer and assigned to an integer variable. The notation a/b, with a and b of type integer, suggests that a rational (or real) number is produced. We therefore require explicit indication of rounding and let a/b always produce a real result, using a <u>div</u> b for integer division.

Languages like ALGOL-60 and PASCAL have some syntactic peculiarities that can cause errors to go undetected. One example is the subtle effect of the semicolon terminating statements prematurely; compare

```
    if B then S;
```

and

```
    if B then; S;
```

Another example is the nested if-statement of PASCAL,

```
    if B then if B1 then S1
         else S;
```

where indentation suggests a meaning different from the actual one. To obtain the intended meaning, it should be written as

```
    if B then begin if B1 then S1 end
         else S;
```

This kind of errors can easily be prevented by adopting the ALGOL-68 convention of terminating all structured statements explicitly (using fi and od). This has the additional advantage that program modifications are easier: it is not necessary to add an extra begin and end when a single statement is replaced by more than one statement. Also more informative error messages can be given because there is more than one terminator symbol.

Characteristics of the PEP Language

In this section we can only give a brief summary of the PEP language. A complete description of the language may be found in [9].

A PEP program consists of two parts: an (optional) section containing declarations and a section containing statements. These parts are held together using the keywords declare, begin, and end. Possible statements are assignment, procedure call, and the if, while, and for statement. The latter three statements use explicit terminating symbols, fi and od. Possible declarations are constant, variable, and procedure declaration. A procedure declaration consists of a heading, specifying formal parameters and possible return value, and a body, which has the same structure as a program, i.e. allows declarations local to the procedure.

We have included an "on-statement" in the language for the programmatic interception of faults, such as arithmetic exception conditions and hardware errors. The main reason for such a mechanism is that testing for exception conditions after every operation makes a program more unreadable than is necessary. The on-statement has the form

```
on C during S1 do S2 od
```

If during execution of S1 an exception condition occurs, execution of S1 is aborted and execution continues with S2; if no exception occurs, S2 is ignored. C specifies the condition(s) one is interested in. We also provide a procedure "alarm(c)", which raises exception condition "c".

We did not include a goto-statement in the PEP language. Its main use would be to "glue" pieces of program together and this leads to unstructured and unreadable programs.

Input/output operations are available in the same form as in PASCAL, i.e. the procedures "read", "readln", "write", and "writeln" (see [10]). These procedures are nonstandard in that they accept a variable number of parameters. In the "write" and "writeln" procedures formatting specifications can be given. Some additions to the PASCAL facilities that we found useful are tabulation and the possibility to write integers in arbitrary radix. We also adopted the convention to initialise input in an end-of-line state as suggested in [11]. Similarly, we made "readln(f,v1,...,vn)" equivalent to

```
readln(f); read(v1,...,vn)
```

which is closer to the normal use of "readln".

We found that, owing to the use of explicit statement terminators for structured statements, semicolons become superfluous. Insertion or deletion of a semicolon can never change the meaning of a program (in contrast with the first example of the previous section); it can only make a program syntactically incorrect. Semicolons also do not add to the error detection capabilities of the language because they can never point out inconsistencies in the program. Because semicolons can still be helpful in making a program more readable, we did not omit them from the language, but instead have made their use optional.

Because we did not need the complete collection of types available in PASCAL, we could keep the type system simple. All types are built-in and the user cannot define new types. We distinguish between simple types, array types, and structured types.

The most important simple types are "integer", "real", "boolean", and "char". For variables of these types the usual collection of arithmetic and logical operations is available. The simple type "range" is useful for specifying ranges in array declarations and in the for-statement. Values of this type are constructed using expressions of the form "a..b", in which "a" and "b" are integer values. Some additional simple types ("int1", "int2", "nat1", and "nat2") allow efficient storage of small signed and unsigned integers in single bytes or words, which is especially useful for compact storage of arrays.

Array types are specified by giving one or more index ranges and the element type, which must be simple. In addition to the conventional index operation, it is possible to select subarrays by giving a range as index or by only partially supplying indices. The assignment statement can be used to copy the contents of arrays and subarrays.

The system may be configured to contain a number of structured types. These types allow declaration of input/output devices and files. Variables of structured type have an internal structure that is completely hidden; they can only be used as parameter to certain built-in procedures.

The following program gives a general impression of the PEP language. It illustrates some of the operations possible on arrays. A percent sign indicates comment until the end of the line.

```
declare
    inprod = procedure (a,b : array 0 of real) : real;
    % Determine inner product of a,b
    % a,b should have identical index ranges
    declare
        sum : real;
    begin
        sum := 0;
        for i in index_range(a) do sum +:= a[i]*b[i] od;
        return sum
    end

    x : array 1..200 of real;
    corr : array 1..100 of real;
begin
    % Statements to fill x

    for k in 1..100 do
        corr[k] := inprod(x[1..100],x[k..k+99]);
        writeln(k, corr[k])
    od;
    writeln(sqrt(inprod(corr,corr)))
end
```

Static versus Dynamic Binding

Because we have chosen for the possibility of declarations local to procedures, a mechanism must be provided to keep track of the, possibly multiple, meaning of identifiers. In the encoded form of the program identical identifiers are indistinguishable, even if they have their origin in different declarations. The simplest scheme is to keep track of the most recent meaning of each identifier only (dynamic binding). This has the disadvantage that sometimes surprising effects are produced if the same identifier is used in different declarations. Because later declarations take precedence over earlier ones, dynamic binding often takes the wrong meaning of identifiers. The alternative is the static binding scheme of ALGOL and PASCAL, but this is considerably more complicated to implement in an interpreter; this would mean, for instance, that differently bound identifiers should be made distinguishable in the code. We have chosen for dynamic binding, but we impose such restrictions on programs that static and dynamic binding always give the same meaning. The rules we use are as follows.

(1) No local declarations may declare identifiers that are already declared global to a procedure (i.e. no static overdeclarations).

(2) Procedures may not be passed as parameter (and no ALGOL-60 by name parameters).

(3) Forward references (use of identifiers declared further on in the program text) are not allowed.

The first two rules guarantee that it is not possible to call a procedure needing a global variable that is hidden by dynamic overdeclarations; the global variables of a called procedure always are a subset of the global variables at the point of the call, and these cannot have been overdeclared. The third rule guarantees that violations of the first two rules can be detected in a single pass through the program.

We do not consider restriction (1) to be a disadvantage; static overdeclarations can be an important error source, and this is removed by this restriction. We do see problems in restrictions (2) and (3), however. These restrictions make it impossible to write general procedures that depend on one or more user-defined procedures (either passed as parameter or accessed via a fixed name). This problem can be solved because the system supports dynamic binding between separate programs, but this is not a neat solution. Restriction (3) also makes it difficult to write mutually recursive procedures (this is only possible if one procedure is declared within the other).

THE PEP SYSTEM

The PEP system consists of three parts:

(1) the interpreter itself in the form of a procedure that takes an encoded program as parameter;

(2) a table (global environment) supplying all standard declarations available to PEP programs (input/output, data types, standard functions and procedures);

(3) a monitor program for communication with the user.

Parts (1) and (2) are necessary for the execution of PEP programs and must always reside in memory. Part (3) is only needed for program editing and could be removed from memory during program execution; to reduce switching time between editing and execution, however, it is generally held resident.

In the following sections we will describe intermediate code, interpreter, monitor program, and some of the facilities offered by the system.

Intermediate Code

The intermediate code forms the interface between the monitor program and the interpreter. It is also used for the compact storage of PEP programs on disk. The intermediate code is split into a part that is necessary for the reconstruction of program text only and a part that is needed by the interpreter to execute a program. The first part contains information about line division, indentation depth, and comments; the second part consists of program code and identifiers. The program code is a sequence of symbols, which are encoded as follows:

- Keywords (<u>if</u>, <u>begin</u>, <u>end</u>, etc.) and special symbols (":=", "+", ",", etc.) are encoded in a single byte using values in the range 1..99.

- Constant denotations (integer, floating point, string) are encoded as a prefix byte (range 100..127) followed by the value in one or more bytes.

- Identifiers are encoded as bytes in the range 128..254. If more than 127 identifiers are used a multiple byte code is used which starts with prefix 255. The text corresponding to an identifier is stored in a separate table which can be indexed by a number derived from the code. We place no limit on the length of identifiers.

The information contained in the intermediate code allows reasonable reconstruction of the original program text (only the number of blanks between symbols may change). Because the encoding is invertible and not context dependent, the program can be encoded immediately when it is typed in. This reduces storage requirements, both because we need not store the program text and because the code is more compact than the text.

The Interpreter

The interpreter has two functions: the detection of errors and the execution of the program. Because detection of syntax errors during execution is too late, we have split the operation of the interpreter into two passes. The first pass through a program is responsible for checking conformity to the syntax, compatibility of data types, and satisfaction of the restrictions concerning overdeclarations. In addition, it leaves information in the program code to speed up forward jumps during program execution. These jumps occur, for example, in the if and while statement when a condition is false. At all such places room is reserved in the code to store the jump distance. The second pass through the program performs the actual execution; this is done only if the first pass did not detect any errors. Both passes through the code are performed by the same program; a flag indicates whether it is the checking pass or the actual execution.

The interpreter program has the structure of a top-down recursive descent parser (see e.g. Gries[12]). In such a parser there is a procedure corresponding to each syntactic construct. Each such procedure checks that the correct symbols appear in the input text (in our case the program code), and it calls other procedures to handle subconstructs. We will show a simple example to illustrate this. The if-statement

```
if <expr> then <stat> else <stat> fi
```

is recognised by the following parser procedure.

```
if_stat = procedure;
begin
    expect('if'); expr; expect('then');
    stat; expect('else');
    stat; expect('fi');
end;
```

The call "expect(sy)" checks that symbol "sy" appears in the program code and skips that symbol; the procedures "expr" and "stat" recognise expressions respectively statements. We will now show how this parser can be transformed into the first and second pass of the interpreter.

In addition to checking the syntax, the first pass must fill in the jump distances after <u>then</u> and <u>else</u>. This is done using the procedures "fjump" and "flabel"; "fjump(p)" stores the current position in the program code in its argument "p" (a reference parameter) and skips the locations reserved for storing the jump distance; "flabel(p)" stores the jump distance at position "p" in the program code. This gives the following procedure for doing the first pass.

```
if_stat = procedure;
declare p1,p2 : integer;
begin
    expect('if'); expr; expect('then');
    fjump(p1);
    stat; expect('else');
    fjump(p2);
    flabel(p1); stat; expect('fi');
    flabel(p2);
end;
```

The second pass must perform the actions corresponding to the syntactic constructs. For the if-statement this means making a choice between the two alternatives based on the boolean value of the expression. The following procedure handles both passes.

```
if_stat = procedure;
declare p1,p2 : integer;
begin
    expect('if'); expr; expect('then');
    if cjump(p1) then goto L1 fi;
    stat; expect('else');
    if fjump(p2) then goto L2 fi;
L1: flabel(p1); stat; expect('fi');
L2: flabel(p2);
end;
```

The only difference between the two passes is in the operation of "cjump", "fjump", and "flabel". During the first pass "cjump" and "fjump" remember the position in the code and return false. During the second pass "cjump" tests the boolean value that is produced by "expr" and left on a stack. If this value is true, a jump is made in the interpreted code and true is returned; otherwise "cjump" only skips the jump distance that is stored in the code and returns false. We use the same return convention for "fjump" and let it always make a jump in the interpreted code during the second pass. The procedure "flabel" does nothing during the second pass. It should be noted that during the second pass a jump in the interpreted code is always accompanied by a jump (goto) in the interpreter program. There are a number of other differences between the two passes that are not shown here. For example, during the first pass of an expression only the types of the intermediate and final values are determined and no further computation takes place.

Declarations are handled by allocating "declaration cells" on a stack. The current meaning for an identifier is found in an "association table", which contains for every identifier a pointer to the corresponding declaration cell (or nil if undeclared). A declaration cell contains type indication, value, identifier number, and a pointer to the previous meaning for that identifier. On procedure entry, declaration cells are allocated and the association table is updated. On procedure exit, previous meanings are restored and the declaration cells are deallocated.

Identifiers for which no declaration appears in the program itself are bound dynamically. Their meaning is found by searching a system environment, which consists of a collection of association tables. Initially, the system environment consists of a table of standard declarations only (the global environment). Each invocation of the interpreter temporarily adds the declarations of the currently executed program to the system environment. This mechanism allows one program to access the declarations of other programs. Declarations never survive the execution of a program, however.

The interpreter has been implemented as a machine language procedure, which can be invoked using a call of the form "interpret(c,t,n)", where "c" is an array containing the program code, "t" an array containing identifier names, and "n" an integer giving the number of different identifiers. This procedure is accessible to all PEP programs. The availability of the interpreter in this form allowed us to write the remainder of the PEP system in the PEP language itself.

The Monitor Program

The monitor program[13] reads commands typed in by the user and executes them. It provides operations on a "current program", which is kept in a number of arrays. Commands are provided for editing the current program, for executing it, for storing it on disk, and for retrieving stored programs from disk. The most important commands are the following:

LIST	list portion of current program
DEL	delete lines
GET	read current program from disk
SAVE	write current program to disk
RUN	execute current program

These commands suffice for simple use of the system. A complete list of commands is given in table I.

The interpretation of the commands is straightforward. First the syntax of the command is checked; if this is correct, the system checks whether there are any semantic problems; only if the command cannot cause errors will the corresponding action be executed. By keeping to this convention it is guaranteed that the system never enters an inconsistent state, like e.g. a program which no longer has increasing sequence numbers. By checking for obvious user errors, the user is also protected against inconsistent actions. For example, it is impossible to leave the system with an unsaved program or to overwrite an existing file accidentally.

Program lines are entered by typing a sequence number followed by the program text. This sequence number determines the location where the line is inserted in the current program. Any line not starting with a command keyword or sequence number is assumed to be a statement from the PEP language that is to be executed immediately. This is done by encoding it, surrounding it with a begin and end symbol, and passing it to the interpreter as a program.

Table I. Monitor commands

Command	Operation
BYE	terminate PEP session
CLEAR	destroy current program
DEL	delete lines
FIX	replace part of a line
GET	read program from disk
HELP	show commands and standard names
JOIN	insert lines read from file
LIST	list part of current program
LOAD	add library to system environment
MON	switch to different monitor program
MOVE	move lines within current program
RESEQ	renumber program lines
RUN	execute program
SAVE	write program to disk
SEQ	enter automatic sequence mode
SHIFT	change indentation of lines
SHOW	show memory usage
TITLE	change name of current program
UNLOAD	remove library
USER	change user identification

The monitor program has been implemented in the PEP language and can also be invoked from user programs by calling the procedure "monitor" (which has no parameters). In the command mode that then is established, new PEP programs and directly executed PEP statements have access to all declarations available at the moment of calling "monitor". This mechanism can be used for debugging purposes or for interacting with the program by means of the command language itself. Often, this possibility will make it unnecessary for a user to implement a separate command language to control his program. The following skeleton for a simple database program may serve as an example of this use of "monitor".

```
    declare
        data : ...
        read_data = procedure ...
        write_data = procedure ...

        % Declaration of access procedures
    begin
        read_data;
        monitor;
        write_data;
    end
```

This program, when executed, will first read its global data from disk, then allow the user to inspect and modify the data by typing in calls of access procedures, and finally write the modified data back to disk.

Program Modularisation

As explained in the description of the interpreter, programs can communicate with each other via the system environment. This mechanism can be used in two ways. A user can build a standard collection of declarations by placing them in a separate program. The executable statements of this program should contain an invocation of the procedure "monitor". By executing this program, the user establishes a new system environment, which includes his standard declarations. These are then available to all later executed programs.

A second application is a kind of overlay mechanism that reduces memory requirements for large programs. If a program contains a large procedure, its body can be moved to a separate program; the original body can then be replaced by an invocation of this program. This transformation always works correctly and reduces the size of the program in exchange for some extra time necessary for invoking the transformed procedure.

Machine Language Procedures

Machine language procedures are necessary to alleviate the lack of speed of interpreter based program execution and for providing operations on special types of peripheral devices. There are two ways to access procedures written in machine language. The first is via the built-in table of standard declarations, but this only gives access to a fixed collection of procedures. This collection is always resident in memory and is therefore restricted to generally useful procedures.

A second way to access machine language routines is available in the form of a variant of the normal procedure declaration. Instead of a body, the address of a piece of machine code may be given in the declaration. This is illustrated in the following skeleton program.

```
declare
    code : array 0 of nat2;
    fun = procedure (a : real) : real at code[0];
    proc = procedure (a,b : integer) at code[1];
    %
    % Other declarations
    %
begin
    load_code(code,'FILE');
    %
    % Statements using fun, proc
    %
end
```

The procedure "load_code" reads machine code from a file into an array, and it relocates any absolute addresses that appear in the code. We use the convention that a collection of machine language procedures always starts with a table containing their addresses.

Although this mechanism introduces some insecurity into the system (arbitrary machine code can now be executed), it cannot cause any problems as long as well-tested routines are used; both parameters and result are checked by the system in the same way as is done for normal procedure calls.

EXPERIENCE WITH THE SYSTEM

Use of the System

The results obtained with the PEP system are very satisfactory. The system has been used for the programming of several new experiments. Most of this work concerns molecular beam scattering and plasma physics. Examples are the averaging time of flight spectra over long times (up to 24 hours), the frequency stabilisation of a dye laser, and the recording of spectral line shapes using a scanning interferometer. It has also been applied to test interfaces, to solve simple numerical problems, and to automate some of the teaching laboratories for undergraduate students. We intend to use it for all new experiments.

Although the PEP language is less simple than BASIC, users have no difficulties with learning the language. Even beginning programmers with only some BASIC or FORTRAN experience have no problems. A general observation we made is that programmers gain more insight in the problems they are trying to solve because a PEP program always reflects the structure of a problem better.

We find the possibility to nest environments by recursively invoking the interpreter a powerful program structuring tool. Every user can create his own standard environment with procedures and functions that are typical for his application. Global data and constants can also be included in such a user environment. The global data survive execution of erroneous programs and can hence be used to save the intermediate results of an experiment; the global constants allow a user to give names to addresses of devices connected to his experiment. In addition, device specific machine language routines can be included in the user environment. No expertise is required to build such environments; they have the form of normal PEP programs.

The dynamic binding mechanism of the interpreter can be used to alleviate the problems caused by the lack of procedure and call-by-name parameters. An example of this is a program for solving nonlinear least squares problems, which uses a fixed name to access the procedure that gives the application dependent parameter function. Another use of this is made in test programs, which rely on certain standard declarations in the user environment to access the interfaces that should be tested.

There exist two versions of the PEP system: one version runs under the RT-11 operating system and the other version runs stand-alone on an LSI-11 and uses the remote file system of our central PDP-11/23. Originally, we planned a central system for file storage because our budget did not allow us to buy background storage for every LSI-11 computer. We now find that such a configuration has additional advantages. The shared file system makes exchange of programs between users and update of shared programs much easier. It also allows us to provide central news and documentation files to all users.

Time and Space Requirements

Execution speed of PEP is of the same order as that of BASIC. Table II gives a comparison between PEP and BASIC. Machine language constructs for the same operations would need times between 10 and 200 microseconds, which is a factor 10 to 100 faster.

We have also compared the execution speed of the PEP system with the times given in [5] for the BASIS system. For the fastest version of the BASIS interpreter these times are approximately equal to the ones given in table II; however, these times were measured on a faster computer (PDP-11/45) and the PEP system must hence be considerably faster than BASIS. It should also be noted that this fast BASIS version would be too large to fit into an LSI-11; the slower version has about the same size as the PEP system, but is a factor 3 to 4 slower than the fast version.

<u>Table</u> <u>II</u>. Speed comparison of PEP and BASIC
(all times in milliseconds measured on an LSI-11)

PEP		BASIC	
Statement	Time	Statement	Time
i := 0	0.8	I% = 0%	0.7
r := 0	1.4	R = 0	1.3
r := r*r	2.4	R = R*R	2.1
a[1] := 0	2.1	A(1) = 0	4.1
for i in 1..10 do od	7.2	FOR I% = 1% TO 10% NEXT I%	19.2
write('A')	1.9	PRINT "A"	2.6

The first pass of the interpreter checks about 200 lines per second. The delay caused by the syntax check is negligible for most programs. Most compilers would need considerably more time to check a program. The main reason for this difference is that the lexical analysis, which takes much time in a compiler, has already been performed by the editor.

Memory requirements for the PEP system depend on the collection of standard procedures included. The interpreter itself takes about 9 kb, the monitor program takes about 13 kb, and the normal set of standard declarations, which includes input/output procedures and mathematical functions, takes about 16 kb. This leaves 18..22 kb for user programs (assuming 56..60 kb total available memory). Memory can be gained by omitting some of the standard declarations or by switching to a smaller monitor program.

PEP programs are stored in a very compact form. We observe a reduction in program size by a factor 2 because of the encoding used. Storage requirements during program execution are also not excessive. Each declaration takes a fixed amount of 10 bytes (the declaration cell). Only for arrays and structures additional space is necessary.

CONCLUDING REMARKS

We have shown that it is possible to construct an interactive programming system that is much more powerful than the so ubiquitous BASIC systems, without having to give up the attractive properties of these systems. The advantages of the PEP system over BASIC are:

- The programming language is more powerful, especially because of the availability of procedures.

- Programs are better readable because of the structured statements and the more meaningful names of variables and procedures. This benefits the exchange of ideas and programs between users.

- More programming errors are detected before actual program execution, instead of being detected as run-time errors or even not being detected at all. This makes the system more user-friendly and programs more reliable.

The slowness of an interpreter based system is, in our view, completely compensated for by the ease with which such a system can be used and by the fast response to program changes. Because we have also provided an easy access to assembly language procedures, execution speed never is a problem.

REFERENCES

[1] P. Verhelst and J.H. Voskamp, "Software for a Simple Computer Network," DECUS Proceedings Vol. 7(1), pp.239-243 (1980).

[2] K.L. Bowles, "UCSD PASCAL, A (Nearly) Machine Independent Software System for Micro and Mini Computers," ACM SIGMINI Newsletter Vol. 4(1), pp.3-7 (Feb. 1978).

[3] R.J. Cichelli, "Pascal-I - Interactive, Conversational Pascal-S," ACM SIGPLAN Notices Vol. 15(1), pp.34-44 (Jan. 1980).

[4] Pascal News(15), pp.63-66,101 (Sep. 1979).

[5] R.P. van de Riet and R. Wiggers, "Practice and Experience with BASIS: an Interactive Programming System for Introductory Courses in Informatics," Software - Practice and Experience Vol. 9, pp.463-476 (1979).

[6] T.E. Kurtz, "BASIC," ACM SIGPLAN History of Programming Language Conference, pp.103-118 (1978).

[7] A.S. Tanenbaum, "Implications of Structured Programming on Machine Architecture," _Communications of the ACM_ Vol. 21(3), pp.237-246 (1978).

[8] P.J. Brown, "More on the Re-creation of Source Code from Reverse Polish," _Software - Practice and Experience_ Vol. 7, pp.545-551 (1977).

[9] P.W.E. Verhelst, "PEP - Language Description," Report VDF/CO 79-18, Eindhoven University of Technology, Department of Physics, Eindhoven (1980).

[10] K. Jensen and N. Wirth, _PASCAL, User Manual and Report_, Springer-Verlag, Berlin-Heidelberg-New York (1976).

[11] C. Bron and E.J. Dijkstra, "A Discipline for the Programming of Interactive Input in Pascal," _ACM SIGPLAN Notices_ Vol. 14(12), pp.59-61 (Dec. 1979).

[12] D. Gries, _Compiler Construction for Digital Computers_, John Wiley and Sons, New York, N.Y. (1971).

[13] P.W.E. Verhelst, "PEP - Monitor Program," Report VDF/CO 80-08, Eindhoven University of Technology, Department of Physics, Eindhoven (1980).