SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 14(2), 177–190 (FEBRUARY 1984)

The Design of an Interactive Program Development System for Pascal

BENGT NORDSTRÖM AND ÅKE WIKSTRÖM

Laboratory for Programming Methodology, Department of Computer Sciences, Chalmers University of Technology and University of Göteborg, S-412 96 Göteborg, Sweden

SUMMARY

A *program language* can be defined as the language in which computer programs are written, and a *programming language* as the language used by the programmer to create programs. This paper presents the design of an interactive program development system which uses Pascal as both program and programming language. Principal properties of the system are a complete immediate syntax check, a program-structure oriented editor, incremental compiling techniques, and interactive interpretation and debugging of programs. The syntax check is split into three phases, and the user can change the degree of check wanted. After a change of the program only part of it is recompiled, and only necessary phases of the compiling process are performed.

KEY WORDS Interactive programming environment Incremental compiler Pascal Structure-oriented editor Program manipulation Programming methodology

1. INTRODUCTION

Programming habits are most certainly influenced by the first programming language learned.[1] Today, BASIC[2] is perhaps the most used beginner's language. It is easy to learn, but a major reason for its widespread use seems to be the interactive environment offered by most BASIC systems. On the other hand, Pascal[3] is widely accepted as a good language for teaching programming techniques. Therefore, an interactive Pascal system would be very attractive in a teaching environemnt. But certainly, interactive systems are also very attractive for professional programmers.

The objective of this project was twofold. We want to produce a good interactive Pascal environment, but we also want to experiment with strongly interactive program development environments in general. Ideally, such systems should give the user immediate response to all types of errors and maximal support in all programming phases. Several systems with, in some respect, ambitions like these exist today.[4–7]

In the current project we experiment with different system properties that we think are useful to the user. Below is a list of some of them, which we believe may be of special interest:

(a) a uniligual environment saving efforts both for users and implementors
(b) a programmable editor and an environment for program manipulation
(c) a special mode for executing programs immediately when they are inserted
(d) a truly incremental compiler doing a minimum of recompilation for any change.

0038–0644/84/020177–14$01.40
© 1984 by John Wiley & Sons, Ltd.
Received 22 April 1982

Also described are ideas and techniques used to get a well-structured and portable system.

The rest of this paper is organised as follows. We describe a user's view of the system in Section 2. In the next section we outline the design of the system. We explain the importance of horizontal and vertical incremental compiling. Section 4 describes our view of a program development system and the next three sections outline the editor, the compiler and the interpreter of the system. We conclude the paper with some comments on our experiences of using Pascal for a rather big program (the system is currently approximately 25,000 lines of text).

2. THE USER'S VIEW OF THE SYSTEM

The user types Pascal statements on the terminal, and these are immediately executed. The system can, for instance, be used as a desk calculator if the user types statements of the form:

write (3*697-12)

There is no particular command language for the system, as all commands are written in Pascal. It is possible to see the system as a machine which understands and executes Pascal and which has a set of predefined procedures and functions. For instance, the editing commands are a set of predefined procedures working on syntactic representations of programs. The editor has special knowledge about the language Pascal. It uses a cursor that can be positioned at various points in the program. The user can move the cursor forwards and backwards and insert, delete or change a program fragment where the cursor is positioned. The editor will check for errors and will (unless told to do otherwise) not accept a change leading to a syntactically erroneous program. The programmer can choose between three levels of error detection: coarse syntax check, local syntax check and total syntax check.

The coarse syntax check only checks the consistency of reserved words, i.e. that **end** matches **begin**, that **then** precedes **else**, and so on. This is useful when the user wants to use the system to manipulate incomplete Pascal programs, for instance in an early design phase of a project or when importing Pascal programs from an installation which uses non-standard Pascal.

The local syntax check will discover all context-independent errors, i.e. it will discover all errors that are not dependent on information in the declaration lists. This makes it possible to use the system for top-down programming; it is, for instance, possible to use procedures that have not yet been defined.

The total syntax check will check that the program text is a legal Pascal program. So, in addition to the local syntax check, it will check that expressions have proper types.

The user has some syntactic liberties while inserting a program. Declarations can be inserted anywhere where a statement can be inserted, and the declaration is automatically moved to the nearest, appropriate declaration list. The system inserts semicolons between statements on different lines.

An interactive system gives the user many possibilities for easy debugging. The user can stop the execution of a program (either from the terminal or from the program), inspect values of variables (by executing the standard output procedures), change them (by ordinary assignments), edit the program (by procedure calls) and

resume the execution. It is possible to move the execution pointer (program counter) to an earlier instantiation of a procedure, i.e. it is possible to move to the environment belonging to the procedure that activated the current procedure. There will also be primitives for tracing a variable, a statement or a routine.

There is a possibility to save an execution, i.e. store a program and all its variables and their values on a backup memory. The programmer can then log out from the terminal and later come back to the same, partially executed, program.

There is a special mode of inserting programs which we call 'append and execute mode'. In this mode the program is executed at the same time as it is entered at the terminal. The programmer can, for instance, type a procedure call, and this call is immediately executed. If the procedure is not yet declared, the programmer has to enter the procedure. The interpreter then executes the statements of the procedure as they are entered by the programmer. After that the interpreter returns to the procedure call.

3. DESIGN OF THE SYSTEM

3.1. System objectives

In the early design phase a set of general principles and guidelines for the design were agreed upon. We will first describe this set and then the actual design.

A danger with an interactive system is that it may tempt the user to write and debug a program before it is thoroughly designed. An interactive system can certainly simplify program development in many respects, but no set of programming tools can substitute for good design work. The goal should be to produce a system that encourages good programming habits and discourages bad ones.

To provide a system that is easy to learn and flexible to use is also important. It should contain those features that the user really needs and no others. We will achieve this by providing a small set of well-chosen primitives and an extension mechanism to combine them. A natural way of providing these primitives is to extend Pascal by suitable predefined procedures and functions and to use Pascal for all types of user interactions. There are several advantages in using one single language within the system. The user does not have to learn one language to use for debugging and one language for communicating with the operating system. This economy of concepts for the user has, of course, had an imapct on the number of concepts (i.e. the size) within the system itself. There is only one scanner, one parser and one interpreter within the system.

Another advantage of using one language is that the user can very easily define his/her own editing and debugging procedures by using the procedure concept in Pascal. It may be convenient to have a shorthand notation for editor commands which, however, internally will be expanded to Pascal notation, and also explained by this expansion.

In an interactive environment a fast response is a necessity. To avoid recompiling the whole program after a change, one may use incremental compiling. The ultimate goal is to recompile only as small a part of the program as possible *and* only redo that part of the compiling process that is necessary. We refer to these two possibilities of incremental compiling as *horizontal* and *vertical incremental compiling*, respectively. If, for example, a programmer changes the type of a variable from integer to real, the system should only redo the type checking and not the syntax analysis (vertical

incremental compiling) and only do this for the procedure to which the variable is local, not the whole program (horizontal incremental compiling). To add or delete a statement should only involve compilation of that statement.

It is natural to let the system help the user with trivial tasks, but it may also be good to let, or even require, the programmer to give redundant information about a program and have the system check all available information for consistency. Most declarations in a Pascal program are used in this way. For the meaning of the program they are almost always redundant. One may also, for example, associate assertions and invariants with statements and variables and check them when executing the program. In this way errors are found earlier during debugging and, more importantly, the user is forced to think more about his or her program during the design.

A final goal is to produce a portable system that is well structured, easy to understand and modify, and adaptable to a mini-computer environment. In order to have a portable system, it was written in a subset of Pascal. This is described later in the sections on Implementation and Experiences with Pascal. We have decided in the initial implementation to select simple solutions to problems even at the cost of an inefficient system. Optimization should be a second phase of the project.

3.2. Design–implementation–evaluation cycle

Most systems are designed, implemented and evaluated just once. Very often, an erroneous design is detected too late, and a change is out of the question. The experience gained by implementing the system cannot be used in the design of the system. It is more economical in the long run to run through the cycle of design, implementation and evaluation several times so as to allow experience from earlier cycles to be used in later ones.[8]

In courses on compiling techniques a similar, but less ambitious, system has been constructed. Experience from that project has been very valuable in the design of the current project.

In the original student project two methods for syntax check were to be used for pedagogical reasons, namely recursive descent and a precedence method. Later, when a program editor was to be written, it turned out to be very difficult to implement commands such as *move*, *change* and *delete*. It was easy to change the program, which was represented as a tree structure (below called the program tree), but to change the stack for the bottom-up method or, even worse, the dynamic chain in the recursion of recursive descent was harder. The immediate lesson was that ordinary syntax check methods were difficult to combine with a flexible editor in an interactive system and with an incremental compiler. A syntax analysis method that can work on smaller pieces of a program and that can be split in several phases is needed. Such a method may still be based on a conventional method, however.

A more general lesson learned was the danger of duplicating information in several places. It was therefore decided that, as a general principle, every piece of information should be represented in only one place in the system. Furthermore, this information should be stored in such a way as to simplify changes of the program tree. As an example, no symbol tables are kept in the system as in an ordinary compiler. That information is, instead, contained in the declaration lists, thereby significantly simplifying, for example, deletions in the tree.

Furthermore, we found that the representation of the tree had to be carefully

designed in order to allow simple and general procedures for program-tree manipulation.

3.3. Top-down data structure design

Design of data structures as well as control structures should first be performed on an abstract level. The lack of understanding of the abstract properties of data structures as well as a lack of a good notation for abstract data structures has made this type of program design less developed than top-down design of control structures.

3.3.1. *The abstract syntax*

In the system, the information to be represented was a program. To find its abstract properties it was natural to start with its abstract syntax or 'deep structure'. By the abstract syntax we mean a structure that is convenient to use for describing the meaning of a program. For example, a while statement in Pascal consists of a Boolean expression and a statement. A repeat statement consists of a Boolean expression and a list of statements. The reserved words and delimiters used are not present in the abstract syntax.

To be able to talk about the abstract syntax, one needs a representation of it. Starting from the BNF representation of Pascal, we tried to find patterns corresponding to the properties of the abstract syntax. A recursive Cartesian product combined with a disjoint union and the sequence (list) concepts proved to be appropriate for describing programs. A notation for these concepts was defined, and the whole syntax for Pascal was rewritten using this notation.

3.3.2. *The internal representation*

The next step was to specify an internal representation for a program, starting with the abstract syntax. There are several requirements on the representation of a program. The very fact that we start with the abstract syntax makes some of them quite natural, whereas others are more artificial and force us to make compromises. Our requirements were

(i) a clean and uniform structure
(ii) no redundant information in the tree
(iii) easy traversal in textual order
(iv) easy manipulation by all program manipulators
(v) one distinguishable object for each possible cursor position.

In the representation chosen, Cartesian products and unions are represented by records with variants, and sequences are represented by singly-linked, circular lists always containing a list head. With one exception, we have a strict, hierarchical structure *never* allowing two pointers to the same object. The structure, therefore, is tree-like, and the internal representation will henceforth be called *the tree*. The exception mentioned above is that records (representing Cartesian products) and list heads contain a father pointer in order to allow easy traversal through the tree. Note, however, these pointers refer to very close neighbours and, therefore, do not disturb the overall structure.

To allow easy manipulation of a program, *all* information about it is stored in the tree, including the symbol table and the values at run time, i.e. the run-time stack. In many compilers, identifiers are represented by pointers to their symbol table entries. In our case that would destroy the tree structure, and the deletion of a declaration

would require a great deal of searching and updating. Instead, each identifier is assigned a unique number, which is used to denote the identifier. This implies a search for its declaration node at run time every time it is used. This decision makes some operations very inefficient, but we strongly believe that optimization should be performed when a running version of the system exists and not in an initial design.

Expressions and simple statements are stored as text strings in terminal nodes in the tree during the first compilation phase, the coarse syntax check. This was done for the following reasons.

(a) In designing a program top-down, one may want to use pseudonotation instead of syntactically correct expressions and statements. By storing them in the leaves in textual form, we can postpone the syntax check.

(b) It allows a split of the syntax check into many phases, thereby making possible a high degree of horizontal incremental compiling.

(c) If a partially correct program is read from a file rather than a terminal, it is advantageous to be able to first build a tree and then correct it with the system rather than to use some other editor. This is possible to some extent in this system, as partially checked programs can be accepted by the system.

This text is expanded into trees in the later compilation phase.

3.3.3. *The external representation*

To be able to inspect the internal representation of a program, one needs a printable, *external* representation of it. An extension of the representation of the abstract syntax was specified for this purpose. It contains *all* information about the internal form. Of the representation-transforming functions *read* and *print* we require that for any program P in internal form, $read(print(P)) = P$ is always satisfied.

The external form has been extremely valuable in all debugging of the system. It has also been used for storing programs on files and transmitting them between processes. This is not possible with the internal form as it contains pointers.

3.3.4. *System structure*

In an experimental system it is of utmost importance to keep open all possible ways of changing it. One excellent way of achieving this is through the use of the module or class concept. Unfortunately, this concept is not supported by Pascal, but was still used in the design and organization of data and procedures. The main module in the system is the tree module containing a *cursor* to the tree and a set of procedures and functions manipulating the tree and the cursors.

As the tree module is the kernal of the whole system, almost all procedures use it. To avoid changing all procedures when changing the tree data structure, we organized them in layers. Procedures at one level are allowed to use only those on the nearest layer below. The lowest layer contains the only procedures working directly on the tree, for example, step cursor forward, backward and to father, insert empty element in list, replace node, delete node, push/pop current cursor, and a set of functions fetching information out of the tree *without* changing it.

By this design we try to minimize the work when decisions to change the representations are made. As the design is derived from the abstract syntax, we hope that the set of primitive procedures chosen is natural and, in some sense, minimal.

4. A VIEW OF A PROGRAM DEVELOPMENT SYSTEM

To be able to make a good system design, one must have a good understanding of what is required of the system, what different facilities and subsystems should be included, what they are to do and how they are to interact with each other. The purpose of an experimental system is to give answers to some of these questions, and it is therefore impossible to make a complete design in advance. However, we have tried to develop a general model and derive system properties from it.

By the *program development process* we mean the activity of transforming a given problem into a program that can be executed on a machine and solving the problem. A *program development system* is a system that makes this process as safe and simple as possible and helps to produce a well-structured, high-quality end product.

To specify what an algorithm shall do and have it mechanically generated is today possible only for small programs, even with user interaction. What a system can do, however, is to give the user languages and tools that simplify the task of expressing his/her thoughts and have them checked in different ways, such as consistency checking of redundant information.

The system can also help the user with trivial tasks such as expansion of shorthand notation, correction of trivial errors, editing and, in general, offer a flexible and easy-to-use environment. Further, the system can provide facilities that make it possible to create an environment that suits a particular user.

A program development system consists of a set of program manipulators. We distinguish the following classes of manipulators:

(i) *Program translators* are manipulators whose source and target representations are significantly different.
(ii) *Program checkers* use redundancy to check the program, or part of it, for consistency and make only minor changes of its representation.
(iii) *Program generators* use redundancy to expand (incomplete) programs into more complete versions.
(iv) *Program optimizers* transform program representation into more efficient versions.

As the actions of checkers and generators are often connected, we will include generators in the class of checkers in the following discussion.

When developing a program the user enters program parts or information about the program and then, either automatically or on command, this information is checked for correctness and/or consistency. There is a whole set of checkers with many properties in common and also with very similar response actions. We distinguish five different types of checker responses: reject, ask for more information, correct, accept and expand program, i.e. generate additional program parts.

There is a logical ordering of checkers, i.e. one checker can do its work only if other checkers have completed their tasks. This ordering is essentially a total ordering. If we include program generators, the ordering is only partial, as program expansion is just a helpful tool, not a necessary checking action. The ordering is shown in Figure 1. We include some possible actions not actually in the system.

All checkers may give several response types and many of them even all five types. Also note that the user is included in the process. The main goal of program development is not to produce useful program results but useful programs.

In our system, these checkers are all *separate* programs working on the tree, thereby

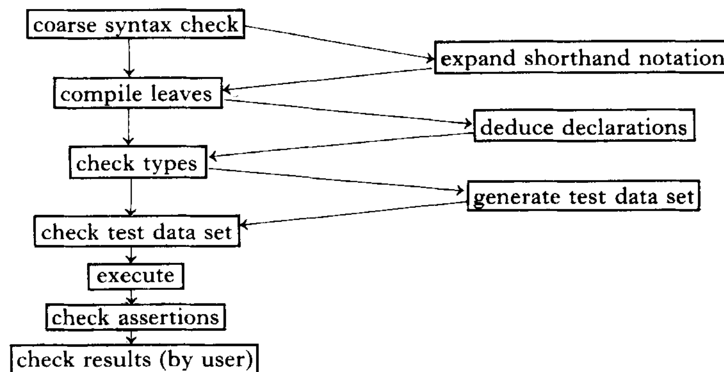

Figure 1. Ordering of checkers

making possible a high degree of horizontal incremental compiling. The smallest unit they can work on is a single statement, expression or declaration, thereby also making possible a high degree of vertical incremental compiling at the same time.

There are several ways of ordering the invocation of the checkers. The two extreme cases are

(i) vertical (many-pass) scanning, where all checkers work through the whole program, one at a time

(ii) horizontal (one-pass) scanning, where all checkers are applied to each program unit inserted by the user (cf the append-and-execute mode above).

Which scheme to use can, to some extent, be chosen directly by the user through commands. Advanced users can get full control by programming their own schemes.

As the checkers are totally ordered, it is easy to describe how much checking has been performed on each node in the tree. There is information associated with each node in the tree showing how much checking has been done, and it is 'incremented' by the checkers. Internal nodes in the tree are marked with a degree of check that is the minimum of all its subtrees. Editing procedures, such as insert, delete and change, may 'decrease' the degree of check for a certain node or even for a larger part of the program. If the user has requested more checking at that time, the appropriate checkers are automatically invoked to perform the demanded checking. It can be described as a yo-yo process. Besides checking, the checkers also transform the program tree into a form suitable for the successor checkers. To get full flexibility one needs the following properties:

(a) The pretty-print routine must give the same result, regardless of the degree of check.

(b) The routine printing the external representation must give all details of the tree, regardless of the degree of check.

The editing routines may 'decrease' the degree of check for program parts of various sizes. To reduce the required work, we want the following property to hold.

(c) The only action that shall be required for 'unchecking' shall be a decrease of its degree of check.

Thus no unparsing shall be required, implying that all checkers must accept nodes in the abstract syntax tree as long as they have a sufficient degree of check. Excessive check shall have no influence. In some cases this forces us to save some extra information in the nodes, but in most cases this is not necessary.

5. THE EDITOR

The editor is the part of a programming system that lets the user create and change the program (and possibly data). This can, of course, be done with an ordinary interactive text editor, but there are many advantages in using an editor which has some knowledge about the structure of the program. For instance, it is easy to change all identifiers named *d* without changing all other occurrences of the letter *d*. It also makes certain ungrammatical changes impossible. It is, for instance, impossible to delete a **then** without deleting its matching **if**. Furthermore, such an editor lets the user view the program as declarations, statements, reserved words, etc. rather than a text of lines. We think this helps the user to view the program in a useful manner and, therefore, encourages good programming practice.

A program editor should also be programmable, i.e. not only simple commands that are executed one at a time should exist, but also compound commands which are built up from simpler commands, for instance, repetitive commands, conditional commands and user-defined commands. As mentioned above, Pascal is used for this purpose, and the user sees the editor as a set of predefined Pascal procedures and functions, extensible by defining Pascal procedures for new commands.

The editor uses a cursor which can be moved within the program. The cursor can, for example, be positioned on a simple statement, the Boolean expression in an if-statement, an identifier in a declaration, or on a reserved word. Such a unit is called an *item*. The cursor can be repositioned by using one of the following tree-oriented commands:

(i) parent	move the cursor to the parent node
(ii) oldest child	move the cursor to the 'left-most' child
(iii) older sibling	move the cursor to the sibling to the left
(iv) younger sibling	move the cursor the sibling to the right.

These are the basic operations moving the cursor. They are used to implement all operations needed for moving the cursor.

There is also a search command that can be used to search for an identifier or a reserved word. The search command can be combined with the positioning commands to obtain search strategies.

6. THE COMPILER

The compiler consists of four parts, the lexical scanner, the coarse-syntax checker, the expression checker and the type checker. The first two work in parallel and perform both controlling and translating actions when entering a Pascal text or when reading a file.

In an interactive system, the possibility of representing a partial user program is a necessity. But partially built data structures may easily lead to both complicated and erroneous programs. It was therefore decided that the tree should always represent a complete program, possibly only partially checked. The way this is done will be exemplified by the **repeat** statement. On entering the word repeat on the terminal, the system creates

(a) a **repeat** node
(b) an until node marked 'not written'

(c) a statement list with list head and one statement, the empty statement marked 'not written'

(d) an empty expression marked 'not written'.

These nodes are used to build a (sub)tree, and the system checks whether this tree can be inserted at the current cursor position and still yield a correct program. When the programmer later enters a 'real' statement after the word **repeat**, the empty statement is replaced, and when **until** is entered, the until node is marked written. The empty objects were introduced into the tree to keep it free from nil pointers. This gives a clean structure, which is simple to work with and which gives an easily debugged system.

This way of handling the tree has as a secondary effect that it is simple to provide the user with a 'button' that, when pushed, automatically may print the next appropriate reserved word, for example, a **then** after a Boolean expression in an **if** statement, or an **end** or **until** after a statement list. The system can also complete (i.e. mark written) the whole program to the final **end** by repeating this process.

If a declaration occurs in the middle of a statement list, the compiler pushes the current cursor, changes environment, inserts the declaration and finally pops the cursor to return to the statement list.

As far as the coarse syntax checker is concerned, only some of the reserved words are of interest. Expressions and simple statements are not checked at all. They are treated as indivisible text objects. The expression checker checks the program as much as possible withouth knowledge of declarations. '$x+*y$' can never be an expression. '$x+y$' is never a Boolean expression, but is considered a legal (but not type-correct) expression even if x and y are declared Boolean. A leaf in the tree produced by the coarse syntax check contains a source text and a string of so-called *lexemes*. The expression checker consumes a string of lexemes and generates a string of tokens representing the expression in reverse-Polish notation as well as a tree representing the expression. This is done for 'historical' reasons and we would prefer having only a tree structure. The tree form is used for editing.

The type checker reads the reverse-Polish notation and generates a slightly modified version of it, for example, discriminating different + operators. It may also read already type-checked Polish notation. This is necessary as, for example, the deletion of a declaration invalidates the type check, and we want to be able to redo the type check without unparsing.

7. THE INTERPRETER

The interpreter starts executing the main block of the program. The current cursor is used as program counter. It can terminate *normally* by passing the final **end**. It terminates *abnormally* if an error, a break-point, an unsatisfied assertion or a not-written statement is encountered. In these cases, the control returns to the user, who may inspect variables, examine a flow trace, change variable values or program text and then restart execution. Further, a program may stop to wait for data to be submitted by the user. Finally, a program may, in the append-and-execute mode, stop to wait for more program text to be inserted. Execution is then resumed as soon as possible. Note, however, that if the **else** part is selected in an **if** statement, the whole **then** part must be inserted before execution can be resumed.

If the interpreter encounters a node that is not fully checked, it invokes the appropriate checker(s) before execution is continued.

8. IMPLEMENTATION

The system has mainly been implemented on a PDP-11 under UNIX,[9] although another machine initially was used, as no Pascal compiler was available under UNIX. To do without a Pascal compiler it was decided that the system itself should be used for translating Pascal programs into C programs; C is a Pascal-like language under UNIX.

The work with a translator from Pascal to C has continued, however, as it is a good example of using the system for program manipulation, and also yields a facility useful for other purposes. On two points we have to restrict Pascal in order to simplify the translation process. Nested procedures are not allowed, and field names in records have to be unique in the system.

The system soon became too big for the small address space of a PDP-11 and it was therefore split into several processes communicating via pipes. This work has been the main obstacle of the project, but also an advantage as it has forced us to partition the system into small parts with well-defined interfaces. The conclusion is, however, that the system is too big for a PDP-11.

The system has now been moved to a VAX computer with UNIX, which allows us to run the system as one process. It also allows us to use the system for the development of the system itself, a goal we had hoped to reach at an earlier stage.

9. EXPERIENCES WITH PASCAL

The most serious problem in using Pascal for writing large programs is the lack of a module concept in the language. A module is a construct which helps the programmer to distinguish between an abstract data type and its implementation. For instance, nothing can prevent a programmer from using the representation of the program tree even though we have designed a particular abstract data type to handle the tree. The only solution to this problem is to create conventions which the programmer has to follow.

Another serious problem is the deallocation of dynamically created storage. There is a standard procedure in Pascal called dispose which is intended to solve this problem. Few Pascal compilers have implemented dispose as it is described in the Pascal report. This makes it impossible to write a *portable* program which uses dispose. Most Pascal compilers use a mark-release strategy for deallocation of memory. This is not standard Pascal, and it is dangerous to use. Some compilers are using an automatic garbage collection which also is unsafe (because of the weak typing restrictions in records with variants). Our solution to this problem has been to write our own procedures for handling dynamically allocated storage (using arrays and records with variants).

Other drawbacks in the language are the insecurity of using variants, unorthogonality of types and the imprecise language definition. There are some drawbacks which are peculiar to an interactive environment. If the language allowed statement lists in **while-**, **case-** and **if**-statements, the editing of a program would be simplier. To

change a **while** loop to contain two statements instead of one is now impossible without adding an extra **begin–end** pair.

Two more problems in interactive Pascal have to do with input/output. The first is that Pascal assumes read-ahead, which is hard to use in an interactive system. The other is that input/output is not defined for arbitrary values. This is a great disadvantage in an interactive system, where you want to inspect and change values of variables. It is easy to define an input/output notation for arrays, records, scalars and sets. It is a more serious problem for pointers, where a complete redesign of the language would be necessary.

We find the choice of using Pascal as the editor language to be an important decision to keep the number of concepts small. This use of Pascal is somewhat unsatisfactory, since the language does not treat functions as first-class objects. It is, for instance, impossible to have functions returning functions, and it is also impossible to have a function as an argument to a function without giving a name to the argument function.

The system is implemented in a 'portable' subset of Pascal. By portable constructs we mean those that are implemented, and implemented in the same way in most compilers. For example, we do not use procedures as parameters or **set of char** even though they would have been useful. The system, however, will accept full Pascal. The system has been ported twice, both times with little effort. The required changes have been of lexical nature and have been performed with a text editor in a couple of hours.

A module structure was forced on Pascal, mainly with the aid of the file system and a copy facility. We also decided to follow a suggestion by Ledgard[10] regarding a Pascal program standard. The standard was extended by some further rules, for example, restricted use of global variables, no backward jumps, assignment to a funtion identifier must always be the last thing executed in a function, and some conventions concerning comments.

10. FUTURE DEVELOPMENTS

As mentioned earlier, the most serious drawback of using Pascal in large programs is the lack of a module facility. We have designed a module concept for Pascal which is similar to the class concept of SIMULA (except that variables cannot be used outside a module). We want to make suitable additions to our system so that it handles Pascal with modules. In addition, we plan to write a program which translates Pascal with modules into standard Pascal.

We also plan to make some extensions to our system such that it is more convenient for different kinds of users. The system should be able to distinguish between an advanced user and a beginner in that a beginner gets a more verbose conservation. The user should tell the system what kind of terminal is to be used. If a hard copy terminal is used, the system should insert a number of blank characters (corresponding to the expected indentation depth) before each line. A user with a screen-oriented terminal should get a pretty-printed version of an input line as a response. This will be printed on the same line as the input line, so that the user always has a pretty-printed version of the program on the screen.

An interesting idea, which has been used in the Mentor system,[6] is to associate attributes with each subtree in the program tree. If this is done in a flexible way such

that the number of attributes is not fixed, it is a very valuable tool, which can be used for

(a) documentation (by associating comments with various parts of the program)
(b) debugging (by inserting break, trace and statistics-collecting procedures)
(c) proving (by inserting assertions, pre- and post-conditions).

With the small address space of contemporary mini-computers, it is an advantage to have some kind of virtual memory in a large programming system. Almost all information that our system uses is contained in the internal representation of the user's program. The system should therefore be able to store subtrees of the program representation on backup memory. The innermost tree module should take care of swapping between different memories and the virtual memory could be absolutely invisible to other modules.

11. COMPARISONS WITH OTHER STRUCTURE ORIENTED EDITORS

The Lisp community has a long experience in using structure-oriented editors.[8]

The first structure-oriented editor for Pascal which we know of is the Mentor editor, developed by Donzeau-Gouge, Huet, Kahn and Lang at IRIA.[6] They have a special language for manipulating abstract syntax trees. Type-checking is not done while program text is inserted and there is no interpreter for Pascal.

Medina-Mora and Feiler have described a structure-oriented editor for a subset of C within the Gandalf project at Carnegie-Mellon.[11] There is no interpreter for the language, instead they want to generate code from the abstract syntax trees. It is not clear how much of the system is implemented.

Shapiro *et al.*[12] have described their plans on a programming environment for Pascal with an interpreter and a structure oriented editor. Their plans seems to be quite close to the system described here.

12. CONCLUSIONS

One way of having the computer take part in the creation of programs is to use interactive programming environments. Such environments let the user create programs and the system immediately checks the programs for errors. They also help the programmer with bookkeeping of programs. Most of such systems are based on Lisp (because of the ease of representating Lisp programs in Lisp). Some ideas present in these systems exist in a premature form in Basic systems. The system presented in this paper is an interactive programming environment for Pascal. It differs from most other systems in that the user can control the amount of checking to be performed. The highest level of check is a complete syntactical check, including type checking. This checking can occur immediately when the user enters a program.

ACKNOWLEDGEMENTS

We would like to thank Bror Bjerner and Kent Pettersson for participation in the design of the system and also thank them, as well as Barbro Atlestam, for programming the editor, the parser and the type-checker of the system. Further, we thank Per Bergsten and Kent Karlsson for programming the interpreter, Sven Wiberg for

programming the translator to the C language, Dan Magnerot for work with the type checker and Thomas Johnsson for the design of the virtual memory.

This work has been supported by The Swedish Board for Technical Development (77-3580, 78-3719, 79-3692).

REFERENCES

1. E. W. Dijkstra, *A Discipline of Programming*, Prentice Hall, Englewood Cliffs, 1976.
2. T. E. Kurts, 'BASIC', History of programming language conference, *SIGPLAN Notices*, **13** (8), (1978).
3. N. Wirth, *The Programming Language Pascal* (Revised report), Eidg. Technische Hochschule, Zürich, 1972.
4. W. Teitelman, *Interlisp Reference Manual*, Xerox Palo Alto Research Center, Palo Alto, Calif., 1978.
5. R. P. van de Riet, 'BASIS—an interactive system for the introductory courses in informatics', *IFIP Congress*, Toronto, 1977.
6. V. Donzeau-Gouge *et al.*, 'A structure oriented editor: a first step towards computer assisted programming', *IRIA report no. 114*, Paris, April 1975.
7. T. Teitelbaum, 'The Cornell program synthesizer: a tutorial introduction', *TR 79-381*, Department of Computer Science, Cornell University, 1979.
8. E. Sandewall, 'Programming in the interactive environment: the Lisp experience', *Computing Surveys*, **10** (1), (1978).
9. K. Thompson and D. M. Ritchie, 'The Unix timesharing system', *CACM*, **17** (7), (1974).
10. H. Ledgard *et al.* 'A basis for executing Pascal programmers', *SIGPLAN Notices*, **12** (7), (1977).
11. R. Medina-Mora and P. H. Feiler, 'An incremental programming environement', *IEEE Trans. Software Engineering*, (5), (1981).
12. E. Shapiro *et al.* 'PASES: a programming environment for Pascal', *SIGPLAN*, **16** (8) (1981).