UCSD WORKSHOP
on
SYSTEMS PROGRAMMING
EXTENSIONS
to the
PASCAL LANGUAGE

Institute for Information Systems
University of California, San Diego

10-14 July, 25-28 July 1978

PROCEEDINGS OF

UCSD WORKSHOP ON SYSTEM PROGRAMMING EXTENSIONS TO THE PASCAL LANGUAGE.

CONVENER KENNETH L. BOWLES

EDITORS TERRENCE C. MILLER
GILLIAN M. ACKLAND

INSTITUTE FOR INFORMATION SYSTEMS

UNIVERSITY OF CALIFORNIA, SAN DIEGO

10-14 JULY, 25-28 JULY 1978

EDITORS FOREWORD

This document is an attempt to record the results of the UCSD Workshop in Systems Programming Extensions to the Pascal Language. Readers should be aware of the following limitations:

1) This document does not reflect changes of opinion that have occurred since the conference took place.

2) This document is comprised of reports prepared by subgroups of the conference. In some cases, the wording of the report was agreed to by all members of the subgroup. However, in other cases, the reports were prepared by a single participant, and were not reviewed by the subgroup.

This is an historical record and should not be used as an authority.

The Proceedings have been distributed to all Workshop participants and will be available at cost to anyone who requests individual copies (supplies permitting). Recipients of this document are requested not to distribute it further.

Terrence C. Miller
Gillian M. Ackland 5/8/79

TABLE OF CONTENTS

SECTION A

ORIGINS OF THE WORKSHOP

By Fall of 1977, a growing number of firms in the computer industry were using Pascal for system programming. Some of these found it necessary to extend Pascal as defined in the User Manual and Report, Jensen and Wirth. Although some of their extensions accomplished similar tasks, they varied widely in choice of syntax and semantic definition. In subsequent months, Ken Bowles of the Institute for Information Systems (IIS) of UCSD contacted certain industry representatives as to their willingness to cooperate in a discussion of possible extensions and to attempt to reach an agreement on syntax and semantics. Most of the contacted industries responded positively, provided agreement was reached soon because of the expense of retrofitting changes in existing software.

On the basis of that response, Ken Bowles convened a 9-day workshop, July 10-14, 25-28 in 1978 at the University of California, San Diego. Participants included representatives of over 30 companies (see list at end of Proceedings), official representatives of the Pascal User's Group, and individuals involved in international efforts on standardization of and extensions to Pascal. In addition, IIS provided about ten people who served in a number of ways to facilitate the Workshop functions.

ORGANIZATION OF THE WORKSHOP

A checklist of topics for potential consideration was distributed to Workshop participants prior to July. The list included virtually all serious suggestion of problems that might require extension and changes to Pascal (as Pascal was understood in the absence of an official standard). Workshop participants were asked to mark their company's priorities on each of those items. The responses to the checklist did not support the idea that the sets of problems deemed important were nearly as homogeneous as previously hoped. The overlap was small.

The checklist was categorized into the following topic groups: control constructs; issues related to parameters; expressions; data types and declarations; input/output; modularity and separate compilation; real-time and concurrency; programmer convenience. To ensure that each of the items in these topic groups was given fair consideration, the Workshop participants divided into eight subgroups for detailed discussion of the eight topic areas. Although format varied, each day was a combination of subgroup meetings and plenary sessions. The plenary sessions covered itens appropriate to the combined group, such as resolution, scheduling, and the subgroup discussion reports.

In the last two plenary session, the Workshop participants voted on each item in the original checklist and decided whether the topic was approved, dismissed, too experimental for any

decision, or recommended for further consideration by one of two groups described below.

ORGANIZATION OF THE PROCEEDINGS

Section B describes general resolution made in plenary session.

Section C contains approved topics. The reader should take care to note that some of these topics were approved "in theory" but no agreement was reached in syntax or semantics. Section C is in three parts; the latter two are intended as packages and each should be implemented as an entire package or not at all. All of these are understood to be extension to the language and should be noted as such in documentation as per the general resolutions' instructions.

Section D contains topics that were requested as extensions but the Workshop participants agreed (after mutual education) that they were unnecessary as extensions because they were already available in the language. Since their request as extensions shows that this decision will surprise some users of Pascal, this section of tutorial notes was included for their education and benefit.

Section E contains topics that were dismissed for other reasons. The reasons varied considerably and the net result was that there was no consensus on any of these topics. This section contains the bulk of the items from the original checklist.

Section F includes topics from C and G (approved and experimental) which the Workshop participants supported but agreed to defer to existing standardization and extension groups for clarification, syntax, and semantics. The standardization group is under the British Standards Institution and is headed by Tony Addyman. This group is preparing a proposal of a Pascal standard for the International Standards Organization (ISO). The extensions group is international in membership and, at the time of the Workshop, was led by Jorgen Steensgard-Madsen. This group is usually referred to in the Proceedings as the Working Group.

Section G contains suggestions for experiments in the areas in which extensions might be desirable. Encapsulation, external compilation and concurrency are the major topics in this section.

A WORD ON THE NUMBERING SYSTEM

Topics have maintained their numbers from the original checklist. Hence, section C has 1.1 followed by 2.2. The gap between those numbers occurs because 1.2-1.5, 1.7-1.12, 1.14 and 2.1 are in section E; 1.6 is in section D; and 1.13 is in section F.

EFFECTS OF THE WORKSHOP

1. The key people in the computer industry with respect to Pascal were brought together.

2. The BSI effort was accelerated.

3. The Workshop participants were able to serve in an advisory role, albeit weak, to inform the Working Group and the BSI group concerning interests and preferences.

4. Many participants were educated regarding how robust the Pascal language already is and how difficult it is to do language design well even after agreement is reached on a particular goal.

5. To our knowledge, this Workshop was the largest non-proprietary Pascal extension effort to date. For the record, of the 95 topics on the original checklist only 17 made it to section C (approved) and the majority of these are approved in principle only, but no agreement was reached for syntax or semantics. Because of this experience, it is expected that most of those who participated will approach future extension proposals with caution and an improved understanding of Pascal's

SECTION B GENERAL RESOLUTIONS

The Workshop attempted to reach unanimity on a number of topics covering its overall impact in the Pascal community, and also on specific topics of language detail. The general recommendations are contained in this section, while the detailed recommendations appear in several of the subsequent sections. The form of the general recommendations was agreed upon at the end of the first week of the Workshop. The wording was revised during the second week.

At the time the workshop convened, two major activities with respect to the definition of the language Pascal were already underway. In light of the shortcomings of the Jensen and Wirth User Manual and Report, a small group had begun working on a complete definition of the Pascal language. This definiton is intended for submission to the International Standards Organization for consideration and possible adoption. A Working Group focused around Professor Jorgen Steensgaard-Madsen had begun working on extensions to the Pascal language aimed at correcting a few well known deficiencies in the language. In light of these activities the Workshop assumed as its primary goal, to address well-defined, consistent, application-oriented extension sets and agreed to pass to the other two bodies such recommendations and information deemed appropriate to their work.

The Workshop recognized the existence of possible modifications to the Pascal language which, due to the impact throughout the language, would defacto create a new language and decided not to act on these modifications at this time.

In order to achieve the purposes stated above the Workshop has resolved to:

I. Publish and distribute the Proceedings of the Workshop. In particular, the Proceedings will be forwarded to the committee preparing a draft ISO standard for Pascal, the Pascal Users Group, and to the Extensions Working Group. Produce a document that provides syntax and semantics for certain dismissed topics. This document will be distributed to participants of the Workshop and others who request it, but it is not to be considered part of the Proceedings of the Workshop.

II. Organize a structure which will permit the orderly continuation of the work begun at the meeting in San Diego.

III. Provide a mechanism to reinforce the importance of standard Pascal by agreeing that all compilers purporting to support the programming language Pascal should include a variant of the following statement in the source code and all documentation:

"The language --(1)--supported by this compiler contains the language Pascal, as defined in --(2)--, as a subset with the following exceptions:

(a) features not implemented

--(3)--	-- refer to page --
-------	--
-------	--

(b) features implemented which deviate from the standard format

------	--
------	--
------	--

Notes:

(1) insert the name of the dialect

(2) insert a reference to a specific definition, such as

"the Jensen and Wirth User Manual and Report(specify edition)" or
"the ISO draft standard" as appropriate.

(3) a brief statement plus reference to more detailed information will suffice. The list should be as complete as possible.

SECTION C WORKSHOP APPROVED EXTENSIONS

This section contains recommendations to the Pascal community regarding conventialized extensions. (The term "conventialized extension" has come into widespread use in the Pascal user community. The Workshop understands it to mean that if one chooses to extend the language for the purposes associated with a conventialized extension, it is recommended that the published syntax and semantics associated with that extension should be used rather than some alternative.) These recommendations were approved by general consensus of the Workshop attendees. In other words, a substantial majority of those attending agreed to the recommendations as stated. Where dissent occurred, as happened on many topics, it represented only a small minority of those attending.

An attempt was made to partition the approved extensions into groups of related items. The hope was, that implementors would include all items within a group, rather than implementing only a partial selection of the recommended extensions. This concept failed to receive enthusiastic support except as regards extensions for numerical applications (section C.2).

The grouping on Files (section C.3) represents an attempt to group related items for clarity, rather than a recommendation that every implementor install all items noted in C.3, if any are installed at all. Section C.1 contains recommendations of general interest which should be considered individually.

SECTION C.1 GENERAL

1.1 Add else clause to case statement.

Extend the case statement to include specification of the action to be taken if the selector expression fails to match any case constant (label) in the statement.

Recommendation:

The Workshop recommends the implementation of this extension (as published in Pascal News #13) according to the syntax and semantics adopted by the Working Group.

Discussion:

The Working Group is addressing this problem and, according to Arthur Sale, there is substantial consensus on the desirablity of this extension and on the method of implementing it. According to members of the Working Group present at the Workshop the syntax is

```
case-statement =

case expression of
case-list-element { ";" case-list-element }
[ "otherwise" statement { ";" statement }]
end
```

*****

2.2 Provide type-checking in parameterized procedures and functions.

Recommendation:

This change is desirable to cover a lapse in type checking. A consensus has not been reached on the method to use. See 2.2 in Section F.

*****

2.3 Add adjustable array parameters.

Add dynamic array (ie. adjustable array bounds) parameters to allow libraries of procedures that can operate on different-sized arrays.

Recommendation:

This change represents a desirable change to the language. The Workshop prefers the form mentioned below, but defers to the Working Group.

Discussion:

We felt the following form, which is similar to that suggested informally by Wirth, was desirable and is being examined by the Working Group.

```
procedure zap( var a: array [ i..j: type ] of char);
```

where type is integer or other scalar (not a subrange).

[Editorial comment: The phrase "not a subrange" was controversial.]

The variables i and j would receive, respectively, the lower and upper bounds of the array being passed. Within the procedure any modification to the variables containing the array bounds should be disallowed. Note that the limitation to var type adjustable bounded arrays may be desirable, because to allow passing value type arrays would be very close to allowing

dynamic arrays, since the space needed could not be calculated until run-time. It was reported by members of the Working Group present at the Workshop that they may not impose this restriction in order to permit use of string constants.

*****

2.7 Allow functions to return any type.

Extend functions to return any type, rather than just scalars and pointers.

Recommendation:

This is a desirable extension to the language, and we recommend it be adopted.

[Editorial comment: Successive votes on this issue oscillated violently. The implications of this recommendation regarding functions returning files were not discussed.]

Discussion:

The only reservation is that this might complicate space management, especially for existing implementations.

*****

4.2 Minimum sizes of data types.

Decide upon the minimum set size, the minimum range of the type integer, the minimum range and precision of type real, and the ability to specify the desired range and precision of real numbers. This would allow small machines to use small, easily manipulated representations for most values, and only incur more expensive representations when extra precision is needed.

Recommendation and discussion:

It was decided not to specify minimum sizes for any types; it might be perfectly reasonable to have only 8-bit integers in some applications. For portability, sets should support set of char. A large number of portability problems related to insufficient maximum sizes are due to small sets. See section E for discussion of the remainder of these topics.

It was also decided not to standardize extended precision forms at this time. The details of the standard would rely on the type (real or integer), precision, application,

and on machine factors. More user experience with extended precision is needed before agreement can be reached on a standard.

Jensen and Wirth implies that the result of an operation on integers may be undefined if it or it's arguments absolute value exceeds MAXINT (which is implementation defined). The question, is what should a compiler do with a declaration like "K: 0..N" where N is bigger than MAXINT. Many implementations will not produce good code where K is used. The compiler should be required to flag an error when it cannot correctly compile something.

*****

4.5 Structured constants.

Add the ability to declare structured constants, for tables and other structured data that currently must be variables and have code space and execution time associated with their definition.

Recommendation:

The Workshop decided that structured constants should be accepted but defers to the Working Group for a final recommendation on the syntax. We recommend the example shown below.

Discussion:

This feature was accepted primarily because it can be implemented easily in such a way as to provide a significant decrease in the size and execution time of code required for many applications of Workshop participants with severe size and speed constraints. It is also clean and consistent with existing syntax.

```
type arraytype = array [0..1] of record
                        i : integer;
                        r :real
                    end; {record}

const arconst = arraytype( (1,2.0), (0,0.0) );
```

A const section is needed after the type section (See 4.5b in section F for discussion of methods)and in this section declarations of the form given above are allowed. The list of values is recursive i.e. ((1,0),(0,1)) can be used to specify the value of a constant two dimensional array of integers or a record of records of two integers each. Note that in a record with a case variant, a record variant selector must

appear in the list, even if a tag is not explicitly a part of the record. The list may not contain variables or expressions. The production for <expression> needs to be extended to include structured constants, along with indexing and selection of structured constants.

*****

4.18 Representation of non-decimal numbers.

Recommendation:

Realizing that this is a highly implementation dependent subject, we recommend that it remain so. In the absence of an existing convention, the Workshop recommends the following syntax:

<non-decimal number> ::= <radix part> # <constant part>

<radix part> ::= 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
13 | 14 | 15 | 16

[Editorial comment: Some participants felt that <radix part> was limited to 2,8,and 16.]

<constant part> ::= <hex digit> {<hex digit>}

<hex digit> ::= <digit> | A | B | C | D | E | F

Examples: 16#FFFF

8#177777

2#1111111111111111

*****

6.2 Specify the syntax for separate compilation of Pascal modules and procedures, and rules specifying what must be recompiled when a procedure is changed.

Recommendation:

The Workshop acknowledges an existing mechanism for separate compilation that is widely used: the substitution of a linkage designation for the procedure block to indicate it's external availability. (See chapter 13.A.2 of the User Manual) If this conceptual approach to separate procedure compilations is used, then this conventional syntax should be adopted. Such a mechanism may not provide the type security that is desired

and expected by Pascal programmers. In section G an experimental solution to this problem is presented.

*****

8.4 Upper/lower case and break characters.

Resolve what to do with identifiers written in both upper and lower case; and whether a break character for numbers and identifiers should be allowed and if it is significant in identifiers.

Recommendation:

We recommend that there be no distinction between upper and lower case except inside <string>.

<string> ::= ' <character> {<character>} '

If a break character exists, we recommend that it not be significant and that it be allowed only in identifiers and only where a digit would be allowed. i.e.

<identifier> ::=<letter> {<letter or digit or break character>}

<letter or digit or break character> ::= <letter> | <digit> | <break character>

We recommend that the underscore character ('_') be used for a break character on any implementation with a character set that includes this symbol.

[Editorial comment: Note that the effects of the underscore character on printing devices vary widely.]

<break character> ::=_ | <empty>

Note: We strongly recommend that any implementation that allows lower case characters and/or the break character in identifiers indicate with a warning message any occurrence of an identifier that does not exactly match its declaration in case and occurrence of break characters.

*****

8.7 Character set enumeration.

Enable one to enumerate a character set, or provide some other device to lessen portability problems, or decide upon one character set. Should all compilers be able to compile for ASCII and EBCDIC? Should control characters be in the type char on ASCII-based systems?

Recommendation:

We follow Jensen and Wirth [User Manual].

We recommend that if the character set contains lower-case letters ('a' to 'z'), that they be in ascending order.

We felt that due to the large number of conflicting character sets we could not recommend anything stronger. If a choice of character sets exists, we recommend the use of the ISO standard character set (ASCII in the United States).

*****

3.4 Exponentiation.

Add an exponentiation operator, or a standard procedure power (or raise), for the number-crunching people.

Recommendation:

We recommend a conventional, pre-declared function,

```
function power(a,b:real):real;
```

A.H.J.Sale presents (Section D) a Pascal version of such a function, so a language extension here is not absolutely necessary.

Minority opinion:

People who are used to '**' as an exponentiation operator want this operator and futhermore, it doesn't conflict with existing syntax. If they can be given complex numbers why should they not also get their exponentiation operator. This introduces a new level of precedence in expressions but not an objectionable one since everyone seems to have the same idea as to what it's precedence should be. (Note that the 'Report' says "sqr(x) computes x**2").

Discussion:

See Section D.

*****

4.9 Complex.

Add and provide the language support of the type complex, for scientific application people.

Recommendation:

We recommend that complex arithmetic should be considered a conventionalized extension to Pascal. All details of the extension have not yet been finalized, so the Workshop's conclusions are presented in section F as suggestions for consideration by the Working Group.

*****

5.1 Define communication between program and operating system through the program heading.

Recommendation:

We recommend that the program heading be used for communication between program and operating system.

We recommend that the nature or the means of this communication not be specified in the language or as an extension.

Discussion:

Jensen and Wirth [Report] specifies only that the program heading include identifiers. We feel that the usage of the identifiers should be implementation defined, since the interaction with the operating system is, by definition, system-dependent.

One of the primary uses of the program heading is to identify Pascal file variables which will be bound to files existing outside, and beyond the lifetime of, the program. However, we do not wish to limit the identifiers to be file names because it may be desirable to communicate other data through the program header. For example, in the C language under UNIX, the system passes an argument count and array of pointers (to strings) to the program when it is invoked. The strings include file names as well as other arguments.

We recognize that it may not always be feasible to follow the convention described here. For example, this mechanism does not permit the specification of an arbitrary number of files at program invocation time (e.g., by the use of "wild card" symbols). In such a case, binding is best done at run-time. For this reason, the mechanism discussed in 5.2 is provided as an alternative specifically for the binding of file variables to system files.

*****

5.2 Decide how to open files that exist outside the program.

Recommendation:

We recommend, as an implementation-dependent extension, the use of RESET and REWRITE described below.

Discussion:

The standard procedures RESET and REWRITE should be extended to allow at least two arguments. The first is the file variable, as in the standard. The second is a string (packed array of char, with implementation-defined bounds); this presumably will name the external file to be bound. (The question of delimiting the end of the valid file name was not addressed.) For example,

```
reset(filevar,'filename')
```

binds the Pascal (internal) file filevar to the externally defined file 'filename'. Additional arguments may be included to provide further information needed to complete the binding. Their content is not specified. The suggestion that a new standard (or predeclared) procedure be added solely for the purpose of binding is rejected since there is a large user base which already uses the recommended extension.

*****

5.4 Add support for random access files.

Recommendation:

We recommend that this capability be considered as an experimental extension to the language.

Discussion:

There is a clear need for "random access" file capability, but the mechanism, as it impacts Pascal, is not yet well enough understood that we can specify how to add it to the language. Two alternative approaches are sketched (see section G); these include the semantics and implementation, but the syntax has not been settled.

*****

5.13 Should RESET be required before GET and REWRITE before PUT.

Recommendation:

We recommend this item as a clarification to the language.

Discussion:

While Jensen and Wirth does not explicitly require the use of RESET before GET and REWRITE before PUT, this clarification is useful for implementors in that it equates RESET with "open for input" and REWRITE with "open for output".

The following procedure written in standard Pascal will permit appending to an existing file:

```
procedure append(var f: file of t);
  var tempf: file of t;
  begin
    rewrite(tempf); reset(f);
    while not eof(f) do
      begin
        tempf^:=f^;
        get(f); put(tempf)
      end;
    reset(tempf); rewrite(f);
    while not eof(tempf) do
      begin
        f^:=tempf^;
        get(tempf); put(f)
      end
  end;
```

A predefined function accepting files of any type could be provided in any implementation where the operating system allows the effect of this procedure to be obtained without copying.

*****

Many proposals for changes of extension to Pascal were apparently founded upon an incomplete understanding of the 'standard' language. In this regard, the question of what the standard contains was often left clouded because of inconsistencies or incompleteness in the Jensen and Wirth Manual and User Report. In many cases Tony Addyman provided information indicating the probable course the British standardization effort will take.

This section contains suggestions on how some frequently encountered problems can be handled within the existing standard Pascal language as it is understood by the Workshop attendees.

Contents of these topics in this section do not imply workshop approval.

1.6 Disallow dependence upon the value of FOR control variables after loop. Disallow altering control variable in loop.

Recommendation:

Implementors of Pascal compilers are encouraged to try to enforce the prohibition specified in the standard against altering the control variable in FOR loops and against any dependence upon the value of this variable after the loop.

Discussion:

Detecting all attempts to alter the control variable at compile-time is difficult (eg procedure side-effects). However use of a hidden count variable permits run-time checking as shown by the following code template:

```
for v := e1 to e2 do s;

TEMPLATE:
    tempcount:=e1;
    templimit:=e2;
    loop: if (tempcount > templimit)
          then goto fini;
          v:=tempcount;
            s; {body of the loop}
          if (tempcount <> v)
          then abort_the_program;
          tempcount:=tempcount+1;
          goto loop;
     fini: v:=undefined;
```

[Editorial comment: See also C. N. Fischer amd R.J. Leblanc, SIGPLAN Notices V12 3 pp19-24 for another implementation.]

The program will abort if the control variable (v) is

changed inside the loop body.

*****

3.4 Exponentiation.

Add an exponentiation operator, or a standard procedure power (or raise), for the number-crunching people.

Discussion:

(as summarized by Arthur Sale, Dept of Inf Sci, Uni of Tasmania, July 14 1978)

3.4.1. Motivation

Many users of FORTRAN have pressed for additions to the language Pascal, in order to make it easier to use in numerical computations. One of the most frequently made requests is for an exponentiation operator, often modeled on the FORTRAN syntax and a '**' token.

3.4.2. Recommendation

This operator not be added to the language as a 'conventionalized extension'.

It was felt that:

(a) the extra precedence rules would complicate the language for little benefit,

(b) the operation is little used except in numerical computations, and the extra operator would clutter the language, and

(c) the need for this operator is not as important as sometimes indicated, as long as there is some way to express exponentiation.

It was recognized, however, that a need did exist which seemed to be predominantly for real values to be raised to an integer or real power. We considered that a single function could be provided to users, either as a function to be included in their program or externally bound into its environment or, as a pre-defined function in a compiler. The action of the program is defined by its listing, provided later; the heading is:

function power(x,y : real) : real;

3.4.3. Optimization

The form of this function enables one to economize on functions, especially pre-defined ones, but enables a compiler which implements it as a pre-defined function to still carry out some effective optimizations.

For example:

(a) If the actual argument y is a small integer constant (say -1 or 3) the compiler can insert in-line code to compute the result by an optimal multiplication (or division) process.

(b) If the actual argument y is known to have integral values,(it may be a constant or be of integer type or a subrange thereof), the square-and-halve algorithm encorporated in the provided function can either be invoked by a function call, whose name is unknown to the programmer, or by in-line code.

(c) Otherwise, it will be necessary to call on a hidden function that uses the exp-ln evaluation.

It is important that the optimization remains completely equivalent to the function program given. If there is a deviation, users, who have to include the function text in their programs, may find the effects different, and there will be an unnecessary loss of portability.

3.4.4. Program

```
function power(x,y : real) : real;

{Author : Arthur H.J. Sale, Pascal Users Group.  1978-Jul-14

    This function is provided for use in Pascal programs
instead of an exponentiation operator.  Users may include it in
their source  text, or implementors may provide a pre-defined
function with the same interface definition and equivalent
computation.

    The function is in standard Pascal, except for the
procedure 'abort_the_program' which is called when the result is
not mathematically defined.  This procedure should cause a run-
 time error.

    If the exponent is integral, the power is evaluated by
 multiplications, otherwise it is evaluated by logarithms, and
 is not defined for negative values of x.}

var
    z: real;                        (* accumulates the result *)
    e: integer;                     (* carries integer exponent *)
    integral: boolean;              (* true if y has integral value *)
    mode: (direct,reciprocal);      (* evaluation mode for integral y *)
```

```
begin
    if (x = 0.0) then begin
        if (y = 0.0) then begin
            (* result is undefined : 0 ** 0 *)
            abort_the_program;
        end else begin
            (* 0 to some other power *)
            power:=0.0;
        end;
    end else begin
        (* determine which algorithm to use *)
        if (abs(y) > maxint) then begin
            integral:=false;
        end else begin
            integral:=(trunc(y) = y);
        end;
        (* so select on it *)
        if integral then begin
            e:=trunc(y);
            (* preserve in mode whether a reciprocal is needed *)
            if (e >= 0) then begin
                mode:=direct;
            end else begin
                mode:=reciprocal; e:=abs(e);
            end;
            (* establish invariant R trivially *)
            (* at this point let e0 be value of e, and similarly
              let x0 be the value of x.  Variable e is already
              set. *)
            z:=1.0;
            (* The invariant R for the loop is:
              R = "(z * (x ** e)) = (x0 ** e0), and (e >= 0)" *)
            while (e <> 0) do begin
                (* R holds here of course *)
                while not odd(e) do begin
                    x:=x*x; e:=e div 2;
                end;
                (* R still holds, but also we know e is now odd *)
                z:=z*x; e:=e-1;
                (* R still holds, and e is even again *)
            end; (* of while *)
            (* R holds, together with (e = 0), imples z = x0 ** e0 *)
            if (mode = direct) then begin
                power:=z;
            end else begin
                power:=1.0/z;
            end;
        end else begin (* of non-integral y treatment *)
            if (x < 0.0) then begin
                (* result is of complex type!! *)
                abort_the_program;
```

```
            end else begin (* the safe case *)
                power:=exp(ln(x)*y);
            end;
        end;
    end;
end; (* of function power *)
```

*****

3.12 Allow programmers the ability to "read" from or "write" to a string.

Recommendation:

We conclude that this item can be done within the language. See also 4.6 in section G and 5.9 and 5.12 in this section.

Discussion:

The aim of this facility is to be able to do formatted transfers to and from strings at run-time (e.g. integer to string, string to integer). It was felt that this capability is already present, using the file mechanism:

To translate the "integer" in string form in S to a Pascal integer I:

```
var
  f: text;
  S: packed array [0..x] of char;
  I: integer;

begin
  ...
  rewrite(f);    {prepare f to receive string}
  write(f,S);    {write the string to the file}
  reset(f);      {prepare f to return an integer}
  read(f,I);     {read the string as an integer}
  ...
end;
```

Note that, if in-memory files (5.12) are not used, this mechanism may invoke some I/O (even though f is not bound to an external file, it may be implemented as a temporary file of some sort).

*****

4.2 Note regarding minimum sizes of data types (and 4.13 Unsigned integers).

See also Sections C and F.

Jensen and Wirth (chapter 2, section B) describes an implementation-defined standard identifier maxint which has the property that a op b is guaranteed to be correctly implemented when a, b, and a op b are related to maxint in a certain specified way.

The same section also defines a value of type integer as 'an element of the implementation-defined subset of whole numbers'. A common implementation of type integer defines a value of that type to be an element of the subrange-maxint ..maxint where maxint is defined to be the largest integer that can be represented on that machine. Although this may comply with the Jensen and Wirth [User Manual] definitions, it is not required by the User Manual.

In particular, the implementation described in chapter 13 defines maxint to be 2**48-1 while the largest possible integer value is 2**59. Given all the operations permitted on integers, maxint is the largest integer that satifies the requirements of chapter 2, section B. A subset of those operations can be performed on integers greater than maxint and are not prohibited by the User Manual's combined definitions of maxint and a value of type integer.

*****

4.5b Note regarding relaxing the required ordering of const, type, var and procedure function declarations.

It is possible to group components without relaxing the restriction. As an example, consider a program which is logically grouped into three sections, named lexical, symbol_table, and the_rest. Each section can be assumed to contain label, variable, procedure and function declarations, type and constant definitions, and initialization statements. Under the relaxation-of-ordering proposal, this program may be written as (some variation of) the following:

```
program whatever;
  ...lexical, except for initialization...
  ...symbol_table, except for initialization...
  ...the_rest, except for initialization...

  begin
    ...lexical initialization...
    ...symbol_table initialization...
    ...the_rest initialization and operation...
  end.
```

One may observe here the inadequacy of the relaxation-of-ordering proposal, in that the initialization statements are still outside of the logical grouping.

The alternative solution would write this program as:

```
program whatever;
  ...lexical, except for initialization...

  procedure symbol_table;
    ...symbol_table, except for initialization...

    procedure the_rest;
      ...the_rest, except for initialization and operation...

        begin
          ...the_rest initialization and operation...
        end (*the_rest*)

    begin
      ...symbol_table initialization...
      the_rest
    end (*symbol_table*)

  begin
    ...lexical initialization...
    symbol_table
  end (*lexical and whatever*)
```

Notice that, by adding a few words here and there, the desired result is achieved without changing the language.

*****

4.12 Should dispose, mark, and release be standard procedures? Can some more reliable means be found to re-allocate space on the heap?

Recommendation and discussion:

Jensen and Wirth [Report] section 10.1.2. defines dispose to be a standard procedure. It is probable that the question of the status of dispose arises from the inadvertant omission of part of this section in several printings of the Revised Report. This typographical error cannot be construed to amend the report or the language defined therein.

Section 10.1 permits any implementation to provide "additional predeclared procedures." It is recommended that mark and release remain in this category. For those who choose to implement them, the following conventional definition is suggested:

mark (p) where p is of any pointer type, dynamically opens a new category of allocation. The procedure NEW allocates variables only in the current category. The value given to p by Mark identifies the category.

release (p) if p identifies the current allocation category, release(p) disposes all variables in the current category and closes the category. The category that was current at the opening of the just-released one is made current again. Where the category identified by p is not current, the current categories are released until that identified by p has been released.

Notice that this description is entirely consistent with the common implementation of mark and release without dispose, and is also consistent with an implementation that also provides dispose.

[Editorial comment: A pointer value returned from the procedure Mark may only be used subsequently as a parameter to Release.]

*****

4.14 Note regarding embedding control characters in strings.

The program fragment below shows how control characters can be inserted in strings (see section E for additional discussion).

```
const  formfeed=55,
       carriagereturn=56;

type string = packed array [1..n] of char;
```

where n is some implementation defined integer.

```
var str:string;

     str:='THIS IS A LINE.  ';
     str[16]:= chr(formfeed);
     str[17]:= chr(carriagereturn);
```

The string str now contains two control characters.

*****

5.3 Allow interactive file communication.

Add facilities to allow reasonable communication in interactive environments.

Recommendation:

We conclude that this item is in the language.

Discussion:

I/O to terminals is available in several implementations, with varying degrees of success. Three of these are treated in a following separate writeup. The one which appears most promising is similar to that used by the Berkeley UNIX Pascal interpreter.

The essence of the solution is to recognize that, for input files (those that must be RESET; see 5.13), binding the next character of the input sequence to the file buffer variable (f^) can be delayed until the character is needed to satisfy a request. The possible requests are:

Assignment: variable:=f^ (or use as parameter)

Check on status: eof(f) or eoln(f)

In each case, the request may cause a physical input transfer to occur (in addition to those which may occur on GET). Note that any implementation-specific procedures which interact with Pascal files may also cause such action. There is also a difficulty arising from using the file buffer variable as an actual parameter corresponding to a VAR formal parameter. For this reason, the file must be checked at the point of its use.

Some comment on overhead and implementation is in order. First, one could be hard-nosed and require that an assignment from the file buffer variable not occur until the status of eof had been determined. This would alleviate the need for checks of the first type.

It may be desirable to specify, say as a pragmat, that a given file does/does not require this special treatment. This interpretation of Pascal I/O can be implemented by treating references to the file buffer variable as an implicit procedure (or function) call. While the representation of the file element (f^) is that of a variable, it need not be treated that way. The implementation here makes more explicit the treatment of the Pascal file as a special object.

Some examples:

To copy from an interactive input file:

```
program copy(input,output);
  begin
    while not eof(input) do
      begin
        while not eoln(input) do
          begin
            read(input,ch); write(output,ch)
          end;
        readln(input);        {finish the line}
        writeln(output)       {write the eoln "character"}
      end
  end.
```

To read a sequence of integers:

```
program readlist(input);
var I: integer;

procedure skipspaces;
  begin
    while (input^ = ' ') and not eof(input) do get(input)
  end;

begin
  write(output,'Prompt: ');
  skipspaces;
  while not eof(input) do
    begin
      read(input,I);
      write(output,'Prompt: ');
      skipspaces
    end
end.
```

Notice the order of processing in the second example:

```
prompt
loop
  read
  prompt
endloop
```

This highlights the fact that this implementation forces a method of coding on the programmer in order to use the language correctly. Specifically, the form:

```
loop
  prompt
  read
endloop
```

which might be considered, will not work as expected here. For the test for eof at the loop head will invoke a fetch of the first item in the file, prior to the issuance of the prompt. The user, not having been prompted, might well sit and wait. Thus, some care needs to be exercised in the use of Pascal I/O, at

least in the interactive environment.

An additional remark is that this implementation technique permits timely recognition of end-of-file on an interactive device. Most implementations wait until a get is performed to do an input operation (if needed). Here, the check for eof itself will perform one; if at that time the user has decided to quit, the input can be terminated with whatever the system-defined file terminator is (^D, ^Z, @EOF, and so on).

There are two basic problems with input from interactive files: the buffer-priming problem and the EOF problem. Both arise because the availability of characters from an interactive file (=tty) cannot in general be determined ahead of time. Thus an attempt to refill a buffer after the last character has been passed to the user may or may not succeed, and the characters in this case may not be available until the program explicitly prompts for them. Similarly, a read may be required to determine whether the end of the file has been reached. Combined with the above constraint on read-ahead, the EOF is not detectable in the same way it would be with a non-interactive file.

If there is a problem with output to an interactive file, it is generally due to the following consideration: some systems may treat a terminal as a single device, for both input and output ("half duplex"), and may even use a common buffer for each (regardless of the program's separation of the two). In this case, there is a need to synchronize the input with the output. This problem is treated, in part, in section 5.11, where a mechanism is proposed to allow the user to handle this explicitly.

Using "interactive" files in Pascal is feasible within the language as it is currently defined. Problems may arise due to "incorrect" implementation of the I/O mechanisms. Following are three possible implementations purporting to solve this problem. The third is recommended as the preferred one.

I. The first implementation attempts to treat the interactive file without special indication from the program. Either the distinction is not made (all files are treated the same) or an attempt is made to check this at runtime (when the file is opened). A variation of this scheme is used in many implementations.

RESET - defines eoln=true and f^=blank

The normal I/O proceeds:

```
readln;
read(A,B,...)
```

leaving eoln=true for the next read

Unless the above interpretation of RESET is indicated in the text of the program, there are a number of difficulties

with this method. Among them are the fact that having differing schemas for interactive vs. normal files makes device-independence difficult or impossible to obtain. Also, while this may work for formatted I/O.

II. The second implementation is that used in UCSD's Pascal. A new type, 'interactive', is defined, which is a file of char (or text) with the following characteristics.

RESET does not define f^

READ is defined as:
GET(f); ch:=f^ (i.e., no look-ahead).

Again, this provides differing schemes for interactive and normal I/O. Since any file can be declared interactive, this may result in a new standard, as all text files will likely be so declared.

An additional problem with this method is that EOF may not be detected properly for certain input sequences. For example, consider the input sequence

123<eof>

With the schema:

READ(input,i); if not eof(input) then process(i)

the number 123 is not processed. Note that with this implementation, it is necessary to check eof after read to assure that the data read is valid; hence the above schema.

III. The third implementation, used by the UC Berkeley Pascal interpreter, depends on delaying the binding of input character to file buffer variable until necessary. It appears to satisfy all the requirements of Pascal I/O, insofar as they can be tested by a Pascal program; and also appears to handle both normal and interactive files in the same way.

A schema from which to define this implementation is presented below, together with a sketch of a PDP-11 implementation.

A number of potential difficulties have been mentioned in connection with this method. These are: tricky implementation, high overhead, difficult to teach.

The implementation has no more problems associated with it than any of the others. If the overhead is high, perhaps a pragmat can be introduced to indicate that this method need not be used with the file in question. As for teaching, the problem of explaining why I/O can sometimes be caused by such apparently static tests as eof and eoln is something to be reckoned with. However, the basic idea is that, to achieve a measure of device independence, some complex manipulations have to go on behind the scenes. This is basically the problem of matching several

different external schemes to a single internal scheme, and should be taught in just that light.

Jim Miner suggests difficulties with the following program:

```
procedure quirk(var ch:char);
  begin
    ...
  end;

    ...
    quirk(f^);
```

Invocation of procedure quirk, as shown above, requires evaluation of the file buffer variable (and possible I/O).

A useful exercise for any of these three methods is writing a program to copy a character file to a standard ('disk') file exactly, with all line structure maintained - the standard program can be found in Jensen and Wirth, p. 59.

Implementation Schema for version III I/O

The following purports to implement the UC Berkeley treatment of files. The key is to delay moving the next character from the physical file to the file buffer variable until it (or knowledge of its presence) is actually required. This means that I/O may occur (for input) during a check on eof or eoln; as well as during an assignment from the buffer variable (including its use as a value parameter). A problem does arise, as indicated above, if the buffer variable is used as a var parameter.

In the following, assume: `f: file of T;`

Also, the following (non-syntactic) implicit variables are referenced (items enclosed in <>'s are optional, and are used for undefined error state checks; items in {}'s are operations which need further elaboration):

```
f.wait,f.eof: boolean;
<f.defined: boolean;>
```

```
procedure intransfer(f);
begin
  if f.eof then error
  else begin
    f^:= {next component};
    f.wait:= true;
    f.eof:={end-of-file};
    <f.defined:=not f.eof;>
  end
end;


procedure get(f);
begin
  if f.wait
  then intransfer(f);
  f.wait:=true
end;


function eof(f):boolean;
begin
  if f.wait
  then intransfer(f);
  eof:=f.eof
end;


function reference(f):T;
begin
  if f.wait
  then intransfer(f);
  <if not f.defined then error else>
  reference:=f^
end;


procedure scopeexit(f);          {or:}
begin
  if f.wait
  then intransfer;
  {close(f)}
end;
```

```
procedure reset(f);
 begin
   {openforinput(f)};
   f.eof:=false;
   f.wait:=true;
   <f.defined:=false>
 end:


procedure rewrite(f);
begin
  {openforoutput(f)};
  f.eof:=true;
  f.wait:=false;
  <f.defined:=false>
end;


procedure put(f);
begin
  if not (f.eof <and f.defined>)
  then error
  else outtransfer;
  <f.defined:=false>
end;


procedure assign(f,v);
begin
  if f.wait
  then intransfer;
  f^:=v;
  <f.defined:=true>
end;


procedure scopeexit(f);
begin
  if f.wait and f.eof
  then error;
  {close(f)}
end;
```

PDP-11 Implementation Sketch

The following represents one possible PDP-11 implementation of the above schema.

```
;File variable definition:
f:.WORD status
        wait = 100000
        eof  =  40000
        def  =  20000
  .WORD pointer
;
;
GET:    TST     f
        BPL     1$
        JSR     PC,INTRANS
1$:     BIS     #wait,F
        RTS     PC

EOF:    TST     f
        BPL     1$
        JSR     PC,INTRANS
1$:     BIT     #eof,f
        ...
        RTS     PC


REF:    TST     f
        BPL     1$
        JSR     PC,INTRANS
1$:<    BIT     #def,f
        BNE     2$
        ERROR
2$:>    MOV     @f+2,v
        ...

; Note abbreviated code
```

```
READ:   ; v:=f^; get(f)
        TST     f
        BPL     1$
        JSR     PC,INTRANS
1$:<    BIT     #def,f
        BNE     2$
        ERROR
2$:>    MOV     @f+2,v
        BIS     #wait,f
        ...
```

*****

5.9 Allow or disallow files as components of structures.

Recommendation:

We conclude that this item is in the language.

Discussion:

There is no prohibition in Jensen and Wirth for the inclusion of files as components of structures. The question addresses the reasonability (i.e., implementability) of the

prototypical constructs array...of file type; record...f: file_types; ...end; and file of file_type. The question arises from a confusion of the Pascal file concept with the widespread use of the word file to denote a dataset on external storage. The abstract notion in Pascal is of a sequence of components: i.e.,file of integer denotes a sequence of integers. The language defines a bundle of properties and operations as part of its notion: this bundle forms, in some usages, an idealization of those properties and operations pertinent to magnetic tape datasets and similar external quantities.

However, when the abstaction is separated from its common implementation, and when the word sequence is used where appropriate, it is readily apparent that there is indeed meaning in the concepts of a sequence of sequences (file of file_type), an array of sequences (array...of file type), and a sequence as a field of a record. Thus the Workshop recommends the recognition that "a file as a component of a structure" is a meaningful notion and that such constructions are permitted by the language. It is noted that this area is not well understood and is deserving of more published explanatory material.

[Editorial comment: There is a question re the semantics of get and put when applied to a file of array of file.]

*****

5.12 Discussion regarding "core" files.

This was suggested as a possible solution to the ENCODE/DECODE problem. See 3.12 and 5.9 in this section and section E. However, as shown below, the current language definition currently permits this useful feature. See also proposal 4.6 in section G.

Discussion:

Jensen and Wirth (chapter 9) defines a file to be a structure consisting of a sequence of components--- all of which are of the same type." Note in particular that no mention is made of a physical device. A few paragraphs later in that chapter, the comment is made that "the sequential processing and the existence of a buffer variable suggest that files may be associated with secondary storage and peripherals." Files are not required to be so associated though they may be.

An example of when it is convenient and advantageous for files not to be associated with secondary storage is when one wants capabilities similar to Fortran's ENCODE and DECODE.

```
Consider: type bufferspace = file of char;
          var buffer:bufferspace;
```

The variable BUFFER can be an in-core file, that is, a

sequence of characters in memory which is ordered and whose length is not fixed by the definition. Standard file-handling operations can be performed on this sequence including read and write.

*****

8.3 Note regarding arbitrary physical record layouts.

Arbitrary physical record layouts can be described with the Pascal record type constructor. If the variant part occurs before the end, that variant can be described in another type definition and its name can be used where it is needed in the record definition.

```
type dcb = record
             I:integer;
             case boolean of
               true: (TF:boolean)
               false: (K:integer)
             end;
             J:integer
           end;
```

The definition of dcb violates Pascal syntax rules because the case variant part is not last. However, if the case variant is defined earlier, its name may be used without violating the syntax rules. That is,

```
type subdcb = record
                case boolean of
                  true: (TF:boolean)
                  false: (K:integer)
                end
              end
     dcb = record
             I:integer;
             sdcb:subdcb;
             J:integer
           end;
```

The use of filler fields when necessary combined with a knowledge of the compiler's packing algorithm permits a syntactically legal Pascal description of nearly all existing records with which a Pascal program must interface.

*****

SECTION E TOPICS DISMISSED

In the case of the majority of the topics considered, the Workshop recommended against making any changes to the language or its definition. The inclusion of a topic in this section does not necessarily imply that the Workshop considered it a bad idea per se. Topics were dismissed for a number of reasons including:

1) the extension proposed already exists in Pascal

2) the extension proposed was really a change to the language and would invalidate existing programs

3) the extension proposed, while desirable, was not considered important enough to justify adding complexity to the language

4) the programming problems motivating a proposal for an extension can be solved in other ways

5) the proposal was considered too implementation dependent to be included in the language

6) the Workshop could not reach a decision

7) no-one attending was interested

1.2 Allow subranges in the CASE statement.

Allow subranges (given by constant..constant) in CASE label lists.

Recommendation:

We recommend that this feature not be included in the standard.

Discussion:

Our review of the subject disclosed the following problems:

a) There is no canonical collating sequence for characters,

b) Checking for overlapping ranges of case-labels may cause implemenation difficulties,

c) There is the argument that this syntax should be consistent with the variant record syntax and that the Working Group has been more negative about ranges in that situtation.

*****

1.4 Add ASSERT statement.

Add an ASSERT statement, eg. ASSERT boolean-expression which causes a run-time error if the program was compiled with assertion checking on and the boolean expression evaluates false.

Recommendation and discussion:

The Workshop's feeling was that there was only minimal value to this feature but that if an implementation chose to provide it, it should do so in the form

<assert statment> ::= ASSERT (< boolean expression>)

so that it could be implemented via a predefined procedure and so that an implementation which did not have the feature could ignore it by providing an ASSERT procedure which did nothing.

*****

1.5 Disallow out-of-block GOTO statements.

Restrict GOTO statements from jumping out of blocks. As the main justification for such GOTO statements is handling exceptional conditions, this is dependent upon some form of structured error recovery. (Most Pascal-P2, P4 derivatives already do not support jumping out of blocks.)

Recommendation:

Although many people dislike this feature, it is EXPLICITLY allowed on page 150 of Jensen and Wirth [Revised Report] and so we recommend NOT disallowing it.

*****

1.7 Allow "FOR i IN" set-expression loops.

Extend the FOR loop to include a FOR i IN setexpression syntax.

Recommendation and discussion:

We recommend that this feature not be included in the

language and do not recommend any syntax for implementing it. A. Sale provided the following equivalent construct. It requires the FIRSTMEMBER function defined in section E, 3.14.

```
var
    i : thing;
    s : set of thing;
begin
    ....
    { the loop invariant R = "the values corresponding to
       the members of the set s have not been processed" }
    while (s <> []) do begin
        i:=firstmember(s);
        ....
        s:=s-[i];
    end;
    { s is now [] (destroyed), and i is undefined }
    ....
```

*****

1.8 Add a generalized LOOP construct.

Recommendation:

We recommend that this feature not be included in the standard and do not recommend any syntax for implementing it. The Working Group has considered this feature and rejected it as unnecessary.

*****

1.9 Extend the WITH statement to allow renaming.

Clear up the semantics of the WITH statement, or extend the WITH statement to be a "renaming" statement. This would allow the use of array abbreviations in addition to pointers and records, and allow abbreviations of two variables of the same type, e.g.

```
with x = b^.c[i], y = d^.c[i] do s1.
```

Recommendation:

We recommend no changes or extensions to the WITH statement in the language.

Discussion:

We recognize problems with the current WITH statement

(e.g., aliases). The solutions suggested may entail significant, even unacceptable, implementation overhead.

Consider the following example:

```
var i:integer;
    r:record x:integer end;
begin
   with i=r do i.x:=1
end.
```

It is not clear whether i should be defined within a new scope or whether this should be disallowed as a redefinition of i within the current scope. (Defining it within a new scope would involve a change (extension) in the scope rules of Pascal.)

This extension creates additional opportunities for aliasing problems.

*****

1.10 Add an EXIT statement.

Recommendation:

We recommend that neither a procedure nor a structured statment EXIT facility be added as an extension. The GOTO statement, in spite of its faults, provides this and many of us feel that duplicating this facility clutters the language.

*****

1.11 Add a NULL statement.

Add a new statement consisting of the single identifier NULL. This statement would have no effect.

Recommendation:

We do not recommend adding this statement to the standard. We note that a comment {NULL} will serve the same purpose.

*****

1.12 Add embedded assignments.

Allow an assignment "statement" to be used wherever an

expression is now allowed. The value of the assignment statement would be the value of its right-hand side.

Recommendation:

We recommend not adding embedded assignments as an extension. We consider this contrary to the basic nature of Pascal because any use of this feature would cause side-effects.

*****

1.14 Clarify the semantics of the WITH statement.

Recommendation:

Jensen and Wirth covers this topic in section 9.2.4 of the Revised Report.

[Editorial Comment: See further the British Standards Institution draft.]

*****

2.1 Allow empty parameter lists.

Allow empty actual parameter lists, so as to distinguish function calls that take no parameters from simple variables for readability.

Recommendation:

It was not recommended because it is a minor syntactic change which is not necessary as it is easily indicated by an empty comment.

*****

2.4 Allow/require repetition of parameter list of forward procedures.

Allow, or even require, that the definition of a procedure or function that has been declared forward include the parameter list. The current situation often results in looking back a number of pages to determine what parameters are being passed.

Recommendation:

This was not recommended because it is easily indicated by a comment.

*****

2.5 Restrict var parameters to use call-by-reference.

Restrict var parameters to be passed by reference and disallow the use of a value-result mechanism. A programmer using var parameters for speed would then be assured that the structure being passed was not copied. However, this proposal might limit even further the ability to link FORTRAN routines to Pascal programs.

Recommendation and discussion:

This is an implementation detail. We recommend that the Working Group examine the semantics of using side-effects in programs.

*****

2.6 Restrict call-by-value parameters to be constants.

Restrict call-by-value parameters to be constants, and therefore unassignable, within the procedure body.

Recommendation:

This represents a change in the existing language and would cause many existing programs to have to be rewritten. We do not recommend this change.

*****

2.8 Restrict functions to have no side-effects.

Restricting functions to have no side-effects whatsoever could facilitate optimization and formal proofs of correctness.

Recommendation and discussion:

We do not recommend this change, especially since it is very difficult to enforce. It was noted that many existing

functions have side effects and that this change would invalidate much existing code. It has also been noted that such things as I/O from functions would not be possible if this were the case (i.e. interactive file EOF, see 5.3).

[Editorial Comment: Does 'interactive file EOF' have a side effect if the program cannot determine that something has been changed by calling EOF? Relative to the program there is no side effect; relative to the programmer (who must adopt a certain scheme to synchronise user and program) there is a side effect.]

*****

2.9 Allow the declaration of INLINE procedures.

Allow the declaration of INLINE procedures, which generate direct code at each invocation, rather than a call to a shared piece of code.

Recommendation:

We reject this feature, because it is an implementation dependent compiler feature, and tends to mix representation with abstraction. Directions to the compiler have no place in the actual language.

*****

2.11 RETURN (value) statement from functions and procedures.

Recommendation and discussion:

This would make a major change in the concept of structure as present in Pascal as it would allow multiple paths out of a function, and create a second method for returning function values. We recommend this proposal be dropped.

*****

2.12 REFIN, REFOUT, INOUT, OUTPUT procedure parameter forms.

These parameter types are proposed to allow the user to specify the manner in which the parameters are passed to the procedure; for instance, read-only and write-only call-by-reference.

Recommendation:

These additional parameters forms are not necessary, and their addition would constitute an unnecessary change to the language.

*****

2.13 Variable number of parameters to a procedure.

Recommendation and discussion:

Implementation of this would complicate many things, and READ is difficult enough now. No clear way to specify a variable number of parameters has been proposed. We recommend that this proposal be dropped.

*****

2.14 Generalized FOR statement.

Recommendation:

No changes in the FOR statement were recommended.

*****

3.1 Redefinition of div and mod.

Redefine div to truncate toward negative infinity, rather than zero (thus also causing mod to return different values).

```
-1 div 3 = -1;
-1 mod 3 = 2;
```

Recommendation:

We recommend the definition of div with respect to negative operands be an implementation dependent feature, and the report should be amended to so state. If the user wishes to assure portability the appropriate interpretation of division may be implemented with a Pascal function as given by the example below.

Discussion:

div is usually implemented according to hardware dictates. If it is truly an implementation choice, it is preferable that div truncates toward minus infinity so that mod would be

mathematically correct.

On a machine that truncates toward zero on integer divide, the following code can be generated or called for i div j. (mdiv represents the hardware divide.)

```
if j < 0 then  {since i div -j = -i div j}
  begin
    i := -i;
    j := -j
  end;
if i < 0 then
  quotient := (i-j+1) mdiv j
else
  quotient := i mdiv j
```

If only a positive hardware divide exists, or if mod need be computed with negative operands, the following code can be called...

```
if j > 0 then  {divide by positive}
  if i >= 0 then  {divide positive by positive}
    begin
      quotient := i div j
      {remainder := i mod j}
    end
  else  {divide negative by positive}
    begin
      quotient := - ((-i-1) div j) - 1;
      {remainder :=  j - 1 - ((-i-1) mod j)}
    end
else  {divide by negative}
   if i > 0 then  {divide positive by negative}
     begin
       quotient := -( (i-1) div (-j) )-1
       {remainder := ( (i-1) mod (-j) ) + 1 + j}
     end
   else  {divide negative by negative}
     begin
       quotient := (-i) div (-j);
       {remainder := -((-i) mod (-j))}
     end
```

*****

3.2 Short-circuit AND and OR.

Either redefine AND and OR to be short-circuit (aka conditional, sequential) operators, only evaluating an expression as far as necessary to determine its value; or leave them as they are and add short-circuit operators to the language. e.g.

```
while (i < arraytop) cand (a[i] <> pattern) do i := i+1
```

Recommendation:

We recommend that AND and OR should be left as defined, i.e. the implementer may choose short circuit or complete evaluation, user beware!

Discussion:

The effect of short circuit AND and OR (CAND and COR) can be programmed in existing Pascal. The majority of the Workshop felt that the cost of implementation (size, introducing features, etc.) does not justify the benefit of extensions.

We firmly reject the concept of introducing complete evaluation operators, such as LAND or LOR.

*****

3.3 Type Transfer Functions.

Extend type-transfer functions, at least for scalars (excluding reals) and pointers (as an implementation-dependent feature) to provide an inverse ord function, thereby reducing the necessity of subverting type-checking by the use of variant records with no tag fields. e.g.

```
c := color(i)
```

(Also specify that enumerated types start at 0, and allow ord(pointer).)

Recommendation:

We recommend that there should not be functions in the language to allow type transfer from pointer to integer or vise-versa. In particular ORD is especially inappropriate, since pointers are not ordered.

We recommend that type transfer functions from integer to ordered scalar types not be added to the standard but an experiment is suggested in G.

*****

3.5 Addition of the operator NOT IN.

Add NOT IN to the relational operators, to provide a nicer method of asking if an element is not a member of a set.

Recommendation:

We do not recommend that NOT IN be added to the language.

Discussion:

The reason for this is that it can be done currently, not too awkwardly, and it would only add complexity to the language. e.g.

```
if I not in [A,B,C] then ...
```

can be written

```
if not (I in [A,B,C] ) then ...
```

*****

3.6 Set Constructors.

Add a method for determining the type of a constructed set, thereby allowing sets to be implemented with non-zero bases.

```
packset = set of 100..115;

var a: packset;
...
a := packset([100..102, 115])
```

Where the set 'a' could be stored in 16 bits, rather than 116.

Recommendation:

The Workshop decided not change the syntax for set constructors.

Discussion:

In compiling a set constructor like [J+K], the compiler cannot determine how much space to reserve as set of integer is too large and it's type is a little vague . If the expression were required to be MYSET[J+K], where MYSET is a type identifier, then better type checking could be done for both the elements of the set being constructed and the set. If it is part of a surrounding expression, exact space requirements are known and range checking can be done on dynamic elements.

Making this change would invalidate large numbers of programs. The proposed syntax resembles array indexing while an alternative, MYSET(J+K), is similar to a function call. Putting the type identifier after the closing bracket would provide only marginally useful information to an LL(1) recursive descent compiler.

*****

3.7 Structured Value Constructors.

Add a general structured value constructor, to fill arrays, pass structured values as parameters without first assigning to a dummy variable, etc.

Recommendation:

The Workshop decided this should be left out of the language, largely because every known use of this is expressable in standard Pascal without difficulty.

Discussion:

A structured value construction would look something like "MYTYPE(" <element list> ")" and would represent an array or record.

These values could not be used in comparisons since the Workshop decided against this for all records and arrays because of variant field and packing attribute problems. If these values are not then allowed to be indexed or selected, the only use would be in simple assignment statements and call by value procedure and function parameters. Typically a compiler will produce code that runs no faster than a user's equivalent that assigns to each field or element separately and explicitly, and almost necessarily will waste space with a temporary in assignment statements because of the possibility of

" A := ATYPE(A[1], A[0]) "

even when this type of anomaly isn't present. Allowing indexing and selecting of these values seems of little use since a large amount of calculations would typically be discarded and would totally waste computation time, in particular, if selecting or indexing by a constant. To index by a variable can be simulated with a case statement. An often claimed use of constructors is, initiallization of variables where some of the fields need computed values but this can be accomplished with an assignment by a constant, see 4.5, and then assign over some parts with computed values.

*****

3.8 Array Slices and Catenation.

Add array slice (sub-array) capabilities and a concatenation operator, e.g.

```
a[4..9] := a[5..8] & b[0..1];
cstring := cstring[1..pos] & dstring &
                              cstring[pos+1..length(cstring)];
```

Recommendation:

We recommend rejection of adding array slice capabilities and a concatenation operator.

Discussion:

We believe this would introduce significant and unacceptable complexity to the language.

*****

3.9 Redefinition of Precedence Rules.

Redefine precedence rules to make the parentheses in, for example, (a < b) AND (c = d) unnecessary.

Recommendation:

We recommend that precedence rules not be redefined.

Discussion:

When A,B, and C are booleans, the expression A = B and C yields A = (B and C) under the current precedence rules and (A = B) and C under one alternative.

Compilers that have set an unfortunate precedence may do the best possible with the situation if they issue a warning to anyone using relational operators with boolean operands:

"WARNING: THIS EXPRESSION MAY BE DIFFERENTLY PARSED ON A STANDARD PASCAL SYSTEM. FULLY PARENTHESIZE IT FOR SAFETY."

*****

3.10 Comparison of Arrays and Records.

Add the ability to use = and <> to compare (at least unpacked) arrays and records (it is likely that most compilers descended from P2 or P4 provide this capability already.)

Recommendation:

We recommend rejection of the proposal to add = and <>

comparisons on arrays and records.

Discussion:

There was considerable experience within the group indicating that significant and unacceptable implementation expense and difficulty would occur when implementing this otherwise desirable feature. In order to keep Pascal easy to implement rejection of this feature is recommended.

1) There are problems in determining the length and the fields to be compared for records with variant structures. This would justify excluding comparison of structures with variants.

2) The structures requiring fill in the storage allocation must assure that the fill is identical or specifically ignore the fill in doing comparisons. This would apply to both arrays and records.

3) A real component cannot be reliably checked using integer arithmetic.

4) If the elements of an array were records with variants they may not be comparable due to the first reason. Thus to allow these comparisons would require specifying a large number of exceptions. It is much simpler not to allow these comparisons, as they can be done with standard Pascal (although awkwardly).

[Editorial Comment: Also, structures containing REAL components present further problems.]

*****

3.11 Packing and Unpacking.

Allow the packing and unpacking of records as well as arrays. One possibility is to simply use the pre-defined procedures pack and unpack, without specifying a length. Another is to allow packing and unpacking via type names. e.g.

```
type a = record ... end; b = packed record ... end;
     c = array [0..15] of boolean;
     d = packed array [0..15] of boolean;
var x: a; y: b;
    z: c; w: d;
...

x := a(y); (unpack y)
y := b(x); (pack x)
z := c(w); (unpack all of d)
```

Recommendation and discussion:

It is an open question whether packing and unpacking of records is allowed in the language already because

1) type compatibility rules are inadequately defined

2) packed is defined in section 6.2 of Jensen and Wirth [Report] to have no effect.

Example of packed compatibility problem:

```
var a:record x,y:real end;
    b:packed record x,y real end;
```

It is not clear whether or not A and B are compatible. If name compatibility is used, these are not compatible. If structure compatibility is used, these are compatible because packed has no effect according to the report.

We recommend that A and B not be compatible and that no other extensions be made. While it might be possible to do it within the language with a predefined procedure, we feel that if the tag field of a variant record is not stored or it is unassigned it may be impossible to do the operation.

*****

3.13-3.14 Additional Functions.

3.13 Add min and max as standard functions.
min(arithmeticexpression, arithmeticexpression),
max(...): real or integer

3.14 Add standard functions to extract the cardinality and the first element of a set.
card(setexpression), first(setexpression)

Recommendation:

MIN, MAX, MEMBERS, FIRSTMEMBER and LASTMEMBER are the names we recommend the implementer use if it is desired to implement these predefined functions. We are not recommending that they be added.

Discussion:

FIRSTMEMBER and LASTMEMBER are undefined when these functions are applied to an empty set.

MIN and MAX determine and return the minimum or maximum value of the two operands supplied.

MEMBERS(X) is the number of elements contained in the set X.

FIRSTMEMBER(X) is the first element from the enumeration which is contained in the set X.

LASTMEMBER(X) is the last element from the enumeration which is contained in the set X.

[Editorial Comment: For further discussion see
O.-J. Dahl, E.W. Dijkstra, C.A.R. Hoare,
Structured Programming, Acedemic Press Inc., 1972]

*****

3.15 Access to Size of Variables.

Add access to machine-dependent information about the size of variables.

```
sizeof(variable): integer;  -- returns size in some machine-
                            -- dependent units
sizeof(variable, t1, t2...) -- get size of variantrecord
```

Recommendation:

We do not recommend adding access to the size of variables.

Discussion:

It is machine dependent and should be implementation dependent. This mixes representation and implementation.

*****

3.17 Conditional Expressions.

Add Algol 60 style conditional expressions so that statements such as

x:= if y >= 0 then y else -y;

can be written.

Recommendation:

We do not recommend adding conditional expressions to the language.

*****

3.18 Eliminate the standard function SQR.

Recommendation:

We recommend not changing the language for the function "SQR".

*****

3.19 Multiple Assignments (eg. x:=y:=2).

Recommendation:

We recommend not changing the language to add multiple assignments.

*****

3.20 Allow / for div.

Allow / to represent div when used with integer operands.

Recommendation:

We recommend that "/" and div not be changed. These operators perform different operations and they are defined.

*****

3.21 Implementation inquiries.

Recommendation:

We recommend that implementation inquiries not be added as predefined functions to the language.

Discussion:

A collection of constant definitions can be made for a given application area and included in programs that need it.

*****

4.1 Allow type b = packed a;

Allow an <unpacked structured type> to be a type identifier, thereby allowing something like b = packed a;

Recommendation:

The amount of effort to implement outweighs the advantages.

Discussion:

This is primarily a shorthand notation which might reduce errors, but it requires a fairly significant compiler overhead to recalculate the new, packed versions of the offsets of the structure.

An issue that falls out of this and should be addressed by the ISO standardizing group is what constitutes valid arguments to the procedures pack and unpack. This is especially needed if strict name type compatability is decided upon for structures.

*****

4.2 Minimum sizes of data types.

Decide upon the minimum set size, the minimum range of the type integer, the minimum range and precision of type real, and the ability to specify the desired range and precision of real numbers. This would allow small machines to use small, easily manipulated representations for most values, and only incur more expensive representations when extra precision is needed.

Recommendation and discussion:

It was decided not to specify minimum sizes for any types; it might be perfectly reasonable to have only 8-bit integers in some applications.

It was also decided not to standardize extended precision forms at this time. The details of the standard would rely on the type (real or integer), precision, application, and on machine factors. More user experience with extended precision is needed before agreement can be reached on a standard.

Jensen and Wirth implies that the result of an operation on integers may be undefined if it or it's arguments absolute value exceeds MAXINT (which is implementation defined). The question, is what should a compiler do with a declaration like "K: 0..N" where N is bigger than MAXINT. Many implementations will not produce good code where K is used. The compiler should

be required to flag an error when it cannot correctly compile something.

See section C for recommendation regarding set of char.

*****

4.3 Scaled integers.

Include scaled integer arithmetic, for use in business applications. e.g.

```
var a: 0..10000000 scaled .01 {or 1//100}
```

would be capable of containing numbers from zero to ten million, with a precision of pennies (hundredths).

Recommendation:

The Workshop decided that this feature should not be added to the language.

Discussion:

The necessity of this has not been shown but merely infered from it's inclusion in DOD's IRONMAN. A programmer can always keep track of his own scaling and introduce scaling constants into expressions where necessary. The proposals seen so far all involve difficult rules and restrictions to the use of scaled integers and complications and exceptions to the concept of type compatablity (a very touchy concept these days).

*****

4.4 Constant expressions.

Add the ability to use simple constant expressions in place of the constant allowed now. e.g.

```
const  bell = chr(7);
       maxelements = 100;
var a: array [0..maxelements-1] of integer;
```

Recommendation and discussion:

The Workshop decided against this feature.

At first the Workshop decided that only a fully general implementation, where constant expressions could appear

everywhere that constants could, would be acceptable. Later, the following examples of code using constant expressions were discovered:

```
type T = (N) { an enumerated scalar type with a single value } ;

     S = (SL)..SH { a subrange };

     R = (A + B) * C .. D;

     R = (A,B,C)
```

The Workshop decided that it was unacceptable to require or suggest that a compiler be able to cope with the amount of lookahead or complexity that appears to be needed to parse these examples, and others not listed here, since this is not necessary in any existing syntax. Note: these problems do not arise if constant expressions are allowed only in the const section or if constant expressions with leading "("s are not allowed.

*****

4.7 Require tag fields.

Require all variant records to include a tag field, and disallow changing the tag field without changing the value of the entire record (see structured constructors above). This would forbid the use of variant records to subvert type-checking (and so would need adequate facilities to "cleanly" do type transferring).

Recommendation:

No. This would be a restriction of the standard. Implementations are free to require tag fields, but realize that such an implementation would not be Standard Pascal.

*****

4.10 Value initialization.

Include some ability to initialize variables in the declaration section, rather than in the actual code. This could allow a savings of code space on many machines, and may enhance readability. e.g.

```
var i: integer := 0;     or, alternately,
var i: integer;
init i := 0;
```

Recommendation and discussion:

The Workshop decided this should not be provided. The proposals examined were generally along the lines of

"var R : T = <value>"

where if T could be structured the <value> would be syntactically as in structured constants (4.5a) and this initialization would take place each time the scope was entered.

The Workshop decided that if this were accepted at all that the <value> should be fully general in allowing arbitrary expressions including other variables. This then begins to look like general structured value constructors which this subgroup rejected (3.7), the reasons for which being applicable here.

Furthermore, no compelling reasons for having this feature were known to this group so it was viewed as language cluttering. Many even felt that textually separating the initialization of a variable from the algorithm that uses it, constitutes a hazard.

[Editorial Comment: It should be noted that nothing in the language forbids a compiler from interpreting assignment statements at the beginning of a procedure's statement part as value initializations.]

*****

4.11 Dynamic Arrays.

Add fully dynamic arrays, whose bounds are determined at scope-entry time, as in ALGOL-60. e.g.

```
var size: integer;
procedure arraydemo;
var a: array [1..size] of integer
```

Discussion:

There are three classes of 'dynamic' arrays; extensible arrays, dynamic arrays, and flexible arrays. Flexible arrays (truly dynamic arrays, with dynamic subranges) are major extensions to the language and are prohibitively expensive to add. Dynamic arrays, which should include dynamic subranges, or arrays with indices specified at scope entry have an obvious syntax and equally obvious semantics, unfortunately some ambiguities exist...

```
program messedup;
  var n: integer;

  procedure one;
    type dynrange = array [0..n] of integer; (n is a variable)
    var dynarray: dynrange;

    procedure two;
      var anotherdyn: dynrange;
    begin
      anotherdyn := dynarrary;
    end;

    begin
      n := 5;
      two;
    end;

  begin
    n := 10;
    one;
  end.
```

Of course, a restriction could be made on the use of the variable n, in the manner that assignments to loop induction variables are disallowed.

Adjustable array bounds is being taken up by the Procedures subgroup. (See section C1, 2.3.)

*****

4.13 Unsigned integers.

Allow the declaration of unsigned integers which may use all the bits available in a machine word to represent magnitude.

Recommendation and discussion:

After much discussion about the trouble one can get in when allowing unsigned integers...(assume a 16 bit word):

```
var  i: 0..65535;
     j: -32767..32767;
begin
  i := 2*65535; {requires different overflow checking}
  i := 0-2; {requires overflow checking on a subtract}
  i := 1;
  j := -1;
    if i < j
    then  {requires check to make sure that j is positive
                  before an unsigned comparison can be made}
```

```
      .
      .
      .
end;
```

we considered what these things are actually used for.

This proposal is submitted by the implementation group. Integer is -32768...+32767 but the need is for values in the range 0..+65535 which only uses one word of storage. The Workshop decided that this is too much of an implementation detail to consider. It was noted however that at least one implementation is known to accept these requests but fails to generate correct code in all cases where such variables are used (watch out for those integer comparison operations).

*****

4.14 Embedding control characters in strings.

The Workshop decided that this was an unnecessary addition to the language. Jensen and Wirth specifies no restriction on which characters may appear in strings in Standard Pascal. If a given editor or operating system DOES prohibit the input of certain characters that is an implementation restriction and not a subject for extension to the language. See section D for a way of inserting control characters in strings.

*****

4.15 Allow type a = 1, 5..7, 14

Recommendation:

We recommend not to add this construct to the language.

*****

4.16 Automatic conversion of types.

Allow automatic conversion of types as in:

```
i := 2.5
```

where i is of type integer (is assigned the value 2).

Recommendation:

Coercion to a type that is a superset of the coerced type is one thing, but automatic conversion the other way has long been recognized as error-prone.

*****

4.17 Pointers to objects in the stack.

Recommendation:

Although this can allow more efficient access than arrays on some machines, other machines keep the concept of the stack and the heap totally separate. Also, programs are that much more prone to errors when pointers into the stack are allowed. We recommend against such an extension.

*****

5.5 Add formatted reading.

Make the IO operations allowable on a textfile symmetric to more easily process business type data. Note that the inclusion of recovery blocks allows the flexible processing of bad input data, e.g.

```
read(cardfile, seq: 6, id: 8, x: 10 :4)
```

Recommendation:

We recommend that this item be dismissed.

Discussion:

This item was not of sufficient interest to justify its inclusion in the language. If it is desired to include it, however, the syntax should be the "inverse" of the WRITE statement.

The semantics, however, are not so obvious, and a number of decisions need to be made.

*****

5.6 Allow the reading and writing of scalar identifiers.

```
var c: (red, blue, green);
...
write(output, c)
```

```
read(input,boolval)
```

Recommendation:

We recommend that this item be dismissed.

Discussion:

If this capability is implemented, it should follow the syntax and semantics used for other types. In particular, field length should be extended to accomodate the identifier being written. For reading, the problem of misspelling must be considered; for example, to read values of the type

```
color = (blue1,blue2,...);
```

the delimiting characters need to be well defined. A similar example (blue,bluegreen,...) shows that not all problems are easily solved. The best solution, perhaps, is to treat these values, for input, as identifiers in the Pascal sense. This eliminates any of the above problems. At the same time, it makes clear the need to specify a minimum length of significance, so that distinct values will be distinct in all implementations.

Implementors should be aware of the overheads involved in this capability before adding it to the language. For both input and output, a symbol table mechanism is required.

*****

5.7 Raise a run-time exception if a specified field width is too small for the writing of a number or string. (Needs structured error recovery to be useful.)

Recommendation:

Jensen and Wirth [Report] specifies that a field width is expanded as required if it is too small for the value to be written. We recommend that this item be rejected.

*****

5.8 Extended formatting to be similar to COBOL.

Extended formatting to allow capabilities similiar to COBOL PICTURE formats, ie. floating dollar-sign, suppressed or non-suppressed leading zeroes, etc.

Recommendation:

This seems to require more effort and introduces more complexity into the language than can be justified by any benefits it produces. This suggestion is rejected.

*****

5.10 Allow access to file attributes.

Allow specification and access to file attributes, like size, etc., yielding facilities similiar to some Burroughs compilers.

Recommendation:

We recommend that this item be dismissed.

Discussion:

Such access was felt to be system-dependent, and also not presenting any implementation/language problems. For this reason, we did not consider it further.

It should be noted that some file attributes may change during the execution of a given program. For this reason, it can be expected that any form of access to these attributes will require the effect of a procedure call (to update these values).

Two suggestions have been considered. The first implements the access as a procedure call with the following prototypical declaration:

procedure status(f:<anyfiletype>; var rec:<fileattributetype>);

The type <fileattributetype> is unspecified, but will presumably be a record whose components represent the possible attribute values. The name "status" is not particularly recommended. Possible names considered are status, attributes, fileattributes.

An alternative suggestion related to this one proposes a separate procedure for each attribute. This is rejected as polluting the name space. Since record field identifiers exist only within the scope of the record name, the names used are still available for use by the programmer in other contexts. The type <fileattributetype> will be a predeclared type and so known in any program. Good programming style may inhibit the use of these names in a program using them as attribute names, but there is nothing syntactical to prevent it.

The second proposal, not fully thought out, specifies that the name of a (Pascal) file variable be treated as a record name, and that the attributes of a file be referenced as fields

of the file: file.attribute. This raises a number of problems, not the least of which is syntactical (a file type is not a record). Another is the hidden cost of the denotation. An assignment or use in an expression will require a system call, at least if the attribute accessed can change dynamically (e.g., terminal status, protection bits).

*****

5.11 Provide a facility to "flush" the buffer associated to a file.

Recommendation:

We recommend that this item be dismissed.

Discussion:

We felt that there were good reasons for providing such a facility. This is, however, an implementation detail and should be decided by the implementor. The most common mechanism is a predefined procedure, declared (prototypically):

```
procedure break(f:<anyfiletype>);
```

Break is a common name for this procedure. Other names which have been used are flush and sync ("synchronize", as in synchronize internal and external versions of this file).

This facility solves a basic need: to force out the contents of a system buffer to the file (or perhaps to dump the contents of an input buffer). This is needed to force out prompts; to get as much data as possible to a file (for debugging; or to synchronize the program's idea of the file with that of the system); or to synchronize the input with the output (as for a half duplex device; also, the system may use a common buffer for input and output to the same device).

*****

5.12 Provide "core" files.

Recommendation:

We recommend that this item be left up to the individual implementor.

Discussion:

It was felt that to require the provision of "files" residing exclusively in core was too much of an imposition on implementors of the Pascal language. No extension to the

language is necesary. See 5.12 in section D.

*****

6.3 Allow OWN variables, (this should only be considered in the absence of approval of some kind of encapsulation scheme).

Recommendation:

We recommend that OWN variables, as known in Algol 60, should not be provided.

*****

6.4 Add capabilites for overlays, if this is possible to do in a standard way. Many operating systems may impose restrictions making agreement upon this impossible.

Recommendation:

We believe that overlays should not be handled in the Pascal language. Such facilities might be supplied by a linker of separately compiled procedures/modules or invoked by compiler directive in a single compilation.

*****

8.1 Include the ability to access hardware memory, registers, or instructions in some clean fashion, for machine-specific code.

Recommendation:

We recommend that access to underlying hardware be handled only by compiler directive as this is highly dependent on the implementation and that assembly language be available only in external routines and not as inline code.

*****

8.2 Full-word, logical operators.

Add the ability to implement boolean operations on full words, and the addition of an XOR operator. Requiring conversion to and from SET to the original (via variant records or some kind of type conversion), type transfer to and

from boolean to the original, and the ability to use AND, OR, and NOT on integers are all possibilities.

Recommendation:

We recommend that boolean operations on non-boolean operands be done only with functions, which may be pre-declared.

Discussion:

Infix operators were rejected because of the introduction of insecurities in the type-checking of the operands.

[Editorial Comment: The Pascal language is a high level language and thus has no need for the concept of a word; sets do not, in programming or in mathematics, have Boolean operations; the exclusive or operator is present in the language in the form of inequality of Booleans; and and, or, and not do not have meaningful application to numbers, either integer or real.]

*****

8.3 Allow the specification of field layout in a packed record, to facilitate using Pascal structures to manipulate low-level objects.

Recommendation:

The Workshop recommended no extension in this area. See section D for a way to obtain the desired effect.

Discussion:

Two differing solutions are offered for consideration:

1) Record specification corresponds to the physical layout, through bit-by-bit packing, i.e. by allocating to each field the exact number of bits required for its specification.

2) Physical allocation is explicitly specified in the record declaration.

The first solution is easier to implement, while the second provides for greater flexibility in allowing independence of logical and physical structures.

*****

8.5 Decide if PROCEDUR is a reserved word, and if functionid is a valid identifier (on compilers with 8 characters of

significance). Should identifiers be significant to an arbitrary length, at least as an option?

Recommendation:

We recommend that PROCEDUR (sic) not be a reserved word.

[Editorial Comment: See further the British Standards Institution draft.]

*****

8.8 Decide upon what characters should be standard substitutes for ^, [, etc.

Discussion:

We refer standard substitution of character sets to the PUG interchange group (see section F, 8.15). This topic requires research in other character sets, which we cannot undertake at this time.

*****

8.9 Define a standard, as far as common options, for compiler directives. The simplest form is the one discussed in the Pascal-6000 implementation, where one-letter options that can be turned on or off. A slightly more complicated, but very useful feature is the ability to set, reset, or pop compiler options.

Recommendation:

We recommend that no effort be made to standardize compiler directives.

Discussion:

1) Widely differing requirements may necessitate widely differing syntaxes.

2) The convention used in the CDC compiler is not to be taken as a standard.

*****

8.10 Add a compiler directive allowing the inclusion of separate source text files, requiring the relaxation of declaration order

under special circumstances e.g. ($Include decs, procs, body).

Recommendation:

An "include" facility is desirable, but we make no recommendation on syntax. See section 8.9.

*****

8.11 Standardize a conditional compilation format, whether by compiler directives or by boolean constants.

Recommendation:

We do not recommend language extensions to support conditional compilation.

Discussion:

1) A simple pre-processor may provide this capability.

2) Compile-time evaluation of boolean expressions will provide conditional code generation, but is inadequate for declarations.

*****

8.12 Allow different forms of comments, eg. a comment by which the compiler ignores the rest of the line.

Recommendation:

We recommend no change to the report's comment convention. The small gain in convenience does not justify the language change.

*****

8.13 Enumerated scalars with designated values.

Recommendation:

No recommendation was made.

Discussion:

Needs for enumerated scalars with funny values can be

handled by using the scalars as indices into a constant array (see section C1, 4.5).

*****

SECTION F RECOMMENDATIONS TO OTHER GROUPS

Several of the topics considered by the Workshop are being (or we recommend should be) considered by other groups in the Pascal community including the International Working Group and the committee preparing a draft ISO standard. In some cases, the Workshop agreed that the final recommendation of those groups should govern action on a given topic. In others support is requested for extensions that are recommended unequivocally.

1.1 Add else clause to case statement.

Extend the case statement to include specification of the action to be taken if the selector expression fails to match any case constant (label) in the statement.

Recommendation:

We recommend the implementation of this extension (as published in Pascal News #13) according to the syntax and semantics adopted by the Working Group.

Discussion:

See section C1.

*****

1.13 Semantics of CASE statement.

Jensen and Wirth [User Manual] states "the CASE statement selects for execution that statement whose label is equal to the current value of the selector; if no such label is listed, the effect is undefined".

Recommendation:

We recommend that "undefined" in this case be interpreted as causing a run-time error.

*****

2.2 Provide type-checking in parameterized procedures and functions.

Redefine the syntax for passing procedures and functions so as to allow type checking and the use of var in the parameter

lists of procedures and functions that are themselves parameters. There are two main approaches: one, to redefine the syntax for a parameter list to reflect the nested parameter list, eg.

```
procedure q (function p (integer): boolean)
```

the other is to allow the declaration of procedure and function types, e.g.

```
type funcparam = function p (i: integer): boolean
```

Recommendation:

This change is desirable to cover a lapse in type checking. This represents an extension to the language. A consensus has not been reached on the method to use.

Discussion:

There are several different methods for solving this problem. The first provides type information inline in the parameter list. This does not work for recursively defined parameter lists, and forces comparison of parameter list types by structure only.

A second method provides another section, similar to const or label, which defines the parameter list form. Although this allows type compatibility checking by name, it introduces a declaration section of a new and different nature.

A third solution allows procedure and function type constructors (see section 2.10 and the example in the question above). Parameter checking can be easily done using name type compatibility. This may open the way for procedure variables to be implemented.

*****

2.3 Add adjustable array parameters.

Recommendation:

This change represents a desirable change to the language. We prefer the form mentioned in section C1 , but defer to the Working Group. See section C1 for discussion.

*****

2.10 Add a procedure type constructor to the language.

Recommendation:

The Workshop recommends that the Working Group consider this as a true extension to the language. Since this extension is mainly offered as a solution to procedural type checking, we do not recommend it by itself.

Discussion:

This change represents a solution to the problem of type checking in procedures. It needs to be examined closely. An example function type declaration is shown below:

```
type funcparam = function p (i: integer): boolean
```

and the declaration of an identifier to be a function of this type would appear as follows:

```
function funcname: funcparam;
  begin
    { function body }
  end;
```

The existing form of function and procedure declarations would remain valid (anonymous type).

This extension leads to, but does not require, a further extension permitting variables but not functions of a procedure type. The value of a variable of procedure type is a procedure or function of the same procedural type, and is defined only during the lifetime of the block containing the declaration of the procedure or function.

*****

3.22 Should NEW of a variant record assign the tag?

Recommendation and discussion:

As can be seen from the two contradictory opinions which follow, the Workshop was divided on the meaning of the current definition of NEW. If, as it says in the latest(?) edition of Jensen and Wirth, it should not be assumed that NEW does anything to these fields, no extension is recommended. In spite of the usefulness of such a feature it should be discarded because it gives rise to portability problems that cannot be detected at compile time if not all installations use the same definitions. It is a totally unnecessary change.

However, the case can be made that NEW must create the new record with values in the tag fields. Section 10.1.2 in Jensen and Wirth [Report] can be interpreted as defining the NEW variable to be created such that the tag fields possess the

given values ab initio. This can be considered by analogy to constants: a constant has a value immediately upon its creation. Thus, the statement in Jensen and Wirth that "This does not imply assignment to the tag fields", is not contradicted. This interpretation is supported by the warning of the next paragraph, forbidding any meaningful assignment to a tag field. Assume that the tag fields are initially undefined: then, the variants are also undefined. The warning requires that the variable must not change its variant - that is, the variant must not be changed from undefined to something alse, as a consequence of an assignment to a tag field. Therefore, it is required that NEW create the record with its tag fields defined.

We recommend that the British Standards Institution resolve this question.

*****

4.1 Allow an <unpacked structured type> to be a type identifier, thereby allowing something like

b = packed a;

Recommendation:

The amount of effort to implement outweighs the advantages.

Discussion:

See section E.

*****

4.5 Structured constants.

Add the ability to declare structured constants, for tables and other structured data that currently must be variables and have code space and execution time associated with their definition.

Recommendation:

The Workshop decided that structured constants should be accepted. We defer to the Working Group for a final recommendation on the syntax and recommend the syntax shown in section C1.

Discussion:

See section C1.

*****

4.5b Should the required ordering of const, type, var and procedure,function declarations be relaxed?

Recommendation:

No consensus was reached on an explicit proposal, but we recommend that the Working Group consider this area.

Discussion:

See section G.

*****

4.8 What rules for type compatibility should be standard ?

Explicitly define the rules for type compatibility, especially for structured types. There exist currently two major definitions, and the two are, of course, incompatible.

Discussion:

The Workshop had lots of discussion on the merits and failing of "name" compatibility. It was generally acknowledged that this topic is too full of minute and subtle details for the Workshop, however a leaning towards some form of "name" compatibility was obviously there. Precise definitions will have to come from the Working Group and/or from the British Standards Institution committee.

Some of the problems discused were:
anonymous types as in A : array ...
the difference between use in assignment and in expressions
the type of a set constructor as in " [10, 11, 12] "
the type of a dynamic array parameter, if they are ever allowed,
is type declaration reflexive i.e.

"type B=A; C=A { B =? C }"

*****

4.9 Add and provide the language support of the type Complex, for scientific application people.

Recommendation:

1) Complex arithmetic should be considered a conventionalized extension to Pascal.

2) The type COMPLEX should be added as a predefined simple type (as opposed to a structured type like record).

3)The standard infix arithmetic operators +, -, *, / should be provided by overloading the existing operators.Further, the infix operators such as <>,= etc need to be extended to apply to complex values.

4)The full set of standard real mathematical functions (i.e., ABS, SQR, SIN, COS, ARCTAN, EXP, LN, SQRT, and POWER (as defined in 3.4)) should be extended to allow complex arguments and to produce complex results with the exception that ABS should still produce a real result. The following new predefined functions should be considered part of complex arithmetic for Pascal:

function arg(z: complex):real	returns argument of z
functioncmplx(x,y: real):complex	returns x+y*sqrt(-1)
functionre(z:complex): real	returns real part of z
functionim(z:complex): real	returns imaginary part of z

The results of all functions are to agree with the ANSI Fortran standard as this standard reflects accepted practices in computational numerical methods.

5) There was considerable debate over the form that complex constants should take. As two members of the Working Group are continuing work on this area, a final decision should be left until the results of their efforts are known. It was recommended that an explicit form of constant not be introduced into the language, but that complex constants be constructed with the function cmplx. It was also realised that an adequate definition for complex constants may be able to be formulated once the final form of structured constants was decided upon.

6) No direct input or output of complex values is supported in this extension.

Discussion:

Complex arithmetic in Pascal should only be added so as not to impact the language as it is currently defined (and as formalised by the ISO standard). To achieve the status of a "pure extension", it must be possible to provide a mechanism for moving a program that uses complex arithmetic features in a particular implementation to another system where the extension has not been installed, by:

1) Providing a set of Pascal procedures/functions that may be added to the program to be compiled and executed on the new machine.

2) Providing a pre-processor that accepts as input the program using complex arithmetic and produces as output an equivalent program in Standard Pascal.

In fact, a combination of the above approach may be required.

Record or Scalar?

There have been a number of proposals for complex arithmetic in Pascal which advocate the use of a predeclared record type 'complex' as follows:

```
type complex = record
                 re,im: real
               end
```

The problem with the above proposal is that functions are then required to return a structured type, a facility not currently available in the language.

A second proposal is to regard the type complex as a new predefined scalar type, with the underlying structure transparent. It then becomes necessary to define a mechanism for the specification of complex constants. Two possibilities were considered:

1) Use the letter J to indicate the real and complex components in a complex number e.g 4J5. The problem with this notation is illustrated by the following:

```
5.0J-6.0
```

which really represents the complex value 5.0 - 6.0j. Yet another example along this line is: 2.0-5.0J-6.0. A further problem involves the specification of a negative real part of a number i.e. -5J6. Is this (-5)J6 or -(5J6)?

2) The alternative appears to allow complex values to be constructed only with the function cmplx. The restriction here is that complex constants are now not allowed in the const section of a procedure (in a similar way that ord(x) is not allowed).

This whole area needs some more work, however it is felt that constant construction with the function cmplx will prove to be adequate.

*****

5.1 Define communication between program and operating system through the program heading.

See section C3.

*****

5.13 Should RESET be required before GET and REWRITE before PUT?

Recommendation:

We recommend this item as a clarification to the language.

Discussion:

See section C3.

*****

8.4 Upper/lower case and break characters.

See section C1.

*****

8.7 Character set enumeration.

See section C1.

*****

8.15 Interchange standards.

Recommendation:

We recommend that a longer term group be formed within PUG to consider interchange standards such as those in 8.4, 8.5, 8.14, and similar topics such as a minimum character set and minimum (maximum) line length.

[Editorial Comment: This group has been organized. See Pascal New #13.]

*****

The Workshop recognized several areas in which extensions might be desirable. This section contains suggestions for experiments in those areas.

1.3 Structured error-handling and recovery.

Add facilities for structured error-handling and recovery, for both hardware (like floating point overflow) and user-defined (like symbol-table full) errors and exceptions. One of three alternatives is a highly modified form of recovery blocks, e.g.

```
try   <statement list 1>
exception   <statement list 2>
end
```

where failure inside statement list 1 causes control to be transfered to statement list 2. Failure in 2 causes failure of the construct, and control is passed to the enclosing (run-time) recovery block.

Another proposal is exception procedures that get called when errors occur, and can be nested, and redefined in inner scopes.

```
exception divide_by_zero;
  begin
    writeln('divide by zero occured');
  end;
```

A third is EPILOGUE PROCEDUREs, as in Burroughs DCALGOL, which always get called at the end of a block, even if errors have occurred.

Recommendation:

No attempts should be made to define a "conventionalized extension" until there is some base of experience. There was some discussion of global error handling for domain and range failure on predefined functions, but no agreement was reached on any aspect of this topic.

*****

3.3 Type Transfer Functions.

Extend type-transfer functions, at least for scalars (excluding reals) and pointers (as an implementation-dependent

feature) to provide an inverse ord function, thereby reducing the necessity of subverting type-checking by the use of variant records with no tag fields. e.g.

```
c := color(i)
```

Also specify that enumerated types start at 0, and allow ord(pointer).

Recommendation:

There are obvious flaws in the following suggestion, e.g. the mixing of types and functions, but we suggest the following form for experimentation:

```
a := color(I);
```

where color is a scalar type and the integer I is coerced to be of type color.

*****

4.5b Should the required ordering of const, type, var and procedure function declarations be relaxed ?

Recommendation:

No consensus was reached on an explicit proposal.

Discussion:

This represents a very small change to most compilers and should make it possible to write more modular and hence understandable programs. This also allows structured constants (4.5a) to appear in more meaningful places. Most of the arguments about this proposal centered around variations of the following Pascal text:

```
type P = ^B;
procedure Q;
  var QP : P;
  begin
    {to use QP here requires it's full type to be known}
  end;
type B = ...
```

The Workshop agreed that this degree of forward referencing is not acceptable; that such a forward reference must be resolvable by the end of the type declaration part that it appears in (i.e. before the procedure in this example).

Section D shows a method for grouping declarations which does not require relaxing order requirements.

*****

4.6 Parameterized type string.

Add the parameterized type string, which has a dynamic length associated with it, plus the necessary language support. This dynamic length is used to determine the validity of indexing, assignment, etc., hence it is difficult to implement as a record in standard Pascal. In addition, methods are needed for insertion, deletion, substring extraction, concatanation of strings, and access to the dynamic length.

```
var str1: string[1..80], str2: string[0..19];
...
str1 :='abcd';
str2 := str1 & str2;
str1 := str2[3..5];
```

Recommendation:

We do not recommend including the parameterized type string as a conventionalized extension, as there are still too many unknowns. However, for those who do want to implement it now, the UCSD proposal is one method, and includes the following...

1) the predefined parameterized type string, with the form:

```
var str : string [MaxStringLength];
```

two variables of type string are always compatible as far as type checking is concerned, irrespective of their maxlen.

2) assignment between strings, the assignment:

```
str1 := str2;
```

is legal if the dynamic length of str2 is less than or equal to the maximum length of str1.

3) subscripting of strings, i.e. str[1] is the first character of the string. Notice that string subscripts start at one. Range-checking is done against the dynamic length.

4) a collection of predefined procedures and functions: position, length, insert, delete, concatenate, copy. Concatenate and copy can either be defined as functions of two string arguments returning a string or as procedures of three. Functions can cause run-time inefficiencies, procedures can cause difficulties in use.

5) allow strings to be read and written

6) allow string constants 'quoted strings'.

There is an alternate proposal by Arthur Sale which is based upon the existing structuring abstraction of files, with the addition of random-access characteristics. This might allow the capabilities desired but require less conceptual additions to the language.

[Editorial Comment: See also the article by Arthur Sale on Strings and the Sequence Abstraction in Pascal to appear in Software Practice and Experience early in 1979. A follow-up article by Judy Bishop is expected to appear also.]

*****

5.4 Add support for random access files.

The I/O Subgroup has concluded that some form of "random access" I/O mechanism would be a valuable extension to Pascal. During our deliberations, however, it became clear that there are several problems to be overcome in the definition and implementation of this capability. These are:

1) define exactly what is and is not meant by the term "random access"

2) determine the extent to which our definition can be straight-forwardly implemented within the scope of existing file/operating systems

3) determine how to treat the file buffer variable

4) assure that the chosen definition(s) fit closely the current standard

The latter is a motherhood statement, but it reflects our overall desire to maintain a language that can reasonably be called Pascal, rather than a desire to define a new language.

Our working idea of a random access file is as follows. It is a sequence of components, each of which has an "index" or position number, with the first being 1. Within the file, any component may be positioned under the "read/write head" by doing a SEEK to that component. Both "reading" and "writing" are defined anywhere within the bounds of the file (see below), regardless of the state of EOF. The exact mechanisms for these operations are defined below.

Each component has a value, although there may be no way of predicting what that value is. A random access file, like a sequential file, must be extended one component at a time (see below): there is no way, in our concept, of writing a random-access file with holes in it. There may be system-dependent problems arising from an attempt to read such a file which has

not been written by a Pascal program. For example, some systems require random access files to be highly formatted and will have a way of determining that certain components (records) are "undefined". In this case, there may be a system-enforced abort when reading such a record. Such features are beyond our scope at this point and are left to the implementor and the documentator.

We have isolated two basic suggestions for random access I/O, differentiated by the presence of a "file buffer variable". These are described below. The Workshop felt that there is not enough experience with specific implementations to justify a recommendation to extend the language with this capability. Rather, the following schemes are suggested as experimental. Our intent is to track implementations of each to determine both their utility and suitability.

In each case, there is no clear agreement on a choice of identifiers. In particular, while it is agreed that files to be used as random access should be specially declared, how this is done is not settled.

Random Access I/O, Version 1

This version of the random access file (RAF) capability has two alternative forms. The distinction is made on the treatment of the file buffer variable. In either case, some problems do exist. These will be discussed below.

Declaration: Each RAF will be declared as follows:

indexed file of <typeid>

Note in particular that "text files" cannot be treated as random access (although "file of char", of course, can). This decision was made due to the very system-dependent nature of "text files": they are often record-oriented files, with variable length records, and of no certain (between systems) format.

As mentioned, there is no fixed agreement on terminology. Most of us disliked the term "indexed" for two reasons: it smacks of "indexed sequential" files, which are not being implemented; and it might conflict internally with many compilers which use the term "indexed" to describe variable access. Other suggestions are:

replace "indexed" with "random"; "randomaccess"; "random" "access" (two keywords); the same, with "direct" in place of "random"; or "relative"

replace "indexed file" with "direct access store" or some variant thereof, the idea being to emphasize the distinction between the two I/O concepts

The reason for requiring a special declaration is that some systems need to know that a file is to be treated as random access, either because such files differ physically from

sequential files or because different system routines must be used to process them.

Operations:

The following operations are the only ones defined for RAFs.

RESET: open for input/output (and SEEK to position 1)
REWRITE: create an empty file (and SEEK to position 1)
SEEK: change the current position in the file
GET: see below
PUT: see below
LENGTH: return the number of components in the file
EOF: end-of-file predicate (see below)

Note that the normal meaning of RESET and REWRITE are somewhat blurred in the context of random access files. Here, RESET is used to open a pre-existing file (either from the point of view of the external world or from that of the program; the latter does not make a lot of sense). REWRITE is used to create a RAF which is initially empty. After components have been added to the file, of course, these may be accessed as if a RESET had been (but need not be) done.

The predicate (boolean function) EOF is true of the indexed file f, iff the position of the file (i.e., as of the last SEEK) is >= LENGTH(f)+1. The effect of SEEKing to a position beyond LENGTH(f)+1 is explicitly undefined, and is left for the implementor to define. Specific implementations should indicate what the effect will be.

Version I, Variation 1

This implementation treats the file buffer variable (f^) as a "window" into the file. In particular, it is always defined, and any modifications to the buffer variable imply an immediate (i.e., without any further action) corresponding effect on the file.

In this variation the operations above are defined as follows.

SEEK: defines the value of the buffer variable as that of the file component at that position

GET: not defined

PUT: valid only when EOF is TRUE and the current position is at the end of the file; this is the only way to extend the file. Note that, when EOF is TRUE, just modifying the buffer variable does not have an effect on the file. PUT must be done to extend.

Version I, Variation 2

This variation differs from the first in that the SEEK does

not define the file buffer variable. Specifically, the only effect of the SEEK is to inform the runtime support which component is to be involved in the next I/O. The GET procedure must be invoked to give the buffer variable the value of the now-current component.

The operations are defined as follows.

SEEK: "positions" the file, but does not affect the file or buffer variable contents.

GET: gives the buffer variable the contents of the current component; advances the position by one

PUT: puts in the file, at the current position, the current value of the buffer variable; advances the position by one

Potential Problems with Version I

There are problems with these treatments which deserve a close look. One of the major ones, and one which can be a problem for standard Pascal, is the treatment of the buffer variable when used as a VAR parameter. It is likely that the standard is going to disallow this usage, as an example of aliasing, but if that is not the case, the choices are run-time checking and arbitrary decision as to treatment.

[Editorial Comment: Consider the following.

```
var f: indexed file of t;

procedure p(var b: t);
begin
   b := expression of type t;
end;

begin    p(f^)       end.
```

This fragment exhibits no aliasing, yet still causes problems when assignment f^ := expr is interpreted as changing the file ('doing I/O' if you wish). It requires no less than either a thunk or else automatic updating of the file after return from p. I don't think this can be avoided sl long as an assignment to the buffer variable is interpreted as changing the file itself. This is apparently the case for both Variations of Version I. (Variation 2 does not explicitly state what happens, but this effect is not mentioned as a difference from Variation 1.)]

Another major implementation problem, which does not arise for sequential files, comes from the use of a "sliding window" buffer. This is an internal buffer, managed by the runtime support, which may contain more than one component of the file. For random access files of Variation 2, the following partial program shows up a problem if the buffer variable is implemented as a pointer into an internal buffer which can contain more than one component:

```
f^ := x;
put(f);
f^:=y;
seek(f,current + abigbunch);
```

In this example, the first assignment represents a legitimate modification to the file (by virtue of the PUT); however, the second does not. Thus, the efficiency of internal buffering is lost, or the file buffer variable is turned into another buffer.

Both variations are possibly in conflict with the restriction recommended in 5.13 (sections C and ~).

Random Access I/O, Version II

This suggests a 'minimal' direct access file capability. Only four operations are defined for 'indexed files'. The operations are defined syntactically and semantically by four Pascal procedures and functions. (All four conform to the current standard except for the use of a predefined procedure 'abort the program'.)

If these operations are recommended as a conventionalized extension, we suggest that an implementation be considered as conforming to the extension if the provided operations have identical semanatics and syntax.

1. Summary of defined operations.

Given f and indexed file of T, the only operations allowed on f are:

file_length(f)	to return the number of components in the file.
make_empty(f)	to cause f to have length zero.
read_indexed(f,p,v)	to read into v the value of the p-th component of f. The type of v must be T (the component type).
write_indexed(f,p,e)	to write into the p-th component of f the value of the expression e. The type of e must be T (the component type). A new component is appended if p = file_length(f)+1.

2. Formal definitions.

Clearly the procedures presented below do not provide an efficient implementation. No attempt has been made to optimize the source code. Nonetheless, it is fairly obvious that a number of such optimizations are feasible.

```
type ftype = {indexed} file of T;

function file_length(var f:ftype) : integer;
   {file_length(f) is the number of components in file f. }
   var count: integer;
begin
  count := 0;
  reset(f);
  while not eof(f) do
    begin
      count := count + 1;
      get(f)
    end;
  file_length := count
end {file_length};

procedure make_empty(var f:ftype);
  { make_empty(f) causes f to be the empty file. }
begin
  rewrite(f)
end {make_empty} ;

procedure read_indexed(var f: ftype;  p: integer;  var v: T);
  { read_indexed(f,p,v) assigns to variable v the value of
    the p-th component of f }
  var count: integer;
begin
  if (p <1) or (p > file_length(f)) then
    { component index is out of valid range }
    abort_the_program
  else
    begin
      { skip the first p-1 components of f }
      reset(f);
      for count := 1 to p-1 do get(f);
      { return the value of the p-th component }
      v:= f^
    end
 end {read_indexed} ;
```

```
procedure write_indexed(var f: ftype;  p: integer;  e: T);
    { write_indexed(f,p,e) assigns to the p-th component of
      f the value of expression e. }
    var count: integer;
        f2: ftype;

begin
  if (p < 1) or (p > file_length(f) + 1) then
    { component index is out of valid range }
    abort_the_program
  else
    begin
      { copy the first p-1 components of f into f2 }
      reset(f);   rewrite(f2);
      for count := 1 to p-1 do
        begin
          f2^ := f^;
          get(f);
          put(f2)
        end;
      { assign e to the p-th component of f2, and skip the p-th
        component of f. }
      f2^ := e;  put (f2);
      if not eof(f) then get(f);
      { copy any components after the p-th from f into f2 }
      while not eof(f) do
        begin
          f2^ := f^;
          get(f);
          put(f2)
        end;
      { copy all of f2 back into f }
      reset(f2);  rewrite(f);
      while not eof(f2) do
        begin
          f^ := f2^;
          get(f2);
          put(f)
        end
    end
end  {write_indexed};
```

*****

6.1 Encapsulation.

Add encapsulation capabilities for constants, types, variables, and procedures and functions.

Recommendation and discussion:

We conclude that it is premature to recommend as a conventionalized extension, any specific scheme for supporting the facilities generally referred to as "encapsulation", and angled up with separate compilation and other topics.

[Note: We have however, in 6.2, made recommendation in the area of external compilation which is a pre-requisite for encapsulation.]

However, a great deal of progress in this area has been achieved by several independent groups, three of whom are represented among us.

We recommend, therefore, that these three implementations be considered as informal experiments to determine the relative merits or pitfalls of the various approaches. After sufficient usage ("Beta site testing") of these implementations has been recorded, we will be in a better position to decide, with some confidence, which approach (if any) should be recommended.

In this preliminary report, we will not detail the individual proposals. Rather, we will try to summarize the aspects which unite and differentiate them. For this purpose, we have adopted the following informal definitions:

Object = constant, type, variable, procedure, or function
Component = Collection of objects
Program = Runnable collection of components

The three candidates for experimentation are proposals from: 1) National Semiconductor Corp., 2) Tektronix, Inc., and 3) the UCSD Pascal Project. We list first the functional capabilities that the proposals share, and then comment on their differences.

Shared Functional Capabilities of Proposed Experiments in Separable Program Components

1. Selective propagation of information made available by a component, thus permitting the hiding of details of data and algorithms.

2. Separate compilation of components.

3. Compile-time type security across (even separately compiled) components (with no linker changes required).

4. Mandatory single point definition of shared objects.

5. Long-lived variables of less-than-global scope (providing the benefits of "own" variables without the difficulties).

6. Renaming facility upon importation and exportation of objects.

Differences:

The NSC and UCSD proposals are similar in concept but differ in certain details. The Tektronix proposal embodies a philosophy that is distinctly different from the others. This has led to many differences of detail.

Among these differences are the answers to the questions:

1. Should the unit of encapsulation be co-existensive with the unit of compilation?

2. To what extent should a renaming capability be provided?

3. Is it necessary, possible, and/or reasonable to provide automatically invoked data structure initialization?

4. How can access to non-Pascal components be provided?

6.2 Specify the syntax for separate compilation of Pascal modules and procedures, and rules specifying what must be recompiled when a procedure is changed.

Recommendation and discussion:

The following is an attempt to represent the agreement of the discussion group on type-secure external compilation. No new abstraction is being provided. This proposal does not address the wider problem of encapsulation (see 6.1). In particular the following interesting and possibly desirable extensions are not considered:

1. Abstract data types

2. Information hiding (in particular the structure of types)

3. Use of languages other than Pascal

4. Renaming of identifiers

However, this proposal does not rule out these extensions and could serve as a basis for further experiments.

[Editorial Comment: The accuracy of this representation has been disputed.]

Type Secure External Compilation

This is a proposal for type secure external compilation in Pascal. The proposal was developed from a variety of module proposals, using the following criteria:

1. Must allow external compilation of all Pascal objects, including constants, types, variables, procedures, and functions.

2. Must support the development of libraries as well as the piecewise development of large programs.

3. Must maintain type security across separately compiled objects.

4. Must use existing linkers, as provided by most operating systems.

5. Must involve minimal extensions to Pascal.

6. The unit of encapsulation is the unit of compilation. This utilises the encapsulation normally provided by Linkers.

7. Must provide a base for experimentation with other modularization methods.

6.2.1 The proposal

The basic unit of modularity is the "compilation unit.' A conpilation unit is simply that Pascal code which is compiled at one time by the compiler. It contains the information necessary to control communication with other compilation units.

<compilation unit> ::= <program> | <unit>

<unit> ::= <unit heading> <unit body> END.

<unit heading> ::= UNIT <unit identifier>;

<unit identifier> ::= <identifier>

The "unit" is a compilation unit which defines objects for use by other compilation units. The "unit identifier" is a name which is used by other compilation units to gain access to these defined objects. The unit identifier is defined both in the outer scope of the unit and in the "external scope" of all compiled objects. Different operating environments will unfortunately have different definitions for this "external scope" and the implementer will usually be forced to follow these definitions.

A unit consists of an "interface part", which contains those objects which are to be made available for reference by other compilation units, and an "implementation part", which is not made available (private).

Some Pascal implementations may allow separate compilation of the interface part and the implementation part of a unit, in which case the unit identifier serves to relate the two parts.

```
<unit body> ::= <interface part> | <implementation part> |
                <interface part <implementation part>

<interface part> ::= INTERFACE
                     <use clause>
                     <constant definiton part>
                     <type definition part>
                     <variable definition part>
                     <procedure and function heading part>
```

All objects defined in the interface part are made available for reference by (exported) to other compilation units which use this unit.

```
<use clause> ::= USES <unit identifier>
                 {, <unit identifier>} : | <empty>
```

When a unit identifier is named in a "use clause", those objects defined in the interface part of that unit are made available for reference in the scope of that use clause. This is said to "import" the objects into that scope. We also speak loosely of importing the unit, which means importing all the objects in the interface part of the unit. Imported objects become defined in the order in which they are mentioned in the use clause.

In the case of a use clause in the interface part of a unit, these objects are not automatically made available for reference by compilation units which import that unit. Since the objects in such a use clause will normally be required in ..e definition of objects in the interface part, they must be imported into any scope which imports that unit. Normal Pascal rules about definition before use apply, so they must be imported prior to any unit which imports them.

```
<procedure and function heading part> ::=
                {<procedure or function heading>}

<procedure or function heading> ::= <procedure heading> |
                                    <function heading>
```

The bodies of procedures and function declared in the "procedure and function heading" part must be provided in the implementation part for that unit.

The <implementation part> contains the bodies of all procedures defined in the interface, and anyother objects private to the unit.

```
<implementation  part> ::= IMPLEMENTATION
                           <use clause>
                           <constant definition  part>
```

<type definition part>
<variable declaration part>
<procedure and function
declaration part>

The parameter list and result type of any procedures or functions defined in the interface part are not repeated in the implementation part. This is similar to the current treatment of "FORWARD" procedures and functions within Pascal. All objects which are defined or imported into the interface part are available within the implementation part for that unit.

The objects imported into the implementation part need not be imported into scopes which import the unit. This allows implicit inclusion of units used only in the implementation. This also allows mutual recursion between units, though some Pascal implementation may disallow this.

The following modification to existing Pascal syntax allows the importation of objects from externally compiled units into the main program.

<program> ::= <program heading> <use clause> <block>.

Note that a use clause is allowed only in the global scope of the program. This avoids semantic problems when objects are imported into nested scopes, possibly with some parts hidden by other declarations. It is also consistent with the conceptual "external scope" in which unit identifiers are defined.

For each program there is only one instance of any unit, even if that unit is imported into many other compilation units. Unit global variables, which are declared either in the interface part or in the main scope of the implementation part, have the same lifetime as variables within the global scope of the main program. The values of these variables are retained between calls to procedures within that unit.

There is no implicit initialization of units. Programmers may program such initialization as a procedure to be explicitly called.

6.2.2 An Example

We illustrate the use of units with the ubiquitous stack, but with some elaboration to illustrate more of the features.

{The following unit defines general types}

```
unit symbol_types; interface

        type
          sym = (a_sym, b_sym, ...);
          {other type definitions}

      end. {symbol_types}
      {The following unit defines a stack of symbols}

      unit symbol_stack;
        interface

          uses
            symbol_types; {used in the interface}

          procedure push_sym(s:sym);
          procedure pop_sym;
          function top_sym: sym;
          procedure init_sym;

        implementation

          uses
            error_handler; {privateimportation}

          const
            stack_max = 100;

          var
            cur_top:  0..stack_max;
            stack: array [1..stack_max] of sym;
          procedure push_sym;  {note:parms not repeated}
            begin
              if cur_top < stack_max then
              begin
                cur_top := cur_top + 1;
                stack[cur_top] := s;
              end
              else
                bomb_program('Symbol stack overflow');
                {A procedure form error_handler}
              end;  {push_sym}
          procedure pop_sym;
            begin
              if cur_top > 0 then
                cur_top := cur_top -1
              else
                bomb_program('Symbol stack underflow');
              end;  {pop_sym}
```

```
    function top_sym; {note: type not repeated}
      begin
        if cur_top > 0 then
          top_sym := stack[cur_top]
        else
          bomb_program('Sumbol stack empty');
        end;
    procedure init_sym;
      begin
        cur_top := 0;
      end;

end {symbol_stack}.
{The following program uses the symbol stack}

program use_stack(output);

  uses
    symbol_types, {needed before symbol_stack}
    symbol_stack; {import the stack}

  {use of the above}

end {use_stack}
```

The first unit (symbol_types) defines types which are expected to see wide use throughout the program being implemented.

The second unit (symbol_stack) defines a more specialized object which uses a type defined in symbol_types. In addition, within the implementation, it uses a procedure defined within the unit error_handler.

The main program imports both symbol_types and symbol_stack, and may freely use identifiers defined within them. Since a type from symbol_types is required within the interface of symbol_stack, symbol_types must be imported prior to symbol_stack.

Note that error_handler may be used by many units throughout the program, but there is still only one instance of the error_handler unit.

6.2.3 Rationale

This section gives the rationale behind specific decisions embodied in the proposal.

6.2.3.1 Environments

In Pascal News #12 Rich LeBlanc proposed the compilation of "environments". This was rejected as an approach because it did not lend itself to the construction of libraries, only

small parts of which will be used by any single program. It seems very well suited for developing large programs.

6.2.3.2 Separate Interface and Implementation

The interface and the implementation were separated for two reasons.

1. The compiler can get all of the interface data it requires by a simple textual scan of the interface part, thus requiring minimum change to existing compilers.

2. With separate interface and implementation, it is possible to change the implementation without affecting those programs which use the interface. This avoids some recompilation. In addition, an implementation may allow separate compilation of the interface and implementation for a unit.

6.2.3.3 No implicit Importation

If a unit is imported into an interface part, it is not implicitly imported into units which import that interface. To allow this has at least two consequences which can complicate the compiler.

1. A compiler may have to do an arbitrarily deep file search to get data on all implicitly imported units. Alternative implementations are certainly possible, but also add compilation.

2. If a unit is imported into a scope twice, by direct or indirect paths, the compiler must detect this and avoid duplicate entries into the symbol table. In addition, it must check that all imports are the same type. This becomes even more complicated if imports are allowed in nested scopes, or with renaming.

6.2.3.4 All Items in the Interface are Imported

This provides minimum change to existing compilers. Implementations wishing to extend this way add explicit export lists to the interface. Hiding the structure of exported types, as is done in Modula, was considered to be outside the scope of external compilation and to be a major change form Pascal. It also requires additional notation to specify when the structure is or is not to be exported.

6.2.3.5 Require Prior Importation of Interface Imports

This restriction is included to allow compilers to have the data which they need to parse the interface. It is consistent with the general philosophy of Pascal, which requires

definition before use.

6.2.3.6 Imports Allowed in the Implementation

This was included to allow implicit nesting and modular libraries. It does present possible difficulties in the case of mutually referencing units, and implementations may forbid this.

6.2.3.7 Importation Allowed Only in Outer Scopes

This avoids problems with part of the definitions needed by a unit being hidden by intervening declarations. It is also consistent with existing linkers, which do not usually have the concept of "scope".

6.3.2.8 Single Instance of a Unit

This is basically the issue of abstract data types. If multiple instances of a unit are allowed, each instance can be considered an instance of an abstract data type defined by that unit. This was considered to be outside the scope of external compilation. If a user wants to implement abstract data types, he can do so by defining a type for the storage structure for that data type, allowing tha user to define variables of that type, and passing that type as a parameter to routines of the unit. See the discussion in the Modula papers for more data.

6.3.2.9 No Initialization Part

If a unit is imported into multiple scopes, such as a main program and an implementation part of another unit, some means is needed to avoid multiple initializations. Since initialization is easily programmed by the user as procedure, no initialization part is provided.

6.2.4 Implementation Notes

These notes describe a trial implementation on a Univac 1110. The Univac linker allows 12 character alphanumeric symbolic references between compilation units. The filing system makes a file search form compiler relatively simple.

The general approach is to store the interface as a symbolic file, generated by the compiler as a side-effect of compilation. A unique reference identifier is generated for each compilation unit, and this is used at compile time and link to provide diagnostics.

1. Storage for variables in the interface and implementation is allocated at compile time. The unit dentifier is externally defined as the address of the start of

this block of storage.

2. Generate a "check identifier" for the unit. The date and time of compilation are expressed as the number of seconds since 00:00,1 Jan 78, then encoded base 32. This results in 6 alpnumeric characters. This result is appended to the unit identifier (up to 6 characters) so that the linker messages will make sense to the user.

3. Generate an external definition for this check identifier within the object module produced for the unit being compiled. The value of this definition is not important. Any unit of program which imports this unit will generate an external reference to this same check identifier. If the unit has been recompiled since the importing unit, the linker will fail. It would be possible to change this check identifier only when the interface part is changed, which avoid some recompilation. This is not proposed for the initial implementation.

4. Generate an external definition for each exported routine, the same conventions currently used for external routines is used.

5. When compiling a unit, generate an attribute file which can be located by the compiler from the unit identifier for that unit. This file is a textfile built according to the following syntax.

<attribute file> ::= <unit heading> <interface part> end.

change:

<unit identifier> ::= <identifier> [<check identifier>]

<check identifier> ::= as described above

This description file contains the interface part for the unit, augumented by the check identifiers for this unit and all units imported into the interface part.

6. The definitions and declarations in the interface are treated as though they are in the outer scope of the implementation part, and the procedure and functions are treated as "forward" declarations. The implementation part is then compiled as usual.

7. The compiler maintains a list internally of unit identifiers and their corresponding check identifiers. When the compiler encounters a use clause in a program or implementation, it does the following for each unit identifier contained therein:

1) Locate the attribute file for that unit.

2) Parse that attribute file with a slight modification to the normal parser.

3) The unit idenitfier and check identifier are compared with the internal list, and entered if unique. An external reference to the check identifier is compiled in the object module.

4) For all units imported into the interface, their unit identifiers and check identifiers are checked against the internal list, and must match identically.

5) All declarations are entered into the symbol table normally, with procedures and function declared as external.

7.1. Concurrency

Add concurrent process facilities, ala Concurrent Pascal, Modula (and 7.2 The addition of interrupt handling facilities).

7.1.1 Introduction

It is the consensus of the workshop that the concurrency and interrupt handling topics be recommended for experimental implementation. The goal of this paper is to find minimal extensions to Pascal which will permit experimentation with concurrency and/or interrupt handlers that can be coded in standard Pascal (with the exception of those specific features which are needed for concurrency or interrrupt handling). The specific features in this area of application should be expressed as nearly as possible in the spirit of Pascal.

In the process of this group's sessions we have heard reports on the mechanisms for expressing concurrency in several higher order languages, including Concurrent Pascal and MODULA. We have found that it is impossible to settle on a standard set of higher level mechanisms for concurrency at this point; that in fact, implementing a full set of such mechanisms can only be implemented cleanly by creating a new language, or switching to one of the already existing concurrent languages, such as those mentioned above. However, we think it may be possible to recommend a set of lower level primitives which can implement any program which can be expressed by any of those programming languages. The cost of such primitives is the lack of user protection present in the higher level languages, yet we encourage experimentation with higher level constructs translated into Pascal with a preprocessor.

It must be observed that there is a severe cost entailed in all these proposals. Whereas a large portion on Pascal has been defined axiomatically [6], these axioms hold only under very strict conditions if a program consists of parallel paths. These conditions cannot be enforced by the low level mechanisms which will be introduced here, and the axioms therefore become theorems to be proven for each statement with respect to every combination of states of all parallel paths.

A second problem is that concurrency has a major impact on many features of Pascal, many of which are difficult to assess. Examples of affected parts are scope rules (what can a descendant process reference, how can you reference variables of a process which has terminated, etc.), and recursion and dynamic allocation of variables (where can you put the space these will take up since the storage for processes cannot be allocated in a stack-like manner). It was from this morass of impossible decisions that we decided that if someone wants higher-level mechanisms for concurrency at this time, he would do well to switch to another language, although he might continue to write his sequential code in Pascal, as most users of Per Brinch Hansen's Concurrent Pascal do.

Thus we became convinced that any specific recommendations for Pascal extensions must restrict themselves to lower level primitives which minimally impact the language, but which depend on quite knowledgeable programmers. Two solutions were proposed. The first proposal (Alternative One) extends the language with operations on semaphores and is presented in section 3. The second proposal (Alternative Two) attempts to specify an even "lower level" set of primitives which would provide greater flexibility for those who need it, and is presented in section 4.

7.1.2 Creating Concurrency

Both proposals use essentially the same mechanism for creating the possibility of concurrent execution. A process is declared along with procedures and functions using the following syntax:

```
<process declaration>  ::= <process heading> <block>

<process heading>  ::= process <identifier> ; |
process <identifier> ( <formal parameter section>
{; <formal parameter section> } );
```

and invoked like a procedure call (Alternative One) or as a function call (Alternative Two).

The syntax of the block is now affected as follows:

```
<block> ::= <label declaration part> <data declaration part>
            <procedure and function declaration part>
            <process declaration part> <statement declaration part>

<process declaration part> ::= {<process declaration>;}
```

Some implementations may wish to impose the following restrictions:

1) All parallel processes must be declared at the highest level. This is to avoid impacting Pascal's lex level structure. It also makes it impossible for a descendant

process to persist beyond the scope in which it was declared.

2) Processes may only be invoked in the outer block.

3) All var parameters must be handled using call by reference.

4) Memory requirements for processes must be calculable at compile time. This requires that if execution of the process could result in recursive procedure execution, the user provides (via pragmat) the maximum possible number of simultaneous activations of a procedure. This allows all memory for processes to be statically declared.

7.1.3 Alternative One: Semaphores

7.1.3.1 Introduction

A process in invoked by a process statement, which has the same syntax as a procedure call.

```
<process statement> ::= <process identifier> |
                        <process identifier> (<actual parameter>{,
                                              actual parameter>})

<process identifier> ::= <identifier>
```

Once parallel processes can be declared and invoked, we need constricts for their safe cooperation:

a) they should be able to synchronize with each other

b) competing for scarce resources has to be resolved; this includes the creation of mutual exclusive access to shared data.

Dijkstra [5] has shown that both type of cooperation can be achieved with two operations (called "P" and "V") on semaphores. We have chosen for the extension of Pascal with semaphores, and not with higher level constructs like monitors, despite the fact that higher level constructs are safer to use. We have done this for the following reasons:

1) At this experimental stage we do not want to impose any higher level mechanism on the user (like monitors, messages, boxes). We would like to remain flexible.

2) Yet it is possible to implement for example monitors as defined in Concurrent Pascal [2] and Modula [3] with semaphores. Thus it is possible to write a preprocessor to translate these higher level constructs into operations on semaphores. Section 3.3 shows how the various monitor definitions can be implemented with semaphores.

3) Although operations on semaphore are at a low level, they are by no means at the lowest "inhibit interrupt" level. The user does not have to write a scheduler, since semaphore operations perform implicit scheduling. However he has some

control over scheduling, as is shown in section 3.4. If the user wants even more freedom, he should use the lower level approach of Alternative Two (see section 4). In section 3.4 it is also explained why a third operation on a semaphore, called "awaited" is desirable.

4) The starting of a process on an interrupt can conveniently be mapped on the semaphore operations. When a process wants itself to be continued by an interrupt, it executes a P operation on a semaphore. The interrupt internally will generate a V operation on the same semaphore. An example is given in section 3.5. When testing, interrupts can be simulated by a V operation performed in the program itself.

5) It appears possible with a minimum extension to Pascal (a predefined type "semaphore" and three predefined procedures and functions) to provide a mechanism for the cooperation between concurrent processes and the mapping of interrupts that will be sufficient for many applications.

7.1.3.2 The Proposed Extensions

The detailed proposal is as follows:

1. There is a predefined type, called "semaphore". A variable of type "semaphore" is essentially a non-negative integer with an associated queue of waiting processes. This queue may be empty. Apart from the predefined procedures and functions described below, a semaphore variable acts as an integer. In particular assignment to an integer value (for initialization) and comparison with an integer value are often used.

2. There is a predefined procedure, called "P", that has as parameter a variable of type semaphore:

```
P (var sema: semaphore) (* sema >= 0 *)
```

with the following definition:

```
if sema > 0 then sema := sema - 1
else put process in queue of sema
```

3. There is a predefined function called "V", that has as parameter a variable of type semaphore:

```
V (var sema: semaphore) (* sema >= 0 *)
```

with the following definition:

```
if sema queue not empty
then dequeue process from sema queue and continue it
else sema := sema + 1
```

4. There is a predefined function, called "awaited", that has as parameter a value of type semaphore:

awaited (sema: semaphore) : boolean

with the following definition:

false if sema queue empty
true otherwise.

5. Association between a specific interrupt (e.g. address, level) and a variable of type semaphore is done via pragmats. No syntax is given here because the interrupt systems vary considerably between different installations. The connection between a process and priority level is also done with pragmats.

Note: The names "P" and "V" are proposed here to emphasize that they represent the same operations as originally proposed in [5], and because the terms "SIGNAL" and "WAIT", which are sometimes used to denote the same operations, have too many other possible connations.

7.1.3.3 Implementation of Monitors with Semaphores

7.1.3.3.1 Comparison of different Monitors Concepts

Hoare defines in [1] the monitor construct. For synchronization he uses the operations 'signal' and 'wait' on a variable of type 'condition'. Any number of processes may be waiting in the queue of a 'condition' variable.

Wirth [3] uses an interface module, with the operations 'send' and 'wait' on variables of type 'signal'. A 'signal' is equivalent to a 'condition' except for one difference in the definition of the 'send'. In Hoare's monitor concept only the 'wait' operation is a singular point in a monitor, that is at any one time any number of processes may be positioned at 'wait' operations but only one process is executing. Wirth [4] specifies that also 'send' operations are singular points, that is at any one time any number of processes may be positioned at 'wait' or 'send' operations, but only one process is executing a monitor procedure.

Brinch Hansen's monitors [2] are essentially the same as Wirth's interface modules. The synchronization operation are called "continue" and "delay" on variables of type "queue". Contrary to Wirth's and Hoare's solution, at most one process may be waiting in a "queue".

Consequently when implementing monitors with semaphores, Hoare's monitors need one extra semaphore in order to ensure that if a process leaves the monitor any process that was held on a 'signal' or 'continue' operation first gets control before other processes can call a monitor procedure.

3.1.1. Implementation of Hoare's Monitor

Hoare presents in [1] an implementation of a monitor, together with the operations 'signal' and 'wait', with (binary) semaphores. There are two semaphores required for the monitor itself:

mutex (initially = 1) for the mutual exclusion of the entire monitor.

urgent (initially = 0) for the continuation of any process held on a 'signal' operation before mutex is released.

In order to be able to test whether there is any process in the queue of 'urgent', we need a counter 'urgentcount', which should be incremented just before the 'signal' operation and decremented after the continuation of the process.

Thus when a process leaves the monitor, that is after a 'wait' operation or at the end of a monitor procedure, we need the statement:

```
if urgentcount > 0 then V(urgent) else V(mutex)
```

For every variable of type 'condition' a separate semaphore is required. As an example:

```
var condvar : condition; condsem : semaphore;
```

A 'wait' operation includes P(condsem). However a 'signal' operation should only perform V(condsem) if any processes are waiting for 'condsem'. Therefore another variable has to be introduced that contains the number of processes waiting in the queue of 'condsem':

```
var condcount : integer;
```

'Condcount' is incremented before every 'wait', and decremented after the continuation of the waiting process.

The implementation of the monitor and the operations 'signal' and 'wait' can now be coded as follows:

```
monitor entry:

  P(mutex)

monitor exit:

  if urgentcount > 0 then V(urgent) else V(mutex)

'wait':

  condcount := condcount + 1;
  if urgentcount > 0 then V(urgent) else V(mutex);
  P(condsem);
  condcount := condcount - 1

'signal':
```

```
  urgentcount := urgentcount + 1;
  if condcount > 0 then begin V(condsem);
                              P(urgent)
                        end;
  urgentcount := urgentcount - 1
```

Notice that in the 'wait' operation the process first releases the monitor to another process by executing V(urgent) or V(mutex) before executing P(condsem) which may be delayed for some time. However this does not matter since condcount was incremented before the process released itself from the monitor.

Example:

assume process A is in the monitor and executes wait(condvar)
process B is trying to enter the monitor and will perform signal(condvar)
and at this time condcount = urgentcount = condsem = mutex = 0

We may get the following dynamic execution pattern:

process A executes wait(condvar) :

```
  condcount := condcount + 1;       (* condcount = 1*)
  V(mutex)                          (* process B enters monitor *)
```

after some time process B executes signal(condvar) :

```
  V(condsem)                    (* condsem = 1; process A continues *)
```

process A finishes the execution of wait(condvar) :

```
  P(condsem)                        (* condsem = 0 *)
  condcount := condcount - 1        (* concount = 0 *)
```

Since 'condcount' is incremented before the mutual exclusion is released by the 'wait' operation, P(condsem) can be performed afterwards. Notice also that replacing 'condsem' and 'condcount' by a general semaphore would not work in this case.

7.1.3.3.2 Implementation of Wirth's interface modules

The implementation if Wirth's interface modules is simpler, since the semaphore 'urgent' and the variable 'urgentcount' can be omitted:

interface module entry:

```
    P(mutex)
```

interface module exit:

```
    V(mutex)
```

'wait':

```
  condcount := condcount + 1;
  V(mutex);
  P(condsem);
  condcount := condcount - 1;

'send':

  if condcount > 0 then
  begin
    V(condsem);
    P(mutex)
  end;
```

7.1.3.3.3 Implementation of Brinch Hansen's Monitor

The implementation of Brinch Hansen's monitor is essentially the same as Wirth's interface module, except that at most only one process can be held in the semaphore 'condsem'. The program should make sure that this is the case. The variable 'condcount' should now be defined as:

```
var condcount:  0..1
```

and run-time checking can be performed.

7.1.3.4 Scheduling strategies

7.1.3.4.1 Scheduling with Signals

The definition of the semaphore in [5] stipulates that a V operation releases that process that is waiting longest (FIFO stategy). In actual situations another strategy may be required, for example one based on priorities. This is presumbly one of the reasons why Brinch Hansen's 'queue' variables can have at maximum only one process waiting, thus hinting to make arbitrary strategies by using arrays of 'queue' variables and performing 'continue' operations according to the desired strategy. However this means that even for implementation of a simple FIFO strategy a sizeable program is required.

Hoare introduces so-called 'scheduled waits' :

```
  condvar.wait(p)
```

in which p denotes the priority of the waiting process in the queue of 'condvar'. The execution of

```
  condvar.signal
```

will wake up the highest priority process waiting on 'condvar'.

Wirth introduces so-called 'ranks' for the same purpose. Assuming :

```
  var s : signal;
```

```
      r : integer
```

the corresponding operations are:

```
wait(s,r)      and
send(s)
```

7.1.3.4.2 Scheduling with Semaphores

Scheduling can be controlled if we introduce for every variable of type 'condition' an array of semaphores.

For example if we want priority scheduling we introduce :

```
const n = .....;     (* highest priority *)
type  priority = 0..n;
var   condsemarray : array [priority] of semaphore;
      i : priority
```

If we want to perform a 'wait' operation while giving the process a priority i in the queue, the operation P(condsem) has to be replaced by :

```
P(condsemarray[i])
```

The 'signal' operation should wake up the process with the highest priority. Therefore we ought to know whether any processes are waiting on the semaphores in 'condsemarray'. Hence the single variable 'condcount' has also to be replaced by an array :

```
var condcountarray : array [priority] of integer
```

Introducing the boolean variable

```
var condfound : boolean
```

we can implement the 'signal' operation as follows :

```
urgentcount := urgentcount + 1;
i := n; condfound := false;
repeat
  condfound := condcountarray[i] > 0;
  i := i - 1
until condfound or i = -1;
if condfound then begin V(condsemarray[i]);
                        P(urgent)
                  end;
urgentcount := urgentcount -1
```

7.1.3.5 Testing whether Process queue is empty

Hoare introduces in [1] another operation on variables of type 'condition' in order to test whether any processes are waiting on a particular 'condition' variable :

condvar.queue

which yields the value true if any process is waiting on 'condvar' and false otherwise.

Wirth does the same in [3] with variables of type 'signal':

awaited(s)

This function is very convenient since there is no other way of finding out whether any processes are waiting apart from keeping counters as we did before.

Let us introduce the following function with a semaphore as parameter :

awaited (sema: semaphore): boolean

which returns the value true if any process is waiting on 'sema', and false otherwise. This would simplify the implementation of the priority scheduling since the array 'condcountarray' is no longer required.

The body of the repeat statement in the implementation of the 'signal' operation now becomes :

```
repeat
  if awaited(condsemarray[i]) then condfound := true;
  i := i - 1;
until condfound or i = -1;
```

7.1.3.6 Interrupts and Semaphores

The following example shows how interrupt routines can be written using semaphores.

```
(* pragmats *)
sem1 [34H]; (*associates sem1 with interrupt location hex 34*)
tty [6];    (*gives process tty priority level6*)
(* end of pragmat *)

program main

var sem1,sem2: semaphore:

process tty (var s: semaphore);

begin
  while true do
  begin
    P(s);
    (* body of interrupt routine *)
  end
end;    (* tty *)
```

```
begin  (* main *)
  sem1 := 0;
  sem2 := 0;     (* initialization of semaphores *)
  .
  .
  tty (sem1);    (* start parallel execution of tty *)
  .
  .
  tty (sem2);    (* start parallel execution of tty *)
  V (sem2);      (* give 'software' interrupt *)
  .
  .
end  (* main *).
```

The invocation of 'tty' concurrently with 'main', however 'tty' will immediately be held on the semaphore 'sem1'. An interrupt, which produces V (sem1), will make the process 'tty' execute one cycle, until it is again held at P (sem1).

At the end of the program 'main' an example is given of another invocation of 'tty' with a different semaphore as parameter. The interrupt is now generated by the statement V (sem2).

7.1.4 Alternative Two: Indivisible Sections

A second suggestion for experimentation is a lower-level set of primitives for controlling concurrency. The extensions provide only facilities for making a section of code execute indivisibly and for selecting the process to be allocated the processor. This very primitive facility was chosen for two reasons.

Using these extensions it is possible to describe in Pascal the higher level operations which are considered as primitives in other models. It is also possible (and of course necessary) to describe the processor scheduling algorithm (thus allowing direct user controlled scheduling which is not possible when the scheduler is buried in the operating system kernel).

Secondly, the primitive operations of most models for concurrency are defined as executing indivisibly. If the extensions to Pascal provide a weaker set of indivisible primitives (ie P and V only), much of their use would be in providing indivisibility for the more complex operations -a possible waste of their power. (Since the more complex operations are still short and provide a queuing mechanism, queuing is not required in the indivisibility mechanism.)

7.1.4.1 The Extensions

The extensions require the following pre-defined types and functions:

```
type
```

```
        process_id = (id1,id2,id3,id4,....);
                     {implementation dependent}

        place = (proc1,proc2,....);
                {implementation dependent}
                {required for multi-processor systems only}

    function whoamI : process_id;

    function wheramI: place; {multi-processor only}

    procedure cause_interrupt(p: place);
       {the interrupt to be caused must be specified
        in a pragmat}
       {multi-processor only}
```

A new statement is also required and is defined in examples given below:

```
    var
        t1,t2: process_id;

    process task(....);
        <procedure body>;


    begin
        {....}
        t1:=task(.....); {process invocation is a function}
        t2:=task(....);  {returning the id of the new process}

        {.....}

        indivisibly do
                  { only one indivisible statement may be
                    executed at one time }
                  { the statement will execute completely
                    without interruption }

                {..... <statement list> }

         switching to t1; {expression of type process_id}
                   { before releasing indivisibility the
                     processor executing this statement is
                     switched to the process indicated }
         {....}

    end.
```

7.1.4.2 Interrupts

The interrupt system can then be modeled by the following Pascal code:

```
var
    what_now : (int1,int2,..,intN);
               { specifies which interrupt has
                 occured }
    interrupt_assignment: array [int1..intN] of
                              process_id;
       { the association between particular interrupts
         and particular processes are established using
         pragmats - the user has no access to the above
         variables}

indivisibly do
    switching to interrupt_assignment[what_now];
```

An interrupt may be simulated by executing equivalent code.

A process responding to an interrupt (as the result it's process_id being in interrupt_assignment [what_now] when the above code was executed {or simulated}) would look like:

```
process task;
  begin
    {.......}
    indivisibly do
        {......}
      switching to <id of some other process>;
    repeat
      {.........}
      indivisibly do
        {......}
        switching to <id of some other process>;
    until false
  end;
```

The process is activated once and executes the loop body once for each interrupt.

If the body of the endless loop which is executed for each interrupt contains a single indivisible statement, a more efficient implementation would place that statement in the interrupt system. This would reduce the number of context swaps.

If for a particular application it was more desirable to have interrupts represented as V operations, either the process switched to could execute a V, or a different pragmat could associate a semaphore with a particular interrupt instead of a process.

7.1.4.3 The Scheduler and Queues

The first step in creating a more useful set of operations is to provide a scheduler. A possible definition is shown below:

```
type
    run_status = set of process_id;
```

```
var
    ready: run_status;
    next, scheduler_id: process_id;

process scheduler(var r:run_status);
    begin
       while true do
         indivisibly do
              {......}
          next:= .....
        switching to next;
    end;

begin
    scheduler_id:=scheduler(ready);
    {.......}
end.
```

In addition the operations defined using this scheduler will use a double ended queue or deque for local storage. The sections which follow assume the following functions and procedures are available: (user written in standard Pascal)

```
procedure empty_deque(var d: deque);
function number_waiting(var d:deque): integer;
procedure on_tail(p: process_id; var d: deque);
procedure on_head(p: process_id; var d; deque);
function off_tail(var d: deque): process_id;
function off_head(var d: deque): process_id;
```

7.1.4.4 Implementation of P and V

We will now write (extended) Pascal procedures for the standard synchronization primitives P and V:

```
type
    semaphore = record
                  count: 0..maxint;
                  holding: deque
                end;

 procedure P(var s: semaphore);
    var next: process_id;
    begin
      indivisibly do
        with s do
          if count > 0
            then begin
                   count:= count-1;
                   next:= whoamI
                 end
            else begin
                   on_tail(whoamI,holding);
                   ready:=ready-[whoamI];
                   next:=scheduler_id
```

```
                    end
          switching to next
        end;

    procedure V(var s: semaphore);
       var next: process_id;
       begin
         indivisibly do
           with s do
             if number_waiting(holding) > 0
               then begin
                      ready:=ready+[off_head(holding)];
                      next:=scheduler_id
                    end
               else begin
                      count:count+1;
                      next:=whoamI
                    end
          switching to next
       end;
```

7.1.4.5 Concurrent Pascal

As a second example of the use of the extended Pascal we give below code for four functions which would be needed to implement Concurrent Pascal using a pre-processor: (two - Delay and Continue - are in Concurrent Pascal and calls to the others which control monitor entry would be generated by the pre-processor)

```
    type
        monitor = record
                    gate: boolean;
                    holding: deque
                  end;

    procedure Enter_monitor(var m: monitor);
       var next: process_id;
       begin
         indivisibly do
           with m do
             if gate
               then begin
                      gate:=false;
                      next:=whoamI
                    end
               else begin
                      on_tail(whoamI,holding);
                      ready:=ready-[whoamI];
                      next:=scheduler_id
                    end
          switching to next
        end;
```

```
procedure Delay(var m: monitor; var d: deque);
   var next: process_id;
   begin
     indivisibly do
       with m do
         begin
           on_tail(whoamI, d);
           ready:=ready-[whoamI];
           if number_waiting(holding) > 0
             then ready:=ready+[off_head(holding)]
             else gate:=true;
           next:=scheduler_id
         end
     switching to next
   end;

 procedure Continue(var m: monitor; var d: deque);
   var next: process_id;
   begin
     indivisibly do
       with m do
         if number_waiting(d) > 0
           then begin
                  ready:=ready+[off_head(d)];
                  next:=scheduler_id
                end
           else begin
                  gate:=true;
                  next:=whoamI
                {process  calling Continue must leave monitor
                 and should not call Exit_monitor}
                end
       switching to next
     end;

procedure Exit_monitor(var m: monitor);
   var next: process_id;
   begin
     indivisibly do
       with m do
         if number_waiting(holding) > 0
           then begin
                  ready:=ready+[off_head(holding)];
                  next:=scheduler_id
                end
           else begin
                  gate:=true;
                  next:=whoamI
                end
     switching to next
   end;
```

7.1.5 References

1. HOARE, C.A.R.
Monitors: An Operating System Structuring Concept
Communications ACM vol. 17, no. 10 (Oct. 1974)

2. BRINCH HANSEN, P.
The Programming Language Concurrent Pascal
IEEE Transactions On Software Engineering vol 1, no. 2
(June 1975)

3. WIRTH, N
MODULA, A Language For Modular Multiprogramming
Software - Practice and Experience vol 7, 3-35 (1977)

4. WIRTH,N
Design And Implementation Of MODULA
ibid.,67

5. DIJKSTRA, E.W.
Hierarchical Ordering Of Sequential Processes
Acta Informatica vol 1, 115-138 (1971)

6. HOARE, C.A.R. and WIRTH, N
An Axiomatic Definition of the Programming Language Pascal
Acta Informatoca Vol 2, 335-355 (1973)

*****

7.2 (Included in 7.1)

UCSD WORKSHOP ON SYSTEM PROGRAMMING EXTENSIONS TO THE PASCAL LANGUAGE

CONVENER

Kenneth L. Bowles	Institute for Information Systems

PARTICIPANTS

Tony Addyman	University of Manchester, England
Durga Agarwal	National Semiconductor
John Ahlstrom	Olivetti
Roger Anderson	Lawrence Livermore Laboratories
Don Baccus	Oregon Minicomputer Software
Jeff Bahr	National Semiconductor
Michael S. Ball	Naval Ocean Systems Center
Winsor Brown	General Automation
David Bulman	Pragmatics, Inc.
Joe Caporaletti	NCR Corporation
Richard J. Cichelli	ANPA Research Institute
Joe Cointment	Texas Instruments
Bob Dietrich	Tektronix, Incorporated
Stephen Dum	Tektronix, Incorporated
Glen Edens	National Semiconductor
Norm Finn	Rolm Corporation
Steve Franklin	University of California,Irvine
Jim Greenwood	Lawrence Livermore Laboratories
Al Hartmann	Intel Corporation

Charles Haynes	Basic Timesharing, Inc.
Carl Helmers	Byte Publications
Scott Jameson	Hewlett Packard Corporation
Dick Karpinski	Northwest Microcomputer Systems
Dennis Kodimer	Terak Corporation
Walt Kosinski	General Automation
Warren E. Loper	Naval Ocean Systems Center
Eugene Martinson	Data 100 Corporation
David C. Matthews	Process Computer Systems
Craig Maudlin	Renaissance Systems
Terrence C. Miller	Institute for Information systems, UCSD
James F. Miner	University of Minnesota
Gabe Moretti	Signetics Corporation
Mark D. Overgaard	Institute for Information systems, UCSD
William Price	Tektronix, Incorporated
Bruce Ravenel	Language Resources
Ruth H. Richert	Burroughs Corporation
Arthur H. J. Sale	University of Tasmania, Australia
William F. Shaw	Systems Engineering Laboratories
Kees Smedema	Philips Laboratories
Barry Smith	Oregon Minicomputer Software
Don Story	General Automation
Robert Strahl	Signetics Corporation
Skip Stritter	Motorola, Incorporated
Jeffrey M. Tobias	AAEC Research Establishment
Justin Walker	National Bureau of Standards
David Weil	Boeing Computer Services
Richard Woodward	American Microsystems, Inc.

STAFF

from the Institute for Information Systems

Chip Chapin
Greg Davidson
Albert Hoffman
Peter Lawrence
Joel McCormack
Keith Shillington
Richard Sites
Dennis Volper

and Rodney C. Steel from Tektronix, Inc.