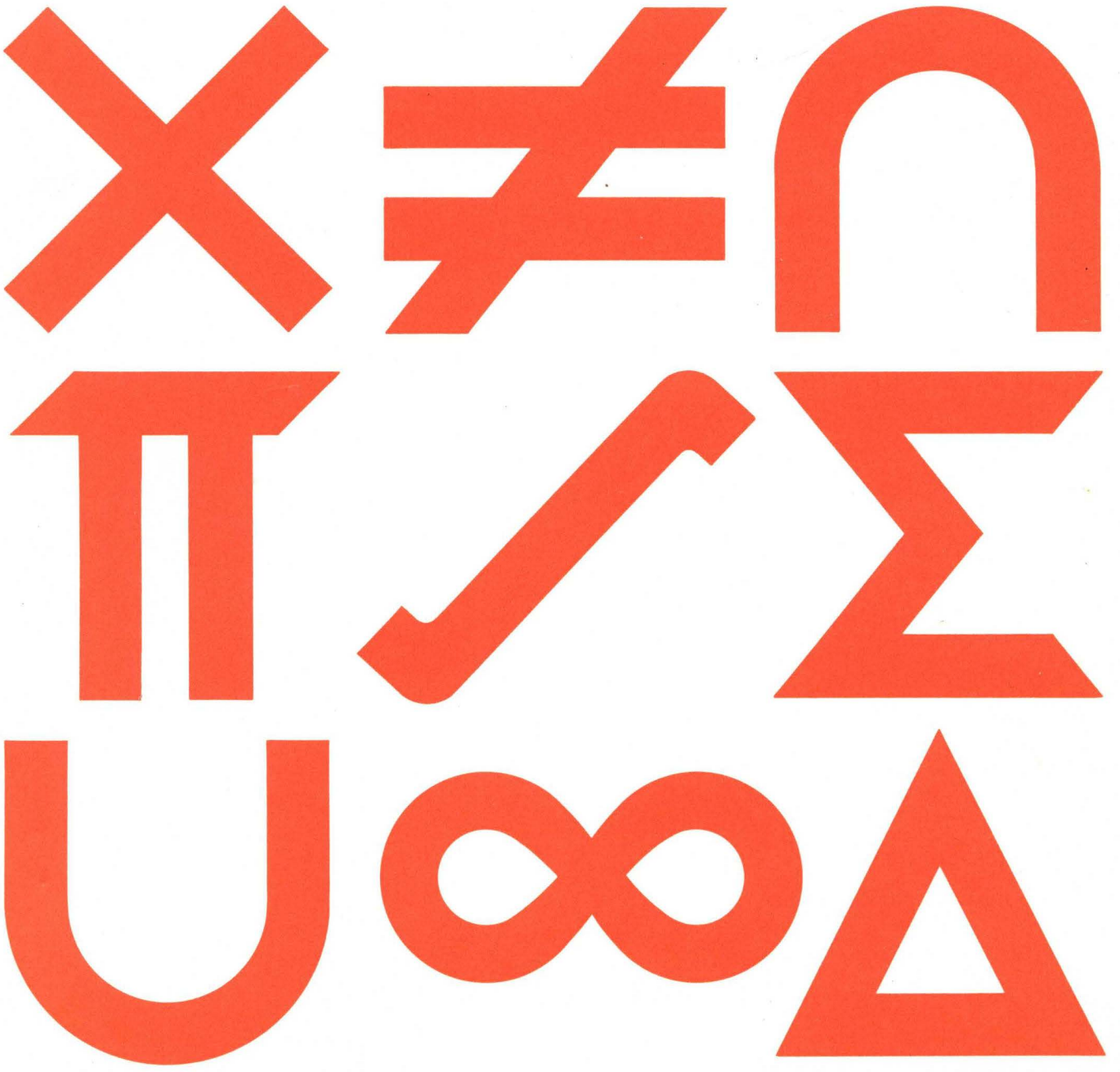
IBM CAMBRIDGE SCIENTIFIC CENTER
G320-2128, August 1979

A COMPARISON OF LANGUAGE C AND PASCAL

A. SPRINGER



1976 IBM CAMBRIDGE SCIENTIFIC CENTER REPORTS

G320-2110 April 1976
J. RUBIN, A Resource Allocation and Job Scheduling Model (22 p.)

G320-2111 April 1976
Y. BARD, A Characterization of VM/370 Workload (40 p.)

G320-2112 May 1976
T. MORE, JR., Types and Prototypes in a Theory of Arrays (65 p.)

G320-2113 May 1976
T. MORE, JR., On The Composition of Array-Theoretic Operations (62 p.)

G320-2114 May 1976
R. I. FRANK, Design Considerations in an Experimental Interactive APL Quadratic/Linear Programming Workspace (28 p.)

G320-2115 May 1976
S. G. GREENBERG, High Level Data Management for the Problem Solver (14 p.)

G320-2116 December 1976
W. P. FISCHER, The Heating Oil Consumption of One-Family Houses in Eastern Massachusetts (66 p.)

G320-2117 December 1976
W. P. FISCHER, A Note on Residential Heating Oil Inventory Policies (31 p.)

G320-2118 December 1976
W. P. FISCHER, Short Term Changes in the Residential Consumption of Heating Oil (25 p.)

G320-2119 December 1976
W. P. FISCHER, Short Term Residential Demand for Heating Energy in New England (45 p.)

G320-2120 December 1976
M. SCHATZOFF, System Performance, Measurement and Tuning (37 p.)

1977 IBM CAMBRIDGE SCIENTIFIC CENTER REPORTS

G320-2121 June 1977
Y. BARD, An Analytic Model of the VM/370 System (43 p.)

1978 IBM CAMBRIDGE SCIENTIFIC CENTER REPORTS

G320-2122 July 1978
P. BERKE, Tables, Files and Relations in Array Theory (27 p.)

G320-2123 July 1978
P. BERKE, Data Design with Array Theory (33 p.)

G320-2124 November 1978
Y. BARD, Some Extensions to Multiclass Queueing (27 p.)

G320-2125 December 1978
R. E. FRANK, Sparsity and APL (12 p.)

G320-2126 December 1978
R. I. FRANK, The Differential Geometry of the C^2 Unconstrained Optimization Surface (21 p.)

1979 IBM CAMBRIDGE SCIENTIFIC CENTER REPORTS

G320-2127 July 1979
G. McQUILKEN, Remote Management and Control of Distributed Interactive Computer Systems: A Preliminary Overview (19 p.)

G320-2128 August 1979
A. SPRINGER, A Comparison of Language C and PASCAL (20 p.)

G320-2275 December 1976
Compiled by LILES SMITH, Abstracts of Cambridge Scientific Center Reports (113 p.)

The availability of reports is correct as of the printing date of this report.

+ Appeared in an outside publication. Not available in Scientific Center report form. Please refer to the list of outside publications on the inside back cover for availability of reprints.

• Copies of report are no longer available from the Scientific Center.

IBM CAMBRIDGE SCIENTIFIC CENTER

TECHNICAL REPORT NO. G320-2128

August 1979

A COMPARISON OF LANGUAGE C AND PASCAL

ALLEN SPRINGER

IBM CAMBRIDGE SCIENTIFIC CENTER
545 TECHNOLOGY SQUARE
CAMBRIDGE, MA 02139

Abstract

This report is a comparison of Language C and Pascal from several aspects, including history, language features, suitability for system programming, suitability for structured programming, and implementability.

TABLE OF CONTENTS

A COMPARISON OF LANGUAGE C AND PASCAL

1. INTRODUCTION

Pascal and C are in the same class of languages as PL/I, FORTRAN, ALGOL 60, ALGOL 68, and COBOL. That is, they are intended to be procedural and compilable languages. This contrasts with the more specialized languages such as APL (for arrays), SNOBOL (for strings), and LISP (for lists), which are extremely difficult or impossible to compile. This also contrasts with non-procedural languages such as RPG, CSMP or GPSS. It is assumed that the reader of this paper knows one or more of the languages in the first group, and has some understanding of data structures, pointers, and recursion. This paper will primarily discuss the aspects of the two languages that make them unique, especially the things that make them easy or troublesome to use.

1.1. THE BACKGROUND OF PASCAL

Pascal was developed in a European programming language "tradition", and shows its ALGOL heritage. Niklaus Wirth developed it primarily as a teaching tool (4). It has become a popular language in universities on many machines. It also has influenced the design of many subsequent languages, including the final four candidates for the proposed Department of Defense "Ironman" standard programming language.

When minicomputers were the smallest computers around, the most popular language on them was BASIC. If another language besides assembler was used, FORTRAN was the usual candidate. With microprocessors BASIC is still the beginner's language, but now PASCAL is often considered the best candidate for a larger language.

The American National Standards Committee X3 has subcommittee X3J9 to prepare a proposal for a standard Pascal. So far, the proposed standard (6) is only a slight change from the Jensen and Wirth book (4) which has been the de facto standard. Standardization of needed extensions to Pascal seems to be further off in time.

1.2. THE BACKGROUND OF C

C also comes originally from European influences, and was originally based on a typeless language called BCPL, which was intended to be a simple and highly portable system programming language. BCPL was distributed widely when it first came out. At Bell Labs, a variant of BCPL was created, called B, which was still a typeless language, and some experimental minicomputer code, the beginnings of the UNIX

operating system, were done in B (UNIX is a trademark of Bell Laboratories). C was designed as a typed language based on B. UNIX was subsequently written almost completely in C. Since UNIX has become popular, especially in universities, the C language has become known in that fashion. C was intended primarily to be a system programming language, but has become a language usable for general purpose programming. It has been ported to several machines, mostly by Bell Labs efforts. Some non-Bell compilers are available for various machines. It is not nearly as well known as Pascal. There is no user group specifically for language C, although there is a UNIX user group that may have some of that function.

1.3. AN OVERVIEW OF PASCAL

Pascal is a relatively small language which has some block structure, recursive functions and procedures, arrays, pointers, and data structures, and the ability to do simple input/output. Unusual features, compared with PL/I or older languages, are the ability to define new data types, and to specify variables that have limited subranges of scalar or integer values. An unusual data type is the power set of a finite range of scalar values; it turns out to be a very useful equivalent of bit strings.

All declarations must appear before any executable statements that refer to the declared items. The form of the language is such that the compiler can be a one pass compiler without much difficulty. The language has some block structure in the ALGOL or PL/I sense, and only has the equivalent of PL/I automatic storage and a kind of based storage, called "heap" storage. All items are required to be completely declared. This implies that if a pointer is declared, then the type of the value it points to must also be declared. There are no defaults in the language. Like ALGOL, the standard language implies that separate compilations are not easy to provide, although there are extensions to some Pascal compilers that allow this.

Basic data types include characters, integers, "reals" (floating), Boolean, scalars, and subranges. Structured data types include arrays, records (similar to PL/I structures), pointers, sets, variant records (a sort of primitive ALGOL 68 union), and files.

Pascal has both assignment and procedure call for simple control structures. Composite control structures include BEGIN...END (which does not denote a block), FOR, WHILE and REPEAT loop statements, and IF...THEN...ELSE and CASE for alteration statements. There is a primitive GOTO statement.

Pascal has the expected relational and arithmetic operators, and some built in arithmetic functions. Except for conversion from integer to floating, all conversions must be explicitly written. Pointer generation and dereferencing is explicitly done, similarly to PL/I. There are no array or structure operations except component selection and assignment.

Input/output is done with built-in functions and procedures.

1.4. AN OVERVIEW OF C

C allows separate compilation, and so interconnection of functions is related to whether they are in the same file when compiled, and whether function names are declared external or not. Functions cannot be nested within functions (C has no procedures), although block structure does exist. Thus C has an overall program structure rather like that of FORTRAN with named common. Unlike FORTRAN, C allows recursion, and has the equivalent of automatic, static, and based storage.

Basic data types include character, two sizes of integer, two sizes of float, unsigned integer, and bit field. Composite types include array, structure, union, and pointer. C recently has had a type definition facility added to the language, although for a long time it did not have it. Boolean is not separate from integer data types in principle.

The basic control structure of the language is the expression; assignment is an expression. If a function is called without using its returned value, the value is simply discarded or ignored and is not considered an error. Thus functions can be used as procedures.

Composite control structures include a grouping statement equivalent to the PL/I BEGIN;...END; and loop statements such as a rather general FOR, and a WHILE. There is a conventional IF statement, and a primitive SWITCH statement that is a kind of computed GOTO which looks rather like a CASE statement. There is a simple GOTO statement, and statements for repeating or ending a loop from within the middle of the loop.

There are more operators in C than in Pascal, and automatic conversion between all basic data types is done, except for pointer. Automatic generation of pointers or dereferencing of pointers is done in many circumstances, unlike Pascal. There are no structure or array operators other than component selection.

Input/output is not part of the language, and is done typically with library routines.

C has a simple preprocessor style macro language that provides file inclusion capabilities, abbreviations, and alternative selection of code based upon programmer defined parameters.

2. DETAILED LANGUAGE COMPARISONS

In this section we will examine in some detail the significant features of C and Pascal, stressing where the two languages are significantly different from each other or from more familiar programming languages. Note that both languages were influenced by availability to their designers of 96 character ASCII terminals. Both use square brackets and curly brackets, for example.

2.1. THE OVERALL LANGUAGE STRUCTURE

Unextended Pascal effectively insists on the complete program being compiled all at once, whereas C can have groups of functions compiled separately. The order of writing program parts is much stricter in Pascal than in C. The Pascal program order is as follows:

```
PROGRAM name (files used by program)
LABEL list of labels;
CONST constant declarations such as PI = 3.141592
TYPE type declarations such as
      FLOAT = REAL;
      COLOR = (RED, BLUE, GREEN, YELLOW);
VAR   all variable declarations except those local
     to procedures or functions declared below
All procedure and function declarations
BEGIN
     the body of the main program
END.
```

The LABEL, CONST, TYPE and VAR sections are omitted if there is nothing to declare in them. Built-in functions and procedures of Pascal do not have to be declared in order to be used. A function or procedure has the same form as that shown by the program, except that the END has a semicolon instead of a period, and instead of PROGRAM, the keywords FUNCTION or PROCEDURE are used, along with declaration of the types of arguments, and the value returned if it is a function. Within any function or procedure, further functions or procedures can be declared.

The CONST declaration provides a way of naming constants. LABELs must be declared before they appear in the text. TYPE declarations can be viewed as an extension mechanism, or more simply as an abbreviation mechanism for declarations.

The structure of a C program is much freer than that of Pascal. It is loosely of the following form:

```
declarations
function
declarations
function
etc.
```

Essentially, declarations outside of functions are normally equivalent to PL/I EXTERNAL declarations. Also all functions are externally known when the program is loaded with other separately compiled programs. It is not possible to nest functions within functions, unlike Pascal. This means there can be a name conflict problem. On the other hand, the grouping statement (denoted by curly brackets), can be a block anywhere, with its own declarations of automatic variables, just as in PL/I, and unlike Pascal. Essentially all variables must be declared before they are referenced.

A C function has the following form:

```
name(argument list, if any)
argument declarations, if any
{
  declarations and statements, if any
}
```

Note that the looser structure of C allows the easy inclusion of routines from other sources, such as libraries of functions. This is not so easy for Pascal, where it may be necessary to split apart the global declarations from the functions. Combination of C programs is also aided by the ability to compile them separately, unlike Pascal.

2.2. THE DATA STRUCTURES

Under the heading of data structures, we will briefly discuss basic types, such as float and integer, and structures, which are combined from basic types. Then storage classes will be discussed. Operations that can be performed on specific types will be discussed in following sections.

2.2.1. BASIC TYPES

In Pascal some attempt was made to define the language independently of specific machine word sizes. The approach was not that of PL/I, where the arithmetic precision rules are machine independent. Instead, the programmer has available a standard constant MAXINT that tells him the integer precision, so that he can write programs in terms of that, and achieve machine independence if it is important. There is only one precision of integer, although the declaration of a variable as being in a subrange will allow the compiler to compile shorter precision, where the compiler writer sees fit. There is no comparable precision information about real arithmetic.

In Pascal, characters are recognized as being machine dependent. There is a function ORD(C) that returns an integer indicating the relative order of the character argument in the character set. CHR(I) accepts an integer, and returns the character that is the Ith character of the character set. With care, it is possible to write programs that are independent of the character collating sequence of the specific implementation of Pascal. Pascal apparently does not promise to implement any particular minimum character set as values of character variables. Note that both in C and Pascal, a character variable only holds one character at a time, and to have strings of characters, character arrays must be used.

In contrast to Pascal, C is more machine dependent, since its types are derived from the natural operand sizes and types available on the PDP-11. They also happen to correspond well to those on the IBM 370 or the Series/1. They include two sizes of integer, SHORT and LONG, and INT is the usual declaration if the choice between the sizes

is left up to the compiler. All that is promised is that size(SHORT) <= size(INT) <= size(LONG). The size of INT must be at least the size of an address. There is also an unsigned integer which is the same size as INT. Characters are variables that hold one character, and otherwise can be treated as integers. Note that Pascal allows comparison of arrays of characters (if of the same size) in a single operation, whereas C does not. There are two sizes of floating point, FLOAT and DOUBLE.

C has no Boolean data type, and uses the distinction between zero and non-zero for this, for all basic data types except floating point. Pascal has Boolean predefined as a scalar data type of two elements, TYPE BOOLEAN=(FALSE,TRUE) and all operators for scalar data types are usable on type Boolean.

Pascal has two data types which are not in C. Any scalar type can be defined by enumeration of its elements. For example, we could declare a new data type called COLOR as follows: TYPE COLOR=(YELLOW, RED, ORANGE, GREEN, BLUE, PURPLE), and then declare variables of that data type. The identifiers in the parentheses are then used as constants of that data type. The order of these constants is known to the relational operators, so that RED<ORANGE is true. A variable of type COLOR would hold only one color at a time.

Pascal allows the definition of subranges of any data type (that is, integer or scalar) that has successor and predecessor functions. Thus we could declare

```
VAR   I, J, K: 0 .. 99;
      ROOM, WALL: RED .. BLUE;
```

I, J and K take on only subranges of integer values, and could in principle be stored in a byte in the IBM 370. Since the data type COLOR has only six possible values, it could be represented by integer values 0 to 5 or 1 to 6 in the implementation of the data type. The variables ROOM and WALL would take on only four of the six possible colors, and in principle their values could be stored in 2 bits.

In C there is a data type called a bit field, which has limited usage. Basically it is unaligned unsigned integer data which is not allowed to overlap integer sizes or boundaries. They are subparts of integers, and have few other operations than assignment or value accessing defined for them. They allow machine dependent accessing of bits.

2.2.2. COMPLEX TYPES

Both languages have arrays, and use square brackets to designate subscripts. In both cases, strictly speaking, arrays are one dimensional, but any data type, simple or complex, may be components, including arrays. Thus an array of arrays is equivalent to a single two dimensional array. Thus in an array of arrays of integers, the expression A[2][3] will access the element at row 2 and column 3, if it is viewed in the conventional manner. In Pascal, this may be abbreviated as A[2,3].

C may not pass arrays or structures as function arguments, or return them as values, whereas Pascal can. On the other hand, both

languages can pass or return pointers to arrays or structures. It is possible tc leave undefined the size of arrays in C, where it must be declared in Pascal. This is a severe restriction in Pascal, because it makes it impossible to define an array handling routine that is independent of the actual size of the array to be manipulated. Thus the same routine could not invert both a 10x10 array and a 11x11 array, unlike in PL/I or FORTRAN. This also causes problems in string manipulation, since strings are arrays of characters. Thus there are strong interests among Pascal users to extend the language to allow at least a dope vector style of handling arrays, as is done for PL/I.

Both C and Pascal have data structures, of a power similar to PL/I's, although not using level numbers. For example, in Pascal we could define:

```
TYPE COMPLEX = RECORD REALPART, IMPART: REAL END
```

where the data type (the structure) has two components of the same data type REAL. Accessing of components is as for PL/I; if we declare

```
VAR X COMPLEX;
```

then we may access the imaginary part of the variable with X.IMPART. Although we show COMPLEX as a new data type, the variable X could have been declared instead as

```
X: RECORD REALPART, IMPART: REAL END;
```

The equivalent declaration in C is:

```
struct { float realpart; float impart} x;
```

As can be seen, the order of types and names in C is reversed from that of Pascal.

It is possible in both languages to define variables that can have one data type at one point, and a different data type at another point. In Pascal this is called a variant record, and in C, a union. In both languages, they are declared very similar to structures, and the "component" names are used to designate what the possible value types are, and access of a component implies that that type is what is currently stored in the variable. In C an example is declared as:

```
union { int p; float r} y;
```

The variable y can hold either an integer, or a floating point value, but only one at a time. To access it for the int value we say y.p, and y.r accesses the value as if it were floating. Neither language checks that you are in fact using the correct accessing method for the value. The storage size and alignment is the maximum necessary to hold the largest and most strictly aligned value declared.

Pascal has a complex data type, called SET, which is very useful, and is not found in language C. A set may be declared to be built out of items of a finite set of data, typically a scalar data type, or a subrange. The number of items cannot be more than the maximum set size allowed by the compiler, which often is about 60 bits or so, for historical reasons. An example will be helpful:

```
TYPE FRUIT = (APPLE, ORANGE, BANANA, GRAPE);
VAR FRUITBASKET: SET OF FRUIT;
```

Subsequently we might set the variable FRUITBASKET to a particular set of the scalar values declared above:

```
FRUITBASKET := [ ORANGE, GRAPE ];
```

After the assignment, the set will contain the two values, ORANGE and GRAPE, but not the other two possible values. In practice, a set is represented by one bit for each possible type of thing that can be stored in it. The value of scalar type FRUIT can be stored in two

bits, if we declared variables of that type. It has four possible values. A set of that type would require four bits, one for each possible type of fruit that might be stored in the set variable. Again, although the space savings is possible in principle, most Pascal compilers implement only one size of set.

Pointers in both languages must have declared the type of value to which they point. Thus a pointer to an integer is not the same data type as a pointer to an array of characters. The checking of the pointer type is much stricter in Pascal than in C, but both are much stricter than what can be checked in PL/I. In Pascal, a pointer may be created only by calling a built-in function NEW, which does roughly the equivalent of PL/I ALLOC. That is, it calls a routine like GETMAIN to supply the storage to be pointed to. This type of storage is called "heap" storage. In C, this is possible, but it is also possible to compute a pointer to a variable (or component of a variable) of any storage class. This corresponds to the ADDR function of PL/I. Thus C is much more flexible than Pascal in this respect. In effect, in Pascal pointer variables can only refer to based storage (in PL/I terms), whereas in C, pointers can be to any class of storage.

In Pascal, functions and procedures may have any type of argument value or variable passed to them, and may return any type except function or procedure names, whereas in C only simple types or pointers to any type can be passed or returned as values. Pointers to functions may also be passed or returned in C.

2.2.3. STORAGE CLASSES

Pascal has only one storage class, which is the equivalent of PL/I automatic storage, for non-pointer variables. For pointer variables, there is only one class, which is roughly like based storage of PL/I, gotten and freed under program control.

By contrast, C has several more storages classes. It has the equivalent of automatic storage, which as usual is acquired at block entry and freed at block exit. In both languages, a stack is a natural place for such storage. C also has static storage, in the PL/I sense, which may be either known outside the compilation, or within the compilation only, or just within a single function. If known outside the compilation, it is of course like PL/I EXTERNAL variables. Basic data type variables can also be declared storage class REGISTER, although there is no promise that the compiler will in fact keep the variable in a machine register. Such a declaration can be taken as advice to the compiler code optimizer. C also can provide the equivalent of based storage via its pointers, but is not restricted in what kind of storage a pointer variable can point to.

2.2.4. INITIALIZATION OF VARIABLES

Pascal has no means of designating initial values of variables. This means that initialization must be done by assignment statements in the program body or a separate function for that purpose. By contrast, C allows initialization specifiers for static, external and local (automatic) variables. For local variables, this means that the variable declaration is not split apart from the setting of its initial value, even though the code generation may be the same as for Pascal. This is an aid to documentation. For globally accessed variables, which can be static in C, the advantage is that no assignment statements are generated to be executed at run time; in Pascal run-time code necessary for initialization.

2.3. THE CONTROL STRUCTURES

Both languages can group sequences of statements together, so that they can be used as if they are a single statement. This is important because most of the complex control structures are in terms of single statements as components, e.g. IF expression THEN statement ELSE statement. The grouping statement of Pascal is BEGIN ... END. That of C is {...}. The semicolon is used as a statement separator in Pascal (like ALGOL and unlike PL/I). The semicolon is used to end a statement in C, unless the "statement" is really a statement group. This is almost but not quite the PL/I convention.

2.3.1. LOOP STATEMENTS

The loop statements are relatively similar in the two languages. The FOR statements are intended for initialization and iteration of a variable that can be used within the body of the FOR. Pascal only allows stepping up or down by a value of 1, and does not consider the iteration value after the end of the loop to be defined, which is a limitation on the user. C allows any initializer, any stepping statement, and any test for completion of the loop, and thus provides more generality.

Both languages have means for looping, with a test either at the beginning or at the ending of the loop, using WHILE or UNTIL keywords.

C has a means of ending the execution of the body of a loop from in its middle. The BREAK statement causes control to go after the loop. CONTINUE (not a well chosen keyword) causes the next iteration of the statement to begin, without executing the rest of the body of the loop. Pascal must get these effects by GOTO, or else by having the remainder of the loop in an IF statement.

2.3.2. ALTERNATE SELECTION STATEMENTS

Both statements have alternate selection statements of the IF form, with an optional ELSE. Both have a form of CASE statement, although Pascal's presents less difficulties to the user. An example of the Pascal form is:

```
CASE expression OF
     L1: statement;
     I2: statement;
     L3, L4, L5: statement;
     I6: statement;
       ...
END
```

The "labels" L1, L2, etc., are really any constant of the data type of the expression. Unfortunately, there is no means for specifying what to do for values of the expression that do not have a label with that value. This means that all possible values that can occur must be enumerated explicitly.

The form of case statement in C is:

```
switch(expression)
{
case1: statements
        break;
case2: statements
        break;
   ...
default: statements
  }
```

The expression must be integer valued, and only integer constants must be individual cases. If break is not used to jump out of the loop at the end of the statements handling a case, then execution will continue in the statements for the next case! However, at least C provides a way (the default case) of not having to mention all possible cases that can occur.

2.4. THE OPERATORS

Both languages have the expected collection of arithmetic and comparison operators. Both have the ability to build complex "Boolean" expressions with AND, OR, and NOT style operators, although their notations differ. Beyond this, Pascal has some set operators, and some standard arithmetic functions, such as for cosines or logarithms.

The Pascal set operators include union, intersection and set difference, and comparison (Boolean valued) operators to test set inclusion, and set membership. For example, if there were two variables SA and SB of type SET OF FRUIT, defined earlier, we could write:

```
IF APPLE IN (SA + SB) THEN statement
```

to test if APPLE is in either or both sets.

Language C was designed to generate code without having to compile subroutine calls to implement operations. Thus it does not

provide trigonometric and similar functions as a part of the language, although obviously they can be made easily available in libraries. The point is that such functions are not standardized as a part of the language. Aside from the compilability restriction, and the lack of sets and operators on them, C has considerably more operators than Pascal. For arithmetic operators, there are left and right shift, and the Boolean operators really do bitwise "and, "or", "not", and "exclusive or" operations. Additionally, there are two more Boolean operations, which return only 0 or 1 instead of doing bitwise "and" and "or" operations. They promise to test their first operand, and if the final operator result is known, the second operand is not computed. By contrast, in Pascal (and many other languages) it is undefined whether the second operand is computed or not, and therefore it is sometimes not safe to write some apparently natural expressions. For example, given that m is an array with subscripts ranging from 0 to 10:

```
if ( (0<=i) && (i<=10) && (m[i]>20) ) statement
```

is a safe statement to write, since the access of array m would only be done if the subscript i is valid. The equivalent in Pascal could only be done safely with nested IF statements:

```
IF (0<=I) AND (I<=10) THEN
    IF M[I]>20 THEN statement
```

If the C programmer used & (bitwise and) instead of &&, it would be an unsafe statement to execute, for the same reason it is in Pascal.

C has many increment and decrement operators. For example, the unary operators ++x and --x increment the value of x before presenting the value to the rest of the expression they are in. x++ and x-- increment or decrement the value in x after delivering the value.

C has many assignment operations, and they all can be used as expressions. A simple assignment is of the form variable = expression.

```
variable =+ expression
```

is equivalent to:

```
variable = variable + (expression)
```

This abbreviated form is available for most binary operators.

C has available what was called a conditional expression in ALGOL 60. It is effectively an IF statement that returns one of two values:

```
x = ( a>b ? a : b )
```

In the example, the larger of the values of a or b is stored in x. That particular conditional expression is equivalent to PL/I MAX(A,B).

Another important difference between Pascal and C is that C allows arithmetic to be performed on pointers in a limited way. I.e. addition and subtraction can be done with the forms p+i, p-i, p-p, ++p, --p, p++, p--, p=+ i, and p=- i, where p represents any pointer, and i any integer value. Normally the pointer is declared to point to an array of some data type. When the pointer is stepped by one, it is really made to point to the next item in the array. If it is an array of integers, which might be four bytes in size, the real operation might be incrementing an address by four. When taking the difference of two pointers, they must point to the same data type. If the integer difference is added to the second pointer, then the first pointer would be the result. Note that there is no check that you run off the end of the array in either direction, although in principle that check might be possible with dope vector style information.

2.5. DATA CONVERSIONS

Essentially all conversions from one type to another in Pascal are done with explicit functions, except for conversion from integer to real. This contrasts dramatically with C, where all non-complex data types are freely converted from one to another, except for pointers. C is rather like PL/I in its freedom of conversions, although the basic types are all essentially arithmetic in nature. Aside from such conversions, which tend to go from smaller to larger, and from integer to floating, when mixed, C also does referencing and dereferencing in a manner rather like ALGOL 68. This may be illustrated by two examples. If an array is written as an argument to a function, since arrays are not passed as arguments, C assumes you meant to pass a pointer to the array (which is legal) and generates the pointer for you. Similarly if a pointer B is declared to point to an array, since subscripting of pointers is meaningless, you may write B[5], and the compiler assumes that you wanted to follow the pointer to the array and access its 5th component. Both of these constructs would be illegal in Pascal.

3. FUNCTIONS AND PROCEDURES

The distinction between functions and procedures is that the latter does not return a value. Pascal has both forms of subroutines, and C only has functions. Since C allows statements to be simply function calls, and then ignores any value returned since the call is not part of a larger expression, there is no important loss of capability in C by having no procedures.

C only allows basic types and pointers to be passed as arguments (no arrays or structures). Pointers may be passed which point to anything. There is a similar restriction on values returned. The arguments are always passed by value, i.e. their value is always copied to the stack, as if the parameter variables are simply local variables of the called program. In fact, the values may be changed in these parameter variables without affecting the original variable. In order for a function to be able to modify something that the caller passes to it, the pointer to that thing must be passed as an argument, and of course the function should declare the parameter as a pointer to the appropriate type.

In Pascal, the conventions are more like that of PL/I, i.e. there is a distinction between functions and procedures, and there is a distinction between arguments passed by reference or by value. Parameters explicitly declared VAR must be variables on the calling side, and can have their value modified by the function or procedure called. Parameters not declared VAR can be any any expression, whose value is copied when passed to the function or procedure. Pascal can pass any data type to a function or procedure, and can return any type except

function and procedure names.

Pascal and C differ considerably in the strictness of their type checking for functions and procedures. In essence, Pascal will strictly check at compile time the correspondence of argument and parameter, and the value returned with the declared type to be returned. Although you declare parameter types in C, since it is necessary for the use of those parameter values within the function, no check is made against those definitions when compiling calls. There is no check to see that the function is returning the correct type either. Thus it is both easy to make mistakes, and easy to "cheat" deliberately, to get machine dependent effects (treating a pointer as an integer, for example). In the UNIX system, although the C compiler does not do these sorts of checks, there is a program called LINT which will do so, when given a series of files of C programs that are intended to run together.

C functions are much more flexible than Pascal functions or procedures in one sense. It is possible to define any variable as having a data type with one of its dimensions (of array) as unknown in size. An example is a character array of indefinite length. It is the program's responsibility to not go beyond the actual length of the array, and since there is no dope vector information, the program must have some way of determining the actual length. For character strings, the usual convention in UNIX is to end the string with a null character (hex 0). Another convention might be to pass the length as a separate argument. Pascal has a much stricter control over its arguments and returned values, as mentioned above. In effect, it is not possible to pass two different size arrays to a routine, and have it adjust to the size. As a result of this strictness, there are efforts to extend Pascal to allow this. One possibility is to add appropriate dope vector information for arrays, as is done for PL/I.

4. INPUT/OUTPUT

Input/output is not specified as a part of the C language, since the designers felt that it was not appropriate. As a result, input/output is done by libraries of subroutines. In UNIX these routines are written in C, which is coded in machine dependent ways if necessary.

Files are a data type in Pascal, but many of the usual operations valid for any other data type are not valid for files. Instead, file operations are done by built-in functions (whose argument rules violate the type checking restrictions imposed on user written routines). Any particular file can be viewed as a sequential file, a kind of array which can be processed only from front to back, with a window looking at only one component of the array at a time. The window is essentially a pointer to a buffer for the file. Although there are many cases of input/output where this suffices, it is not general enough for all types of files, e.g. variable length record files or

random access files.

5. EVALUATION OF THE LANGUAGES

The two languages are evaluated from several viewpoints in following subsections, and the conclusion section has a brief summary of significant differences or tradeoffs taken by the two languages.

5.1. LANGUAGE SIZE AND EASE OF COMPILATION

It is clear that C is a larger language, mostly because it has more simple data types, and more operators on them, and more automatic conversions between them. Both languages were designed to be simple to compile. Pascal compilers often are based upon methods of compilation designed to make the compiler easily portable. Many compile to an intermediate language usually called P-code, which may then be interpreted, compiled into machine code, or perhaps processed a macro assembler. P-code is not necessarily the best intermediate language for all machines. C compilers normally generate machine code. Some C compilers have been designed to be portable.

5.2. LIBRARIES

Since it is easy to combine programs from several sources into a single file in C, and in Pascal it may be necessary to split apart such programs to combine them with others, C is clearly superior to Pascal on this point. However, since C has no nesting of functions, and insists that all function names are external, there can be some name clashes, which can cause difficulties. An extension of Pascal could be made that retains its type checking, but allows both nesting of functions and a C-like freedom of ordering of items. Such an extension would probably require giving up the ability to compile Pascal in one pass.

The C preprocessor has the ability to include source files within a program being compiled. Pascal does not have this capability. The preprocessor also supplies a simple macro and abbreviation facility.

5.3. STRUCTURED PROGRAMMING

For structured programming, the languages can be compared on several points. One is the library issue already mentioned.

In comparison of statement types, Pascal is ahead of C for the case statement, since it allows a cleaner way of stating alternate cases. However, Pascal needs a default statement in its CASE.

It is clear that the lack of nesting of functions in C is a problem, and that Pascal is superior in this area. However, Pascal does not have any blocks other than whole functions or procedures, whereas C allows any grouping statement to be a block with its own local variables. Thus variables used only within a limited area could be declared in that area, and would exist (on the stack) only when that area is being executed.

It is clear that the type definition facility of Pascal and C are very useful features for hiding details (encapsulation) of new data types as needed. With the C define facility, it is also possible to define macros to encapsulate sequences of code that are generated in-line. In Pascal this can only be done by subroutine calls. In C, for example, we may define an in-line MAX operation by:

```
#define max(A, B)  ((A) > (B) ? (A) : (B))
```

Then the line:

```
q = max(m-1,20) * j;
```

would expand as:

```
q = ((m-1) > (20) ? (m-1) : (20)) * j;
```

Although the macro facility is useful, it is not a substitute for the ability to define new operators, such as is available in more recent languages like CLU.

5.4. EASE OF LEARNING

It seems clear that on many counts, Pascal is much easier to learn. C has many more tricky points, and as in APL or PL/I, almost any expression has a meaning. Pascal will catch you on "strange" combinations of operations. Also the order of declarations is obscure in C, compared with Pascal, as the following illustration shows.

```
INT * * ( * QQSV() ) [] ;
 6  5 4   2     1    3
```

QQSV is the variable, and the numbers shown below the C declaration illustrate the order of declaration. QQSV is a (1) function returning a (2) pointer that points to (3) an array of indefinite size whose components are (4) pointers to (5) pointers to (6) integers. In Pascal the order is like that of PL/I, from variable name to type, from structure to component of structure, from pointer to thing pointed to, all from left to right. The only thing that can be said for C is that the declaration order tends to be written in the same pattern as when writing expressions to access the variable. E.g. `*(**QQSV(X))[30]` would end up with an integer. This ability to build up many operators in a single statement, which can have several assignments, has the same lack of clarity as APL "one-liners".

The Pascal equivalent of the above function declaration would be:

```
FUNCTION QQSV(P: INT): @ ARRAY[0..40] OF @ @ INTEGER;
BEGIN
     function body
END;
```

The Pascal expression equivalent to the C expression is: QQSV(X)@[30]@@ (Note we have used "@" instead of the Pascal up-arrow.) The order of access is left to right, which is also the order of declaration. The conclusion is that Pascal is clearer both for declaration and usage in this language area.

5.5. SYSTEM PROGRAMMING

It is clear that C was designed for system programming, and its data types reflect the machine it matured on, the PDP-11. With a few hardware dependencies, gotten mostly by what Pascal would consider type violations, C can be used for almost all system programming situations. It has no provisions for execution of privileged instructions, which must be written in another language.

Pascal was not designed for system programming, but it has been extended or modified to provide such languages as Concurrent Pascal, and MODULA, which are more suitable for system programming. They do not allow you to get as close to the machine as C, and are extensions in the direction of parallel processes, multitasking, etc.

C was explicitly designed to not address the problem of multitasking, parallel programming, process synchronization, etc. These things are carried out by functions called by the programs. This makes the C programs simpler, although some overall assumptions about the nature of tasks in UNIX, and also about how stacks are implemented, have combined to make this a reasonable choice. This approach has allowed all except about 1000 lines of code of UNIX to be written in C.

Pascal is often translated to an intermediate language called P-code, which is the rather limited instruction set of a hypothetical stack oriented machine, which is then interpreted, or compiled into some real machine instruction set. There is no reason in principle why Pascal cannot be compiled efficiently, which is one necessity for system programming. For example, a set should not take up more room than needed, but usually most compilers do not try to optimize such a thing.

The main restrictions on Pascal, compared with C, for system programming, are the lack of external and static variables. Also it would help to be able to point to other than "based" (heap) storage. Pascal can pass function and procedure names as parameters, but cannot store pointers to functions in variables. C (and PL/I, for example) can do this. This facility allows one program to dynamically load another, or store which function is associated with some resource without having to compile those functions together. The ability to have dynamically specified dimensions for arrays is also necessary. It would be very useful for structured programming to relax the order of declarations of items, e.g. intermixing constants, variables, and

types. It would still be reasonable to insist that an item must be declared before use.

It is not certain that Pascal can reasonably take advantage of a machine that has several precisions of arithmetic. There is some hope for integers, since one can declare an integer to have a subrange, and if it can be declared to have a value in the range -127..127, the compiler conceivably could allocate the integer in an 8-bit byte. However, there does not seem to be a way of using two floating point hardware precisions in the current language, and it may pay to extend it to allow declaration of two precisions of floating point. It is very useful to have separate compilations of Pascal, which do not require later linking that make the modules appear as if they were compiled together. It is clear that the pointer arithmetic of C is useful and powerful. It is a potential candidate for Pascal extensions.

5.6. APPLICATION PROGRAMMING

The fact that C has been used for many applications in the UNIX system shows that it can be a good language for this purpose. Part of this facility for programming comes from the UNIX system itself, and perhaps much more from the fact that libraries can be separately developed (such as for input/output), and then shared among users. Pascal does not provide this in most implementations.

By contrast, Pascal is probably easier to use due to its cleaner language design, as long as its restrictions (lack of dynamic arrays, for example) do not get in the way. It certainly seems to be a more readable language and therefore is a better candidate for application programming, with some minor extensions.

6. LANGUAGE CHANGES

In this section will be summarized the language changes that are recommended for Pascal. It is felt that in the long run Pascal has a wider audience, and a bigger potential for use in a wide range of areas. This is because C has several liabilities that indicate it has already grown about as far as it can go. The C liabilities are: (1) expression syntax that is too complex, when taking into account implicit referencing and dereferencing and conversions, (2) a defective case (SWITCH) statement, and (3) many ways of escaping the type checking mechanisms such that unsuspected mismatches might not be easily detected. (4) It is all too easy to make mistakes in writing operators, and end up with a different operator. For example, a common bug in C is writing IF (A = 1)... when it is intended to compare A with 1. But "=" is assignment, and "==" is comparison for

equality. This mistake is not caught by the compiler since assignment is an expression and the statement is therefore legal. The value of A in the example would be tested for 0 value, and since it would be non-zero, it would be considered true. This sort of deficiency cannot be corrected without making drastic incompatible changes to C. By contrast, extensions to Pascal could be upwards compatible with the standard Pascal.

The extensions recommended for Rascal include (1) external variables and separate compilation of functions and procedures, (2) the additions of blocks with their own static or local variables, (3) STATIC storage types, (4) the ability to point to more types of storage, (5) the ability to store function and procedure names in variables, (6) "dynamic arrays" in the sense that their size need not be known by a function or procedure until the array is passed to it as an argument, (7) declaration of initialization of variables, (8) a default case for the CASE statement, and grouping of cases by subrange, (9) a method of specifying precision of floating and integer variables such that various precisions of real hardware can be easily taken advantage of, (10) the ability to declare things in a more flexible order, so that functions written elsewhere or on library files can be included more easily, (11) the means of ending the current loop iteration or leaving the loop without having to reach the bottom of the statement, (12) making the order of evaluation of operands for AND and OR explicit, and (13) possibly some pointer arithmetic somewhat along the lines of C, although with the ability to check range violations at run time.

Besides the above language extensions, I would want the compiler to generate reasonably efficient code, and with the option of omitting run time checking for certain things, in order to do a minor amount of system or machine dependent operating system coding. It would be necessary to generate code that would be reentrant, if that is possible for the target machine, and to have the ability to have multiple modules and multiple program stacks (one per task) in handling multiple tasks. Unfortunately some Pascal run time support facilities grab all of free storage for their stack and heap storage.

Several extensions and modifications to Pascal exist such and MODULA and Concurrent Pascal. Some data abstraction languages such as CLU or EUCLID have been influenced by Pascal. ADA (7), the newly proposed DOD standard, was influenced by Pascal and its descendants, and contains all of the improvements to Pascal that were suggested above. ADA is a larger language than C or Pascal, to judicious extensions to Pascal seem still worthwhile. It is perhaps too soon in the development of ADA to consider a subset of that language instead of an extension to Pascal.

7. CONCLUSIONS

We can summarize the differences between the two languages as follows: Pascal has fewer basic data types, and checks them more strictly. Pascal sets are better than C bit fields. Pascal does better type checking, and as a result in C you can "cheat" more easily, or make inadvertent and undetected mistakes. For overall program structure, C is more convenient and flexible, except that it does not allow nesting of functions, and insists on making them all external. C is superior in pointer arithmetic, has more conversions, but because it does it implicitly in many circumstances, subtle mistakes can remain undetected for a long time. Pascal is safer in this regard. Pascal's structured statements are less general, but also safer. C has more storage types, some of which are necessary for certain types of programming.

In summary, Pascal is cleaner and easier to use without making subtle mistakes, but is smaller and more restrictive. For the languages as they currently exist, C has more power and is better for system programming and possibly for general purpose programming. But with extension of relatively simple sorts, Pascal would be better and cleaner for most purposes. I would recommend that Pascal extensions be made rather than using a standard C, mostly for subsequent maintainability and readability of programs.

8. BIBLIOGRAPHY

Some information for this report was derived from an ACM Professional Development Seminar, "C vs. Pascal", taught by P. J. Plauger, Washington, D. C., December 8, 1978.

8.1. C LANGUAGE REFERENCES

(1) Brian W. Kernighan and Dennis M. Ritchie, "The C Programming Language", Prentice-Hall, 1978.
(2) D. M. Ritchie, et al., "The C Programming Language", Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978.
(3) S. C. Johnson and D. M. Ritchie, "Portability of C Programs and the UNIX System", Bell System Technical Journal, Vol. 57, No. 6, Part 2, July-August 1978.

8.2. PASCAL REFERENCES

(4) Kathleen Jensen and Niklaus Wirth, "Pascal User Manual and Report", 2nd Edition, Springer-Verlag, 1974.
(5) Kenneth Bowles, "Microcomputer Problem Solving Using Pascal", Springer-Verlag, 1977.
(6) "The BSI/ISO Working Draft of Standard Pascal", Pascal News, January 1979. Also "Toward a Pascal Standard", Bruce W. Ravenel, Computer, April 1979.

8.3. OTHER REFERENCES

(7) "Preliminary ADA Reference Manual", SIGPLAN Notices, Vol. 14, No. 6, June 1979, Part A. "Rationale for the Design of the ADA Programming Language", J. D. Ichbiah, et al., SIGPLAN Notices, Vol. 14, No. 6, June 1979, Part B.

SCIENTIFIC CENTER REPORT INDEXING INFORMATION

1. AUTHOR(S) :
Springer, Allen

2. TITLE :
A Comparison of Language C and Pascal

3. ORIGINATING DEPARTMENT
Cambridge Scientific Center

4. REPORT NUMBER
G320-2128

5a. NUMBER OF PAGES
20

5b. NUMBER OF REFERENCES
7

9. SUBJECT INDEX TERMS

Language C
Pascal
Programming Languages

21 - Programming

6a. DATE COMPLETED
06/28/79

6b. DATE OF INITIAL PRINTING
August 1979

6c. DATE OF LAST PRINTING

7. ABSTRACT :

This report is a comparison of Language C and Pascal from several aspects, including history, language features, suitability for system programming, suitability for structured programming, and implementability.

8. REMARKS :

1977 IBM CAMBRIDGE SCIENTIFIC CENTER OUTSIDE PUBLICATIONS

A. P. DEMPSTER, M. SCHATZOFF, N. WERMUTH, Simulation Study of Alternatives to Ordinary Least Squares, Journal of the American Statistical Association, March 1977, Vol. 72, No. 357, 77-106

Y. BARD, The Modelling of Some Scheduling Strategies for an Interactive Computer System, in: Computer Performance, North Holland Publishing Co., 1977, 113-137

1978 IBM CAMBRIDGE SCIENTIFIC CENTER OUTSIDE PUBLICATIONS

Y. BARD, M. SCHATZOFF, Statistical Methods in Computer Performance Analysis, in: Current Trends in Programming Methodology, Vol. III, Software Modelling, Prentice-Hall, Inc., 1978, 1-51

Y. BARD, The VM/370 Performance Predictor in Computing Surveys, Vol. 10, No. 3, September 1978, 333-342

Y. BARD, An Analytic Model of the VM/370 System in IBM Journal of Research and Development, Vol. 22, No. 5, September 1978, 498-508

Y. BARD, Design of an Integrated Measurement Facility, SEAS Proceedings, Spring Technical Meeting, Berne, Switzerland, April 3-7, 1978, 243-252

N. ROCHESTER, F. C. BEQUAERT, E. M. SHARP, The Chord Keyboard, Computer, Vol. 11, No. 12, 1978, 57-63

A SPRINGER, L. LAZZERI, L. LENZINI, The Implementation of RPCNET on a Minicomputer, Computer Communication Review, Association for Computing Machinery, Vol. 8, No. 1, 1978

1979 IBM CAMBRIDGE SCIENTIFIC CENTER OUTSIDE PUBLICATIONS

L. SEAWRIGHT, A Perspective on Virtual Machines, Virtual Machine Workshop Proceedings, Gesellschaft für Informatik e.v., Munich, West Germany, March 15-16, 1979

L. H. SEAWRIGHT, R. A. MacKINNON, VM/370 – A Study of Multiplicity and Usefulness, IBM Systems Journal, Vol. 18, No. 1, 1979, 4-17

1979 IBM CAMBRIDGE SCIENTIFIC CENTER OUTSIDE PUBLICATIONS

R. A. MacKINNON, The Changing Virtual Machine Environment: Interfaces to Real Hardware, Virtual Hardware, and Other Virtual Machines, IBM Systems Journal, Vol. 18, No. 1, 1979, 18-46

L. H. HOLLEY, R. P. PARMELEE, C. A. SALISBURY, D. N. SAUL, VM/370 Asymetric Multiprocessing, IBM Systems Journal, Vol. 18, No. 1, 1979, 47-70

IBM

Cambridge Scientific Center, 545 Technology Square, Cambridge, Massachusetts 02139