AD-A031 451 FLORIDA UNIV GAINESVILLE DEPT OF COMPUTER AND INFORM--ETC F/G 9/2
PRELIMINARY SPECIFICATION OF REAL-TIME PASCAL.(U)
JUL 76 G J HANSEN, C E LINDAHL N61339-76-C-0017

UNCLASSIFIED NAVTRAEQUIPCEN-76-C-0017-1 NL

1 OF 1
AD
A031451

END
DATE
FILMED
12-76

AD A031451

TECHNICAL REPORT NAVTRAEQUIPCEN 76-C-0017-1

PRELIMINARY SPECIFICATION FOR REAL-TIME PASCAL

Department of Computer and Information Sciences,
University of Florida
512 Weil Hall
Gainesville, Florida 32611

July 1976

Final Report for period October 1975 through July 1976

DoD Distribution Statement

Approved for public release:
distribution unlimited.

Prepared for

NAVAL TRAINING EQUIPMENT CENTER
Orlando, Florida 32813

NAVAL TRAINING EQUIPMENT CENTER
ORLANDO, FLORIDA 32813

NAVTRAEQUIPCEN 76-C-0017-1

GOVERNMENT RIGHTS IN DATA STATEMENT

Reproduction of this publication in whole or in part is permitted for any purpose of the United States Government.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 76-C-0017-1	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) PRELIMINARY SPECIFICATION OF REAL-TIME PASCAL		5. TYPE OF REPORT & PERIOD COVERED Final Report, Oct 1975 thru Jul 1976
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Gilbert J. Hansen and Charles E. Lindahl		8. CONTRACT OR GRANT NUMBER(s) N61339-76-C-0017
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Florida, Dept. of Computer & Information Sciences, Gainesville, Florida 32611		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 5741-02
11. CONTROLLING OFFICE NAME AND ADDRESS Computer Laboratory, Naval Training Equipment Center, Orlando, Florida 32813		12. REPORT DATE Jul 1976
		13. NUMBER OF PAGES 85
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution is unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

High level language	PASCAL	Portability
Concurrent processes	Kernel	Microprogramable Machine
Monitors	Scope	Reliability
Abstract data types	Directly Executable Language	Security
Scheduling	Real-time System	Structured programming

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report is the preliminary specification of the PASCAL based language, Real-Time PASCAL, which is designed to meet the requirements of real-time training device applications. The language is an amalgamation of ideas that for the most part have already been proved to be effective and useful. Modifications and extensions were incorporated in a manner consistent with the style of PASCAL. They mainly consist of the introduction of concurrent processes and constructs to control their creation, termination, scheduling and synchronization; real-time control; and different primitives for perform-

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

(6) (9) (10) (15) NEW (11) (12) 86p. (18) NAVTRAEQUIPCEN (19) 76-C-0017-1

409899

Block 20

ing I/O. The language features permit a user to program a stand alone system assuming no underlying operating system support. Requisite operations system features can be built upon the kernel of real-time PASCAL which performs process management, memory management, I/O management, real-time control, and gives exclusive access to monitors. Further research effort is recommended to develop better language facilities for I/O, exceptional conditions, allocation of processors to concurrent processes and protection mechanisms. Also presented are the development phases to be carried out to efficiently implement and improve the language.

ACCESSION for
NTIS White Section ☑
DDC Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

SUMMARY

This report presents the preliminary specification of a high-level language designed to support all the requirements of real-time, training device applications. The features of this language are based upon the requirements of real-time systems and the latest software technology. They were determined after carrying out an analysis of actual training device programs and a careful study of significant literature pertaining to the area. An attempt was made to specify a language which is transportable, reliable and secure, simple, and readable.

The motivation for designing such a language was to attempt to reduce the total programming costs - development, maintenance, and modification - which now exceed hardware costs by an order of magnitude.

The principal result is the specification itself. Further research is recommended to develop and refine the language constructs necessary to perform input/output, handle exceptional conditions, control the allocation of processors to concurrent processes, and provide protection mechanisms using capabilities (access rights).

PREFACE

In the development of the preliminary language specification presented in this report a large number of references were read, and a careful study was made of those most pertinent to the area of programming language design. As time passed and the language goals and design criteria became firm, it was evident that one language stood out above all others: PASCAL developed originally by Niklaus Wirth. In recognition of this fact and to indicate our indebtedness to PASCAL, the authors have chosen to call the language specified herein, "Real-Time PASCAL." However, other languages were of significant import, namely, Concurrent PASCAL, PLATON, ALGOL 68, and SIMULA.

Generally, only tested language facilities were included. It is hoped that the language specification will provide a detailed frame of reference for future discussion and that it adequately addresses the problems inherent in real-time systems.

TABLE OF CONTENTS

Section		Page
	SUMMARY	1
	PREFACE	2
I	INTRODUCTION	5
	Contract Objectives	5
	Design Philosophy and Background Information	5
	Design Goals and Criteria	7
II	OVERVIEW OF THE LANGUAGE	9
III	RECOMMENDATIONS	14
	Appendix A (Real-Time PASCAL Language Specification)	19
	1. Introduction	20
	2. Notation	21
	2.1 BNF (Backus-Naur Form)	21
	2.2 Syntax Diagrams	21
	3. Character Set	22
	4. Basic Symbols	23
	4.1 Special Symbols	23
	4.2 Keyword Symbols	23
	4.3 Identifiers	23
	4.4 Constants	24
	4.5 Separators	24
	5. Blocks	25
	6. Label Declarations	26
	7. Constant Definitions	27
	8. Data Type Definitions	28
	8.1 Simple Types	28
	8.1.1 Enumeration Types	29
	8.1.1.1 INTEGER Type	30
	8.1.1.2 BOOLEAN Type	30
	8.1.1.3 CHAR Type	31
	8.1.1.4 Scalar Type	31
	8.1.1.5 Subrange Type	32
	8.1.2 REAL Type	32
	8.1.3 QUEUE Type	34
	8.1.4 REF Type	35

Page

8.2 Structured Types 35
8.2.1 ARRAY Type 36
8.2.2 RECORD Type 38
8.2.3 SET Type 40
8.2.4 Pointer Type 40
8.2.5 MONITOR Type 41
9. Variables 44
9.1 Variable Declaration 44
9.2 Variable Denotations 44
9.2.1 Entire Variable 45
9.2.2 Array Component 45
9.2.3 Record Component 45
9.2.4 Pointer Component 46
10. Expressions 47
10.1 Type Compatibility 49
10.2 Operators 49
10.2.1 NOT Operator 49
10.2.2 Multiplying Operators 49
10.2.3 Adding Operators 50
10.2.4 Relational Operators 50
10.3 Sets 51
11. Statements 52
11.1 Assignment Statement 53
11.2 Compound Statement 53
11.3 GOTO Statement 54
11.4 IF Statement 54
11.5 CASE Statement 55
11.6 FOR Statement 55
11.7 LOOP-WHILE Statement 57
11.8 UNTIL Statement 58
11.9 WITH Statement 59
11.10 Routine Calls 60
11.10.1 Procedure Call 61
11.10.2 Function Call 62
11.10.3 Event Call 62
11.11 INIT Statement 63
11.12 TERMINATE Statement 63
12. Routine Declarations 64
12.1 Procedure Declaration 66
12.2 Function Declaration 67
12.3 Event Declaration 68
13. Process Declaration 70
14. Concurrent Program 72
15. Scope Rules 75
16. Input/Output 77
17. Real-Time Control 81

SECTION I

INTRODUCTION

CONTRACT OBJECTIVES

The objectives of the current phase of this contract and this report are respectively to develop and to present a preliminary specification for a high-level computer language designed to support all the requirements of real-time, training device applications. The features of this language are based upon the requirements of currently operational simulators and the latest software technology. They were determined after carrying out an analysis of actual training device programs and a careful study of significant literature pertaining to the area.

DESIGN PHILOSOPHY AND BACKGROUND INFORMATION

First and foremost the specified high-level language must address the problems inherent in what might be called "hard" real-time situations, that is, circumstances in which responses to signals are required within a few microseconds at best. Such response times obviously require efficient object code and a run-time representation of data for which highly efficient access is possible. Since both application and system programs are to be written in the specified high-level language with no degradation in response time, some technique had to be devised to solve the problem. The best answer appears to be to consolidate the code for time critical sections, for the basic real-time control facilities, for necessary primitive input/output operations, and for the allocation of central processing unit(s) to concurrent processes into a small, machine code section to be designated the "kernel" of the system. The high-level language rests upon this minimal kernel for its run-time support. High-level constructs have been provided in the language to allow direct coupling to appropriate portions of the kernel. Hence, the language specified does not depend upon the run-time environment provided by an associated operating system. Instead it is intended that the user will have the appropriate high-level language facilities supported by a minimal, primitive kernel with which to implement the requisite operating system or support resources required by an application. Thus, medium term scheduling, sophisticated input/output procedures, and custom tailored control of an external simulation environment, for example, can all be carried out in a high-level language without resorting to imbedded machine code.

In addition to requiring that efficient object code be generated by the compiler for the language, a most important goal was to attempt to achieve practical portability of the compiler from machine to machine so that the language would be as machine independent as possible. This objective has greatly influenced the language design. For example, it is planned that the compiler will be written in the specified language and that it will not

generate machine language for a particular target machine. Instead it will generate a directly executable language or DEL. This DEL represents the optimal instruction set and data types of an abstract machine for executing the specified language. Obviously realization of this abstract machine must be carried out. The best solution for efficiency would be to design a machine to execute the DEL directly or to write an emulator for a microprogrammable machine which directly executes it. Otherwise, the code generators for the compiler can be rewritten to generate optimized machine code for any target machine. Transporting the compiler from one machine to another consists, therefore, of rewriting the small kernel, and either the code generation phase of the compiler or an interpreter for the DEL.

At the present time it should be emphasized that the language specification presented herein is a "preliminary" specification. The reasons for this statement are many. In the first place, the language's compiler has not been written; hence the language has not yet been suitably tested. Secondly, in preparation of the language specification it was found that certain areas require additional research and development effort. These consist of the following: input/output, allocation of processors to concurrent processes, exceptional conditions, and protection mechanisms. Preliminary constructs have been selected, but they must stand the test of application and time before a definitive conclusion as to their efficacy can be ascertained and a definitive language specification issued.

No attempt has been made to include many "new", untested language facilities. Basically the language being defined is an amalgamation of ideas that for the most part have already been proved to be effective and useful. Those who are familiar with the current state of the art of programming languages will note that the language is not new and is easily recognizable as a variant of the PASCAL language developed originally by Niklaus Wirth[1]. The high-level languages from which most of the constructs have been taken include PASCAL[2], Sequential and Concurrent PASCAL[3,4], ALGOL 68[5], PLATON[6] and SIMULA[7]. Since the basis of the specified language is

1. Wirth, N. "The Programming Language PASCAL and its Design Criteria" in State of the Art Lecture Report 7: High Level Languages, Infotech Information, Maidenhead, Berks England (1972), 451-473.
2. Jensen, K. and Wirth, N. PASCAL User Manual and Report, Springer-Verlag, Berlin (1974).
3. Hansen, P.B. and Hartmann, A.C. Sequential PASCAL Report, Information Science, California Institute of Technology (July 1975).
4. Hansen, P.B. Concurrent PASCAL Report, Information Science, California Institute of Technology (July 1975).
5. Lindsey, C.H. and Van der Meulen, S.G. Informal Introduction to ALGOL 68, North-Holland, Amsterdam, The Netherlands (1971).

essentially PASCAL, it is proposed that this language be called Real-Time PASCAL. It has been devised to address the problems inherent in the design and construction of simulation, process control, and computer operating systems in the context of a real-time environment.

DESIGN GOALS AND CRITERIA

The overall objective was to specify a language which was transportable, reliable and secure, flexible, simple, and readable. Obviously many compromises have had to be made, for some of the above goals conflict with each other. Let us now consider the design goals and their associated criteria in order to better understand the compromises and the specification itself.

TRANSPORTABILITY. As indicated above, practical transportability of the compiler was one of the important goals to be achieved. This has required that the compiler be written in the specified language, that it be parameterized with respect to the storage units utilized for each data type, that the language require minimal run-time support, and that the run-time support package called the "kernel" of the language be the only code written in the assembly language of the given target machine.

RELIABILITY AND SECURITY. Reliable and secure software systems require that the language utilized have the facilities to protect or limit access to certain program and/or data elements, in some cases as a function of time. In order to meet this goal the following criteria were established. The program syntax should allow automatic and rigorous checking of data types and the interfaces between program modules at compile time. Constructs should be provided to detect and hence guarantee at compile time that time-dependent errors cannot occur when concurrent processes are operating. The sharing and exchange of data between processes should be aided by providing the language facilities to restrict the scope of the access to the variables involved. To aid the compiler all variables and their associated data types should be declared with no default conditions allowed.

FLEXIBILITY. The programming language should have the flexibility to build abstractions that are natural for the problem at hand. This implies that the data types of the language should provide a convenient match to the abstractions of the problem. Therefore, the language should allow a programmer

---

6. Sorensen, S.M. and Staunstrup, J. PLATON Refernece Manual, RECAU, Aarhus, Denmark (July 1975).
7. Ichbrah, J.D. and Morse, S.P. "General Concepts of the SIMULA 67 Programming Language", Annual Review in Automatic Programming, 7, 1 (1972), 65-93.

to define easily new data type and that these can be defined in terms of other types previously defined. Only with this capability can the programmer efficiently match a given problem with the algorithm and the computer hardware used to solve it.

SIMPLICITY. A programming language should be as simple as possible and yet retain sufficient power to express easily the algorithms required in real-time systems. This goal has resulted in the following criteria. The syntax of the language should be natural for the problem and encourage clear thinking; it should be designed so that it facilitates fast translation; and it should be such that the compiler can generate efficient object code without the need for extensive code optimization.

READABILITY. A reasonable goal is that the programs written in the language can be easily read and understood. This goal has resulted in the following criteria being applied. The language structures used should encourage lucid thinking and should be natural to use. In addition it implies that the extent of specific language constructs be clearly delineated.

SECTION II

OVERVIEW OF THE LANGUAGE

A Real-Time PASCAL program consists of statements which describe the operations to be performed on data, and declarations and definitions which describe the data which are manipulated by the operations.

Statements denote operations on constants and variables. An identifier may be a synonym for a constant through an association introduced in a constant definition. Variables occurring in a statement must be introduced by a variable declaration which associates an identifier naming the variable with a data type. The data type defines the set of values the variable may assume and possibly operations that may be performed on the variable. The data type may be either directly described in the variable declaration, or it may be referenced by a type identifier previously introduced by a type definition.

A data type is either simple or structured. The simple types consist of the enumeration, REAL, queue and reference types. The enumeration type defines a linear ordered set of distinct values. The values may be defined by identifiers or by one of the standard enumeration types: BOOLEAN, INTEGER or CHAR. Values of type BOOLEAN are denoted by the identifiers TRUE and FALSE, values of type INTEGER and REAL are denoted by numbers, and values of type CHAR are denoted by quotations. The set of values of type CHAR is the character set defined on the object machine.

A type may be defined as a subrange of any other already defined enumeration type be indicating the smallest and largest value of the subrange.

A queue type can only be used within a monitor entry routine (see below) to delay and resume the execution of a calling process. A reference type is used to reference an instance of a process.

Structured types specify the types of their components and a structuring method. A component of a variable of structured type is denoted by a selector. The structured types consist of array, record, set, pointer and monitor types.

An array consists of a fixed number of components of the same component type. An array selector consists of computable indices of the index type specified in the array type definition. The index type must be an enumeration type. The time to select an array component is independent of the indices.

A record consists of a fixed number of fields which may be of different types. A record selector is not computable, but an identifier uniquely denoting the

component to be selected. These field identifiers are declared in the record type definition. As with arrays, the time needed to access a record component is independent of the selector. A record type may have several variants. Thus, variables of the same type may have different structures, i.e., contain a different number of components with different types. A component of the record, called the tag field, indicates the currently valid variant. The part common to all variants usually consists of several components, including the tag field.

A set defines the set of all subsets of values of its base type. The base type must be an enumeration type.

A variable may either be declared explicitly in which case it is referenced by its identifier, or generated dynamically by an executable statement in which case it is referenced through a pointer. Pointers are variables of pointer type. The pointer type is bound to the given component type so a pointer variable may only assume values pointing to variables of the same component type. The pointer constant NIL is an element of every pointer type; it points to no variable. The use of pointers permits any finite graph to be represented.

A monitor type defines a data structure and the operations that can be performed on the data structure by concurrent processes. The monitor can be used for process communication, synchronization and scheduling. Variables declared within a monitor are accessible only through the monitor's entry routines. Monitor entry routines are accessible outside the monitor type, but not within it, only to a process or another monitor. Simultaneous calls on monitor entry routines by concurrent processes are executed one at a time. Thus a monitor entry routine has exclusive access to variables declared in a monitor type. A monitor type also defines an initial statement that will be executed when a monitor variable is initialized. A system can only have a fixed number of monitors.

One of the basic operations on a variable is assignment of a new value to it by an assignment statement. The value is obtained by evaluating an expression. An expression consists of operands (variables, constants, sets or functions) and operators in infix notation. The data types of two operands must be compatible in order for an operation to be performed on them. The fixed set of operators are:

a. arithmetic operators: addition, subtraction, sign inversion, multiplication, division, and remainder.
b. Boolean operators: negation, conjunction (and), and disjunction (or).
c. set operators: union, intersection and difference.
d. relational operators: equality, inequality, ordering, set membership and set inclusion.

Statements are either simple or structured. The simple statements are the assignment statement, procedure call, event call, GOTO statement, INIT statement and terminate statement. A procedure call causes the execution of the designed procedure (see below). An event call causes an event (see below) to be invoked. A GOTO statement breaks the normal sequential execution of statements by causing the next statement to be executed to be the one labeled with the specified label. The label must be declared in the block containing the GOTO statement, and a statment marked with the label within the same block. The INIT statement causes a process or monitor to be initialized. For a process, an instance is created and its statements executed concurrently with all other processes. For a monitor, its initial statement is executed. A monitor can only be initialized once within the process it is declared. A terminate statement terminates the execution of the specified process instance.

Simple statements are components of structured statements which specify sequential, selective or repeated execution of their components. Sequential execution of statements is specified by the compound statement and WITH statement. Selective execution is specified by the IF statement and the CASE statement. Repeated execution is specified by the FOR statement, the LOOP statement and the UNTIL statement.

A compound statement and WITH statement group a sequence of statements into a unit. The statements are executed sequentially in the same order as they are written. The WITH statement permits record fields and monitor entry routines to be used without qualification of the record or monitor variable within the statement group.

The IF statement selects for execution one of two statements depending on the value of a Boolean expression. The second statement is optional. The CASE statement allows for the selection of one of several statements according to the value of an enumeration expression.

The FOR statement is used when the number of iterations is known beforehand. The loop statement is used to execute certain statements repeatedly while a Boolean expression remains true. The UNTIL statement is used to execute a structured statement until one of the designate events bound to it is invoked.

An identifier can be associated with a statement. Such a statement is called a routine and the declaration a routine declaration. There are three kinds of routines: procedures, functions and events. They are structurally identical. A routine may contain additional variable declarations,

type definitions and routine declarations. The scope of these identifiers is the program text which constitutes the routine declaration. A routine has a fixed number of (formal) parameters which are denoted within the routine by an identifier. There are five kinds of parameters: variable parameters, constant parameters, universal parameters, and procedure and function parameters. Upon an activation of a routine, an actual parameter known as an argument is substituted for the corresponding parameter. For a variable parameter the arguments must be a variable. The parameter stands for this argument and may be assigned a value within the routine. For a constant parameter the argument must be an expression which is evaluated once before execution of the procedure and assigned to the parameter. Its value cannot be changed by the routine. For a universal parameter, compatibility checking of its type and the arguments type is suppressed. In the case of procedure and function parameters, the arguments must be a procedure or function identifier.

A function is declared like a procedure except the declaration specifies a result type which must be an enumeration or pointer type. The functional result is defined by assigning a value to the function identifier within the function declaration. Function parameters must be constant parameters.

A function or procedure may be called recursively by a function or procedure call within its declaration. A procedure or function declaration prefixed with the symbol ENTRY is known as an entry routine. An entry routine declaration may only occur within a monitor type and cannot be nested within another routine declaration.

An event declaration represents an event that is bound to a structured statement by an UNTIL statement. An event call invokes the event and upon completion causes control to return to the statement after the UNTIL statement instead of to the point after the call.

A process declaration is the prototype for a class of sequential processes. An instance of the process is created by the INIT statement. Execution of the sequential processes is concurrent and under control of the kernel. Concurrent processes communicate only by means of monitors.

The kernel represents the minimal run-time support needed to support the execution of a concurrent program. An underlying operating system is not required. The kernel permits a user to construct the requisite operating system facilities and policies needed to support his stand alone system.

Basic I/O is handled by a standard procedure. It is a primitive operation that permits a user to construct various device handlers, interrupt handlers,

I/O exception handling routines, generalized I/O and control routines, and file systems.

Real-time control is provided by standard routines that permit a process to be delayed a specified amount of time and to measure the time for a process to perform some action.

The complete language specification is contained in Appendix A.

SECTION III

RECOMMENDATIONS

The development of the preliminary language specification presented in this report has established the realization that much yet remains to be accomplished. In particular, a compiler for the language specified must be written, and the language features must be evaluated by the programming of an actual, real-time system. Only in this manner can the language's adequacy and capabilities be suitably tested and its language specification definitely ascertained.

It is recommended that research and development effort be directed toward a better understanding of high-level language facilities pertaining to input/output, the handling of exceptional conditions, the control of allocation of processors to concurrent processes, and the provision of appropriate protection mechanisms using capabilities (access rights).

In addition, if the preliminary language specification presented herein is to form the basis for a continued development effort, it is recommended that the following steps or development phases be carried out.

The first phase should be the implementation of the compiler. Since portability of the compiler from machine to machine is of importance, it is envisioned that this compiler would output a directly executable language or DEL instead of machine language for a specific computer. The DEL represents the optimal instruction set and data types of an abstract machine for executing the specified language. This phase would also include the specification of the DEL.

Phase II should be devoted to developing the means for executing the DEL. This could be accomplished by writing an emulator on a microprogrammable machine (perhaps the best method), an interpreter on a specific computer, or by using the DEL as input to a program to generate machine code directly for a target machine.

Once the compiler and a run-time system have been developed for the language it must be adequately tested, evaluated, and changed if necessary in Phase III. Initially algorithms pertaining to real-time applications should be written and tested to determine the effectiveness and suitability of the language. Finally an actual prototype, real-time system should be programmed as an exercise. With these tests as a basis it is envisioned that modification and extensions to the language would be incorporated as needed.

Once the language constructs have been finally determined, Phase IV should be devoted to developing the most efficient realization of the abstract machine. Both software and hardware techniques should be considered. Perhaps

additional hardware processors (microprocessors) should be considered.

In summary, the goal should always be kept in mind, namely to support all the requirements of real-time training device systems in the most cost-effective manner.

BIBLIOGRAPHY

Programming Language Design

1. Enslow, P.H. et. al., "Implementation Languages for Real -Time Systems. Part 2. Language Design--General Comments", European Research Office London (England), (April 1975).

2. Hoare, C.A.R. Hints on Programming Language Design. Stanford University Report STAN-CS-73-403 (Dec. 1973).

3. McKeeman, W.M. "Programming Language Design" in Compiler Construction: An Advanced Course, Springer-Verlag, Berlin (1974), 514-524.

4. Pratt, T.W. Programming Languages: Design and Implementation, Prentice-Hall, Englewood Cliffs, N.J. (1975).

5. Wasserman, A.I. (editor) Special SIGPLAN Notices Issue on Programming Language Design 10, 7 (July 1975).

6. Wirth, N. "The Programming Language PASCAL and its Design Criteria" in State of the Art Lecture Report 7: High Level Languages, Infotech Information, Maidenhead, Berks England (1972), 451-473.

7. Wirth, N. "On the Design of Programming Languages" in Information Processing 74, North Holland, Amsterdam (1974), 386-393.

Programming Languages

8. Bekic, H. "An Introduction to Algol 68", Annual Review in Automatic Programming, 7, 3 (1973), 143-170.

9. Department of Defense, Requirements for High Level Computer Programming Languages "TINMAN" (April 1976).

10. Enslow, P.H. et. al. "Implementation Languages for Real-Time Systems. Part 3. Command and Control Languages--Specific Comments", European Research Office London, England (April 1975).

11. Hansen, P.B. Concurrent PASCAL Introduction, Information Science, California Institute of Technology (March 1975).

12. Hansen, P.B. Concurrent PASCAL Report, Information Science, California Institute of Technology (June 1975).

13. Hansen, P.B. and Hartmann, A.C. Sequential PASCAL Report, Information Science, California Institute of Technology (July 1975).

14. Ichbrah, J.D. and Morse, S.P. "General Concepts of the SIMULA 67 Programming Language", Annual Review in Automatic Programming, 7, 1 (1972), 65-93.

15. Jensen, K. and Wirth, N. PASCAL User Manual and Report, Springer-Verlag, Berlin (1974).

16. Lindsey, C.H. and Van der Maulen, S.G. Informal Introduction to ALGOL 68, North-Holland, Amsterdam, The Netherlands (1971).

17. Miller, J.S. et. al., CS-4 Language Reference Manual, Intermetrics, Inc., Cambridge, Mass. (Dec. 1973).

18. Ministry of Defense, Official Definition of CORAL 66, Her Majesty's Stationery Office, London (1970).

19. Newbold, P. HAS/S Language Specification, Intermetrics, Inc., Cambridge, Mass. (July 1974).

20. Palme, J. Making SIMULA into a Programming Language for Real-Time, Research Institute of National Defense, Sweden (June 1974).

21. Sorensen, S.M. and Staunstrup, J. PLATON Reference Manual, RECAU, Aarhus, Denmark (July 1975).

Operating System Techniques

22. Dijkstra, E.W. "Hierarchical Ordering of Sequential Processes" in Operating System Techniques, Academic Press, New York (1972).

23. Goos, G. "Some Basic Principles in Structuring Operating Systems" in Operating System Techniques, Academic Press, New York (1972).

24. Hansen, P.B. "The Nucleus of a Multiprogramming System", CACM 13, 4 (April 1970), 238-241.

25. Hansen, P.B. Operating System Principles, Prentice-Hall, Englewood Cliffs, N.J. (1973).

26. Hoare, C.A.R. "Monitors: An Operating System Structuring Concept", CACM 17, 10 (Oct. 1974), 549-557.

27. Wulf, W. et. al. "HYDRA: The Kernel of a Multiprocessor Operating System", CACM 17, 6 (June 1974), 337-345.

Concurrency

28. Dijkstra, E.W. "Cooperating Sequential Processes" in Programming Languages (ed. F. Genuys), Academic Press, New York (1968).

29. Hansen, P.B. "Structured Multiprogramming", CACM 15, 7 (July 1972), 574-577.

30. Hansen, P.B. "A Comparison of Two Synchronizing Concepts", Acta Informatica 1 (1972), 190-199.

31. Hansen, P.B. "Concurrent Programming Concepts", Computing Surveys 5, 4 (Dec. 1973), 223-245.

32. Hoare, C.A.R. "Towards a Theory of Parallel Programming" in Operating System Techniques, Academic Press, New York (1972).

Structured Programming

33. Dahl, O.J. "Hierarchiacal Program Structures" in Structured Programming by Dahl, O.J., Dijkstra, E.W. and Hoare, C.A.R., Academic Press, New York (1972).

34. Knuth, D.E. "Structured Programming with GOTO Statements," Computing Surveys 6, 4 (Dec. 1974), 261-302.

35. Ledgard, H.F. and Marcotty, M. "A Genealogy of Control Structures", CACM 18, 11 (Nov. 1975), 629-638.

36. Wirth, N. Systematic Programming, Prentice-Hall, Englewood Cliffs, N.J. (1973).

Portability

37. Poole, P.C. "Portable and Adaptable Compilers" in Compiler Construction: An Advanced Course, Springer-Verlag, Berlin (1974), 427-497.

Exceptional Conditions

38. Goodenough, J.B. "Exception Handling: Issues and a Proposed Notation", CACM 18, 12 (Dec. 1975), 683-696.

APPENDIX A

Real-Time PASCAL Language Specification

1. INTRODUCTION

Real-Time PASCAL is a language designed to meet the requirements of real-time training device applications. It is mainly based on PASCAL developed by Niklaus Wirth [6] and Concurrent PASCAL developed by Per Brinch Hansen [11, 12]. Ideas and constructs were also taken from the programming languages ALGOL 68 [15], SIMULA [13] and PLATON [21].

There are a number of modifications and extensions incorporated into the language in a manner consistent with the style of PASCAL. The main extensions consist of the introduction of: 1) concurrent processes and constructs to control their creation, termination, scheduling and synchronization; 2) real-time control, and 3) control structures consistent with the philosophy of structured programming. The main modifications were: 1) addition of delimiters to terminate statements; 2) change in the scope rules, and 3) different primitives for performing I/O.

The features of the language permit a user to program a stand alone system assuming no underlying operating system support. From the language constructs the user can program the requisite operating system features needed to support his system. The users operating system is built upon the kernel of Real-time PASCAL. This kernel controls: 1) the allocation of concurrent processes to processors, 2) the exclusive access of concurrent processes to shared data, 3) the peripherals, and 4) interrupts.

It is recognized that there are some technological problems that have to be overcome in order to obtain the efficiency needed for real-time systems. Some of the solutions are just now becoming available, e.g., hardware to permit efficient access to local and global variables and microprogrammable machines the user can microprogram. Other solutions are not readily available. To overcome these inefficiencies, modification may have to be made to certain language constructs, e.g., require the user to give an upper bound on the number of simultaneous activations of a given concurrent process that can exist simultaneously, and require the user to give an upper bound on the storage requirements for a processe's stack.

2. NOTATION

The syntax of Real-time PASCAL constructs is given both in BNF and in the form of syntax diagrams.

2.1 BNF (Backus-Naur Form)

Syntactic constructs are denoted by metalinguistic variables enclosed between angular bracks < and >. The metalinguistic variables are English words which are suggestive of the meaning of the construct. A sequence of constructs enclosed by the meta-brackets { and } imply their repetition zero or more times. The symbol <empty> denotes the null sequence of symbols.

Terminal symbols constitute the vocabulary of Real-time PASCAL, i.e., its basic symbols (cf. Section 4). They are represented by capital letters and special characters.

Example:

<compound statement> ::= BEGIN <statement> {;<statement>} END

2.2 Syntax Diagrams

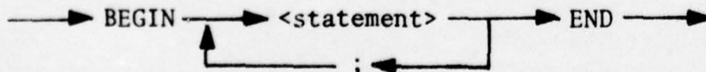
The BNF specifies the language syntax whereas the syntax diagrams are designed for human readability. They are not designed as a basis for parsing, e.g., the syntax diagrams for expressions do not reflect the precedence of operators.

A syntax diagram is a directed graph with a single input edge and a single output edge. It is a graphical representation of a syntax rule. A traversal of a syntax diagram, starting at the input edge and terminating at the output edge, corresponds to an application of the syntax rule. To each metalinguistic variable corresponds a syntax diagram.

The nodes of a syntax diagram are either metalinguistic variables or terminal symbols.

Example:

compound statement:



3. CHARACTER SET

Programs are written in a subset of the Extended ASCII character set.

<character> ::= <graphic character> | <control character>

<graphic character> ::= <special character> | <letter> | <digit> |
<space>

<special character> ::= + | - | * | / | " | . | , | ; | : | = | ' | @ |
< | > | (|) | [|] | { | } | ↑

<letter> ::= A | B | C | ... | X | Y | Z | _

<digit> ::= 0 | 1 | ... | 8 | 9

Letters are used for forming identifiers (cf. Section 4.3) and strings (cf. Section 8.2.1). Digits are used for forming numbers (cf. Section 4.4), identifiers and strings.

<control character> ::= (: <digits> :)
<digits> ::= <digit>{<digit>}

A control character is an unprintable character.

Each character is represented by its ordinal value (cf. Section 8.1.1.1). The ordinal value of a control character must be in the range 0..127.

4. BASIC SYMBOLS

A program consists of symbols and separators.

<symbol> ::= <special symbols> | <keyword symbol> | <identifiers> | <constant>

4.1 Special Symbols

<special symbol> ::= + | - | * | / | = | <> | < | <= | >= | > | ↑ | := | . | ; | : | , | (|) | [|] | { | } | ↑ | .. | ' | " | (: | :) | 10

Special symbols have fixed meanings which will be given in the appropriate section.

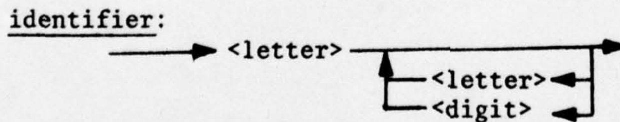
4.2 Keyword Symbols

<keyword symbol> ::= AND | ARRAY | BEGIN | CASE | CONST | DIV | DO | DOWNTO | ELSE | END | EVENT | FI | FOR | FORWARD | FUNCTION | GOTO | IF | IN | INIT | LABEL | LOOP | MOD | MONITOR | NEXT | NIL | NOT | OF | OR | PACKED | PROCEDURE | PROCESS | PROGRAM | RECORD | REPEAT | REF | SET | SHARED | TERMINATE | THEN | TO | TYPE | UNIV | UNTIL | VAR | WHILE | WITH

Keyword symbols are reserved words with a fixed meaning; they may not be used as identifiers. They are written as a sequence of letters and are interpreted as a single symbol.

4.3 Identifiers

<identifiers> ::= <letter>{<letter> | <digit>}



Identifiers are names denoting constants, types, variables, procedures, functions, and events.

4.4 Constants

A constant represents a value that can be used as an operand in an expression.

<constant> ::= <constant identifier> | <enumeration constant> |
<real constant> | <string constant> | NIL

Each type of constant will be discussed in the appropriate section.

4.5 Separators

At least one separator must occur between any two constants, identifiers, or keyword symbols, and no separator may occur within such.

<separator> ::= <space> | <end of line> | <comment>

<comment> ::= <left curly bracket> <any sequence of graphic characters not containing}> <right curly bracket>
<left curly bracket> ::= {
<right curly bracket ::= }

A comment may be removed from the program text without altering its meaning.

5. BLOCKS

A block is the basic program unit. It consists of declarations of computational objects and a compound statement that operates on them.

<block> ::= <declarations><compound statement>

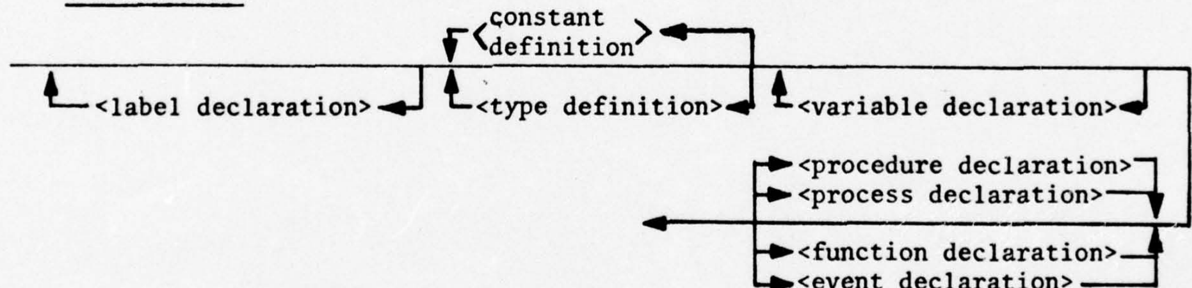
Declarations serve to define a label, constant, type, variable, procedure, function and event, and associate with them an identifier (except for labels which are integers). The order of the declarations is prescribed: label declarations, constant and type definitions, variable declarations, and procedure and function declarations.

All computational objects must be declared before they are referenced except:

1) for a type identifier in a pointer type definition (cf. Section 8.2.4),
2) when there is a forward reference in a procedure or function call (cf. Sections 11.10 and 12).

<declarations> ::= {<label declaration>}{<constant or type definition>}
{<variable declaration>}{<routine declaration>}
<constant or type definition> ::= <constant definition> | <type definition>
<routine declaration> ::= <procedure declaration> | <function declaration> |
<event declaration>

declarations:



A compound statement is a sequence of statements separated by semicolons and enclosed by the delimiters BEGIN and END (cf. Section 11.2).

The structure of a program (cf. Section 14), routine (cf. Section 12), process (cf. Section 13); and monitor (cf. Section 8.2.5) each consist of a heading and a block. Thus there are no anonymous blocks (as in ALGOL or PL/I). However, blocks may be nested since routine and system type declarations may be nested (cf. Section 15 for the scope rules for names declared in a block).

6. LABEL DECLARATIONS

A label declaration serves to list all labels defined in a block. Any statement in a block may be marked by prefixing the statement with a label followed by a colon. A label is defined to be an unsigned integer.

```
<label declaration> ::= LABEL <label>{,<label>};
<label> ::= <unsigned integer>
<unsigned integer> ::= <digit>{<digit>}
```

label declaration:

```
----> LABEL --+--> <unsigned integer> --+--> ; ---->
              ^                         |
              +----------- , -----------+
```

Example:

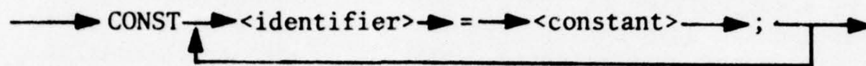
```
LABEL 1, 15;
```

7. CONSTANT DEFINITIONS

A constant definition introduces an identifier as a synonym to a constant.

<constant definition> ::= CONST <identifier>=<constant>;{<identifier>=
<constant>;}

constant definition:



Example:

```
CONST PI=3.1415927; N=20;
```

8. DATA TYPE DEFINITIONS

A data type defines the set of values a variable may assume. Each variable can be associated with one and only one type. All types are built from simple types. Variables of simple type can only be operated on as a whole. The simple types consist of the enumeration types, real, queue and reference type.

A structured type is a composition of other types. It is characterized by its components types and its structuring method. A component of a variable of structured type is denoted by a selector. The structured types consist of array, record, set, pointer and monitor types.

Monitor, queue and reference types are active types; all other types are passive types.

Data types of two operands must be compatible in order for an operation to be performed on them (cf. Section 10.1).

<type definition> ::= TYPE <identifier>=<type>;{<identifier>=<type>;}
<type> ::= <simple> | <structured type>

type definition:

→ TYPE → <identifier> → = → <type> → ; →

8.1 Simple Types

A simple type is either defined by the programmer or is one of the standard types: INTEGER, REAL, BOOLEAN, CHAR, QUEUE, or REF.

<simple type> ::= <enumeration type> | REAL | <type identifier> |
<queue type> | <reference type>

8.1.1 Enumeration Types

An enumeration type is characterized by the set of its distinct values, upon which a linear ordering is defined.

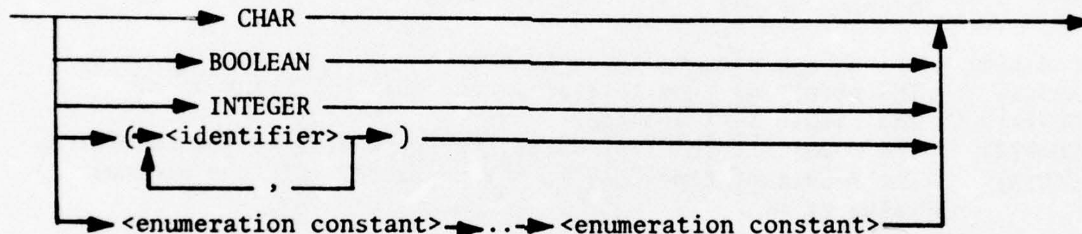
<enumeration type> ::= INTEGER | BOOLEAN | CHAR | <scaler type> |
<subrange type>

<scalar type> ::= (<identifier>{,<identifier>})

<subrange type> ::= <enumeration constant>..<enumeration constant>

<enumeration constant> ::= <identifier> | <character constant> |
<boolean constant> | <integer constant>

enumeration type:



The basic operators for variables of enumeration type are assignment (:=) and the relational operators (<, =, >, <=, <>, >=). The standard functions applying to enumeration types are:

SUCC(X) The succeedor value of X in the enumeration (if it exists).
PRED(X) The predecessor value of X in the enumeration (if it exists).

An enumeration value can be used to select one of several statements for execution (cf. Section 11.5). An enumeration type can be used to execute a statement repeatedly for a subrange of the enumeration values (cf. Section 11.6).

8.1.1.1 INTEGER Type

A value of the standard enumeration type INTEGER is an element of the implementation defined subset of whole numbers represented by integer constants.

<integer constant> ::= <digits>
<digits> ::= <digit>{<digit>}

The following operators are defined for integer operands and yield an integer value:

+	plus sign or add
-	minus sign or substract
*	multiply
DIV	divide and truncate
MOD	modulo. a mod b = a -((a DIV b)*b)

The standard functions applying to integers are:

ABS(X)	The result of type integer is the absolute value of X.
SQR(X)	The result is X squared.
CONV(X)	The result is the real value, corresponding to the integer X.
CHR(X)	The result of type CHAR is the character with the ordinal value of X.

8.1.1.2 BOOLEAN Type

A value of the standard enumeration type BOOLEAN is one of the logical truth values denoted by the predefined boolean constants.

<boolean constant> ::= FALSE | TRUE

Type Boolean is defined as:

TYPE BOOLEAN = (FALSE, TRUE)

so that FALSE < TRUE.

The following operators are defined for Boolean operands and yield a Boolean value:

AND	(logical conjunction)
OR	(logical disjunction)
NOT	(logical negation)

Each of the 16 Boolean operations can be defined using the above operators and the relational operators. For example, if p and q are Boolean's values:

p<=q	implication
p=q	equivalence
p<>q	exclusive OR

A Boolean value can be used to select one of two statements for execution (cf. Section 11.4), or to repeat the execution of a statement while a condition is true (cf. Section 11.9).

8.1.1.3 CHAR Type

A value of the standard enumeration type CHAR is an element of the ordered set of ASCII characters represented by character constants.

<character constant> ::= '<character>'

The ordering of characters is defined by their ordinal numbers which are strickly implementation dependent.

The following standard function applies to characters:

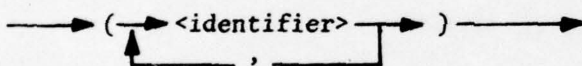
ORD(X) The result of type integer is the ordinal number of the character X in the underlying ordered character set.

8.1.1.4 Scalar Types

A scalar type defines an ordered set of values by enumeration of identifiers which denote these values.

<scaler type> ::= (<identifier>{,<identifier>})

scalar type:



The same identifier may not appear in two scalar types.

The standard function with arguments of scalar type is:

ORD(X) The result of type integer is the ordinal number of the scalar X in the underlying ordered enumeration. The ordinal number of the first identifier listed is 0.

Example:

```
TYPE PRIMARY = (RED, YELLOW, BLUE);
     SUIT = (CLUB, DIAMOND, HEART, SPADE);
     DAY = (MON, TUES, WED, THURS, FRI, SAT, SUN);
```

8.1.1.5 Subrange Type

A type may be defined as a subrange of any other already defined enumeration type by indication of the least and largest value in the subrange. The first enumeration constant specifies the lower bound, and must be less than the upper bound.

<subrange type> ::= <enumeration constant>..<enumeration constant>

Examples:

```
TYPE DIGIT = '0'..'9';
     INDEX = 0..16;
     WORKDAY = MON..FRI;
```

8.1.2 REAL Type

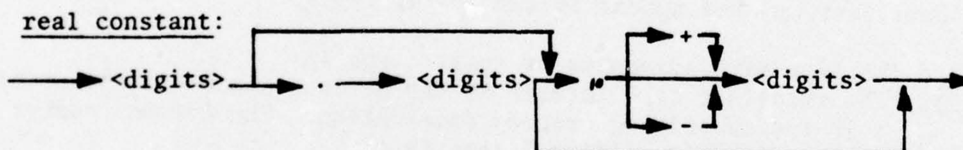
A value of the standard type real is an element of the implementation defined subset of real numbers represented by real constants.

<real constant> ::= <digits>.<digits> | <digits>.<digits>$_{10}$<scale factor> |
<digits>$_{10}$<scale factor>

<scale factor> ::= <digits> | <sign><digits>

<sign> ::= + | -

real constant:



The following operators yield a real value if at least one of the operands is of type real and the other being of type integer or real:

*	multiply
/	divide (both operands may be integers)
+	add
-	subtract
:=	assignment (the left operand must be real)

The following relational operators are defined for reals and yield a boolean value:

<	less
=	equal
>	greater
<=	less or equal
<>	not equal
>=	greater or equal

Standard functions accepting a real argument and yielding a real result are:

ABS(X)	absolute value
SQR(X)	X squared

Standard functions with a real or integer argument and a real result are:

SIN(X)	trigonometric functions
COS(X)	
ARCTAN(X)	
LN(X)	natural logarithm
EXP(X)	exponential function
SQRT(X)	square root

Standard functions with a real argument yielding integer results are:

TRUNC(X)	The result is the whole part, i.e., the fractional part is discarded.
ROUND(X)	The result is the rounded integer. = TRUNC(X + 0.5), X≥∅ = TRUNC(X - 0.5), X<∅

8.1.3 QUEUE Type

The type QUEUE may be used within a monitor type (cf. Section 8.2.5) to delay and resume the execution of a calling process within a shared routine (cf. Section 12). A queue variable does not have any stored value accessible to the program. It is either empty or non-empty. Initially it is empty. Queue variables can only be declared in a permanent variable in a monitor type.

The following standard function applies to queues:

EMPTY(X) — The result is a Boolean value defining whether or not the queue is empty.

The following standard procedures are defined for queues:

DELAY(X,P) — The calling process with priority P is delayed in the queue X and looses its exclusive access to the given monitor variables. The monitor can now be called by other processes. In order to avoid the risk of indefinite overtaking, priority should be a nondecreasing function of the time at which the delay commences.

CONTINUE(X) — The calling process returns from the monitor routine that performs the continue operations. If a process is waiting in the queue X, that process with the lowest priority immediately resumes its execution of the monitor routine that delayed it. The resumed process now again has exclusive access to the monitor variables. The continue operation is followed immediately by resumption of the delayed process. There is no possibility of an intervening call on a monitor procedure by a third process. Thus the resumed process is guaranteed it will get exclusive access to the monitor variables.

The above standard procedures and function may be called only within a monitor type.

8.1.4 REF Type

A variable of type REF is used to reference an instance of a process. The value of such a variable must be well-defined. If the variable is not a reference to a process, its value is zero.

A reference variable is assigned a well-defined value by means of the standard procedure

PROSID(X)

This procedure assigns the identity of the calling process to X. If the procedure is called within a routine, the value assigned to X is the identity of the process that called the routine. A process identity is a unique integer. During the initialization of processes, a unique consecutive integer 1,2,... is associated with a process starting with the initial process (cf. Section 14). The value of X prior to the procedure call must be zero.

8.2 Structured Types

A structured type specifies the type(s) of its components and a structuring method.

```
<structured type> ::= <unpacked structured type> | <pointer type> |
      <monitor type> | PACKED <unpacked structured type>
<unpacked structured type> ::= <array type> | <record type> | <set type>
```

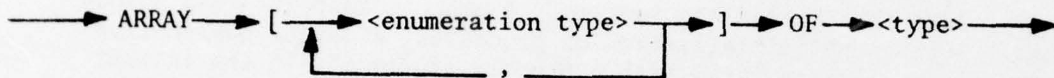
The prefix PACKED means the internal data representation of the type is economized with respect to storage at the expense of a possible loss in efficiency of access.

8.2.1 Array Type

An array consists of a fixed number of components of the same component type. Components of the array are designated by computable indices of the index type (cf. Section 9.2.2).

```
<array type> ::= ARRAY[<index type>{,<index type>}] OF <component type>
<index type> ::= <enumeration type>
<component type> ::= <type>
```

array type:



The dimension of an array is the number, n, of index types. An array component is designated by n indices. Index types are static and cannot be varied dynamically.

Example:

```
TYPE TABLE = ARRAY[0..N, 0..M] OF INTEGER;
     LATE = ARRAY[WORKDAY] OF BOOLEAN;
     HASHTABLE = ARRAY['A'..'Z'] OF PERSON;
```

The component of an array may be structured, in particular, it may be an array type. Thus, arrays are stored in row major order.

Example:

```
    TYPE = MATRIX = ARRAY[0..10] OF ARRAY[0..10] OF REAL;
and
    TYPE MATRIX = ARRAY[0..10, 0..10] OF REAL;
are equivalent.
```

String type is defined by:

```
PACKED ARRAY[1..N] OF CHAR
```

i.e., is a one-dimensional array of N characters

Example:

```
TYPE STRING = PACKED ARRAY[1..10] OF CHAR
```

Strings are represented by string constants of length N.

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

A quote mark in a string is represented by two quote marks.

The ordering of strings is determined by the ordering of the underlying character set.

The following operators apply between operands that are passive, compatible array types:

:=	assignment
=	equal
<>	not equal
<	less
>	greater
<=	less or equal
>=	greater or equal

For strings, the operands must be of the same length.

Standard functions for arrays are:

```
PACK(A, i, Z)        means  Z[j]:=A[j-u+i]   u≤j≤v
UNPACK(Z, A, i)      means  A[j-u+i]:=Z[j],  u≤j≤v
```

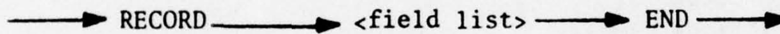
where A is a variable of type ARRAY[m..n] OF T
Z is a variable of type PACKED ARRAY[u..v] OF T
$(n-m) \geq (v-u)$

8.2.2 RECORD Type

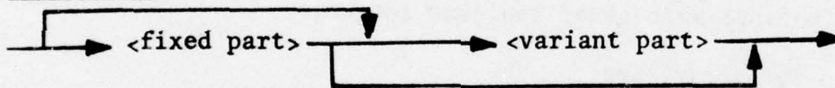
A record consists of either a fixed number of components called fields, which may be of different types, or a variant part, or both. For reference purposes each component must be given a distinct name, called the field identifier, and an associated type must be specified. Components are selected by constant field identifiers (cf. Section 9.2.3).

```
<record type> ::= RECORD <field list> END
<field list> ::= <fixed part> | <fixed part>;<variant part> | <variant part>
<fixed part> ::= <record section>{;<record section>}
<record section> ::= <field identifier>{,<field identifier>}:<type> | <empty>
<variant part> ::= CASE <tag field><type identifier> OF <variant>{;<variant>}
<variant> ::= <case label list>:(<field list>) | <empty>
<case label list> ::= <case label>{,<case label>
<case label> ::= <enumeration constant>
<tag field> ::= <identifier>: | <empty>
```

record type:

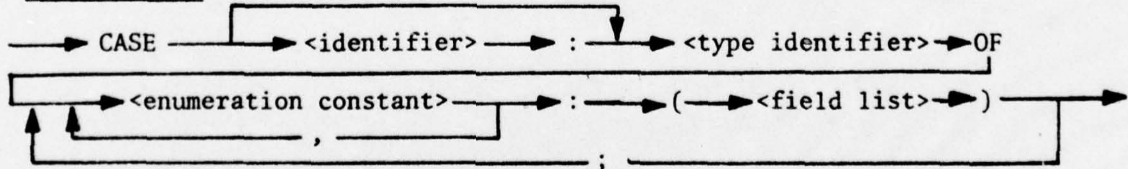


field list:



fixed part:

variant part:



A record type may have several variants. The list of components for a variant is enclosed in parenthesis and labeled by a case label which is a unique enumeration constant(s) of the type of the tag field. The tag field is a component of the record itself and indicates the currently valid variant. A component of a variant can only be referenced if the value of the tag field is equal to one of the variants case labels.

The notation for referencing a component of a record type can be abbreviated by using the WITH statement (cf. Section 11.9).

The following operators apply between operands that are passive, compatible records:

```
:=    assignment
=     equal
<>    not equal
<     less
>     greater
<=    less or equal
>=    greater or equal
```

Example:

```
TYPE STATUS = (STUDENT, FACULTY, ADMINISTRATION);
     COMPLEX = RECORD RE, IM: REAL END;
     DATA = RECORD MONTH: (JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEPT,
                           OCT, NOV, DEC);
                   DAY: 1..31;
                   YEAR: INTEGER
            END;
     PERSON = RECORD
              NAME: RECORD FIRST, LAST: PACKED ARRAY[1..10] OF CHAR;
                    END;
              SOCIALSECURITY: INTEGER;
              SEX: (MALE, FEMALE);
              BIRTH: DATE;
              COLLEGE: (AS, ENG, BUS);
              CASE S: STATUS OF
                   STUDENT: (GRADUATE: BOOLEAN;
                             YEAR: (1..7));
                   FACULTY: (TENURE: BOOLEAN;
                             RANK: (INST, ASSTPROF, ASSOCPROF, PROF));
                   ADMINISTRATION: (POSITION: (ASSTDEAN, DEAN, CHAIRMAN, OTHER);
                                    STAFF: BOOLEAN)
            END;
```

8.2.3 SET Type

A set type defines the set of all subsets of values of the base type, including the empty set. The base type must be an enumeration type.

```
<set type> ::= SET OF <base type>
<base type> ::= <enumeration type>
```

A set type offers facilities similar to a bit string.

Sets are built up from expressions of the base type (cf. Section 10.3).

The following operators apply between operands that are compatible sets:

+	set union
*	set intersection
-	set difference
<=, >=	set inclusion (contained in, contained)
:=	assignment
=	set equality
<>	set inequality
IN	set membership. The first operand must be an enumeration type and the second must be its associated set type. The result is TRUE when the left operand is an element of the right operand; otherwise FALSE.

Example:

```
TYPE COLOR = SET OF PRIMARY;
     BYTE = SET OF 0..7;
     CHARSET = SET OF CHAR;
```

8.2.4 Pointer Type

A pointer type consists of an unbounded set of values pointing to components of a given component type. The pointer type is bound to the given component type and may not be bound to any other type. The component type must be a passive type.

```
<pointer type> ::= ↑<type>
```

The operators applying to pointer operands with compatible component types are:

:=	assignment
=	equal (result is true if pointer operands are associated with the same component)
<>	not equal

The pointer constant NIL is an element of every pointer type; it points to no element at all. All pointer variables have the initial value of NIL.

New pointer values may be generated by the following standard dynamic storage allocation procedure:

NEW(P)   Allocates storage for a new component with P's component type and assigns the pointer to it to the pointer variable P.

Storage for a pointer component can be deallocated by the standard procedure:

DISPOSE(P)   Storage allocated to the component referenced by the pointer P is freed.

Examples:

```
TYPE CHAIN = ↑ NODE;
     MODE = RECORD
                VALUE: INTEGER;
                LINK: CHAIN
            END;
```

8.2.5 Monitor Type

A *monitor type defines* a data structure and the operations that can be performed on the data structure by concurrent processes. The monitor can be used to synchronize concurrent processes, transimit data between them, and schedule processes competing for shared physical resources.

The variables declared within a monitor type are accessible only within the monitor type. Only monitor entry routines have access to them. (cf. Section 12). These variables exist forever after initialization of a variable of type monitor. They are permanent variables.

The routines declared within a monitor are either simple routines or entry routines (cf. Section 12). A simple monitor routine is accessible only within the monitor while a monitor entry routine is accessible outside the monitor type (but not within it). Only a process or another monitor type may call a monitor entry routine. If concurrent processes simultaneously call monitor routines, the calls will be executed strictly one at a time. In this way a monitor entry routine has exclusive access to the permanent monitor variables while it is being executed. A process waiting to enter a monitor is delayed for a short period of time until a monitor entry routine is finished executing. This short-term scheduling of simultaneous monitor calls is handled by the virtual machine on which concurrent processes run.

However, it must also be possible to delay processes for longer periods of time because certain conditions are not satisfied (e.g., a process requesting information from an empty buffer must be delayed until another process sends more data). Monitor entry routines can control this medium-term scheduling of processes by using the queue data type (cf. Section 8.1.3). A process delayed in a queue loses its exclusive access to the permanent monitor variables until another process calls the same monitor and wakes it up again.

Monitor entry routines can call entry routines defined within other monitor types (in order to facilitate hierarchical design). However, entry procedures cannot be called recursively, either directly or indirectly.

A monitor type also defines an initial statement that will be executed when a variable of the monitor type is initialized by means of the INIT statement (cf. Section 11.11).

A variable of type monitor must be declared within a process or monitor type. It cannot be declared within a routine. Because monitor variables are declared, there are a fixed number of monitors in a system.

```
<monitor type> ::= <monitor heading><block>
<monitor heading> ::= MONITOR
```

Examples: (cf. Hoare [26])

```
          ...
CONS N = ...; M = ...; {note: M = N-1, N is the number of buffers}
TYPE PORTION = RECORD...END;

          ...
VAR BOUNDED_BUFFER : MONITOR

   VAR BUFFER: ARRAY[0..M] OF PORTION;
       LASTPOINTER:  0..M; {points to the buffer position into which
                            the next append operation will put a new
                            item}
       COUNT: 0..N; {number of filled buffers}
       NONEMPTY, NONFULL: QUEUE; {COUNT>Ø, COUNT<N respectively}
```

```
PROCEDURE ENTRY  APPEND(X:PORTION);
    BEGIN IF COUNT = N THEN  DELAY(NONFULL, 0) FI;
    BUFFER[LASTPOINTER] := X;
    LASTPOINTER := (LASTPOINTER + 1) MOD N;
    COUNT := COUNT + 1;
    CONTINUE(NONEMPTY)
    END {APPEND};
PROCEDURE ENTRY  REMOVE(VAR X:PORTION);
    BEGIN IF COUNT = 0 THEN DELAY(NONEMPTY, 0) FI;
    X := BUFFER[(LASTPOINTER - COUNT) MOD N];
    CONTINUE(NONFULL)
    END {REMOVE};
BEGIN {initial statement of monitor}
COUNT := 0; LASTPOINTER := 0
END {BOUNDED_BUFFER};
```

(cf. Section 13 for examples of concurrent processes that use the monitor BOUNDED_BUFFER)

9. VARIABLES

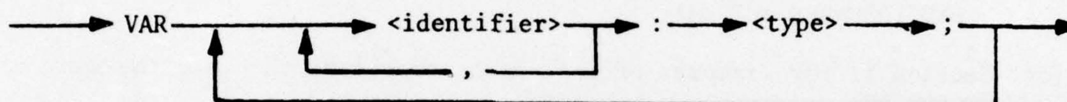
A variable is a named data structure that can contain values of a single type. The basic operations on a variable are assignment of a new value to it and a reference to its current value.

9.1 Variable Declaration

All variables must be declared in a variable declaration prior to their use.

<variable declarations> ::= VAR <variable declaration>{;<variable declaration>};
<variable declaration> ::= <identifier>{,<identifier>}:<type>

variable declarations



Examples:

```
VAR  I,J: INTEGER;
     M1, M2, M3: MATRIX;
     P1: PERSON;
     K: 1..10;
     B1, B2: BOOLEAN;
     HUE: COLOR;
     OPCODE: (ADD, SUB, MPY, DIV);
     X,Y: REAL;
     WORKWEEK: WORKDAY;
     PT1, PT2: CHAIN;
```

9.2 Variable Denotations

An entire variable is a variable declared with a simple type and is denoted by its identifier. A component variable is a variable declared with an array, record, or pointer type. A component of such a variable is denoted by the variable's identifier followed by a selector specifying the component. The form of the selector depends on the structured type of the variable.

<variable> ::= <entire variable> | <component variable>

<component variable> ::= <array component> | <record component> |
<pointer component>

9.2.1 Entire Variable

An entire variable is denoted by its identifier.

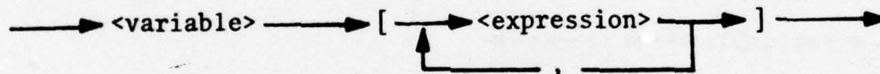
```
<entire variable> ::= <variable identifier>
<variable identifier> ::= <identifier>
```

9.2.2 Array Component

A component of an n-dimensional array variable is selected (denoted) by the variable identifier followed by n index expressions enclosed in square brackets and separated by commas. The index expressions must be compatible with the index types declared in the definition of the array type, and equal in number to the dimensionality of the array variable.

```
<array component> ::= <array variable>[<expression>{,<expression>}]
<array variable> ::= <variable>
```

array component:



Examples:

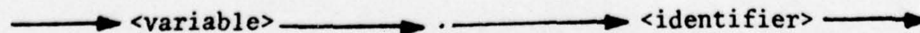
```
VAR M: MATRIX; LATEDAYS: LATE;
M[I+J, K]
LATEDAYS[TUES]
```

9.2.3 Record Component

A component of a record variable is selected (denoted) by the record variable's identifier followed by the field identifier of the component separated by a period.

```
<record component> ::= <record variable>.<field identifier>
<record variable> ::= <variable>
<field identifier> ::= <identifier>
```

record component:



Examples:

```
P1.NAME.LAST
P1.SEX
P1.DATE.DAY
P1.S
P1.RANK
```

9.2.4 Pointer Component

The component of a pointer variable is selected (denoted) by the pointer variable followed by the symbol ↑. Given

```
VAR P:↑T;
```

then P denotes a pointer variable and its pointer value while P↑ denotes the variable of type T referenced by P.

```
<pointer component> ::= <pointer variable>↑
<pointer variable> ::= <variable>
```

pointer component:

```
——► <variable> ——► ↑ ——►
```

Examples:

```
VAR P:↑NODE;
P↑.VALUE
P↑.LINK
P↑.LINK↑.VALUE
```

10. EXPRESSIONS

An expression is a rule of computation for obtaining a value by application of operators to operands. Expressions are in infix notation. The sequence of operations is from left to right with the following priority rules:

first: factors are evaluated
second: terms are evaluated
third: simple expressions are evaluated
fourth: expressions are evaluated

<expression> ::= <simple expression> | <simple expression><relational operator><simple expression>

<simple expression> ::= <term> | <simple expression><adding operator> <term> | <adding operator><term>

<term> ::= <factor> | <term><multiplying operator><factor>

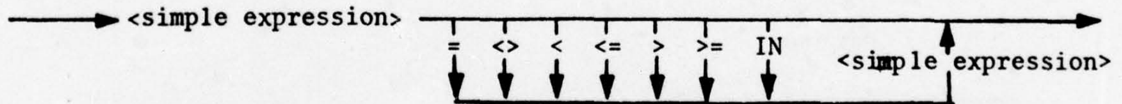
<factor> ::= <variable> | <constant> | <function call> | <set> | (<expression>) | NOT <factor>

<set> ::= [<element list>]

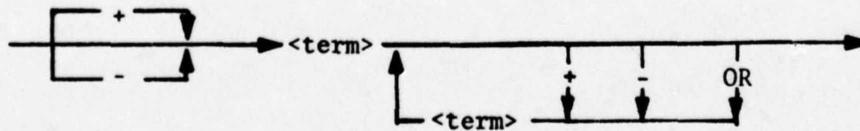
<element list> ::= <element>{,<element>} | <empty>

<element> ::= <expression> | <expression>..<expression>

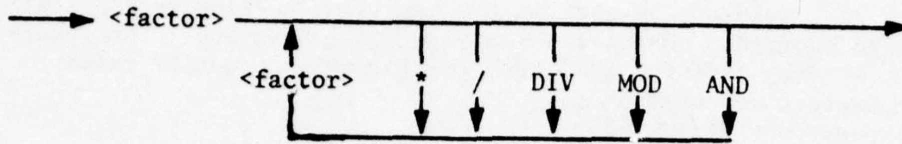
expression:



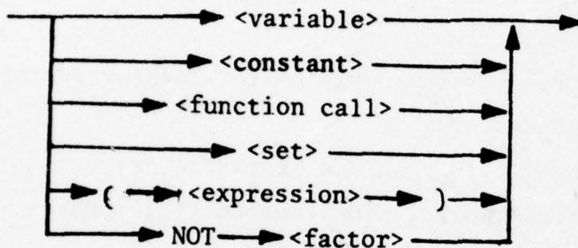
simple expression:



term:



factor:



Examples:

Factors:
```
I
10
F(X+Y, K)
[ ]
[RED]
(X+Y)
NOT BOL
```

Terms:
```
X*Y
I DIV J
X/Y
(X<>Y) AND (Y>Z)
```

Simple expressions:
```
X+Y
--Z
B + SQRT(B*B-4*A*C)
[RED, YELLOW] + HUE1
```

Expressions:
```
X = 5
P>=Q
[RED] IN HUE1
```

10.1 Type Compatibility

An operator can only be applied to two operands if their data types are compatible. Two types are compatible if:

1) both types are defined by the same type definition, or
2) both types are subranges of a single enumeration type, or
3) both are string types of the same length, or
4) both are set types whose members are of compatible base types (the empty set is compatible with any set), or
5) one is of type integer or a subrange thereof and the other is of type real.

Thus there is no conversion of types except integer to real.

10.2 Operators

10.2.1 NOT Operator

NOT denotes the negation of its Boolean operand.

10.2.2 Multiplying Operators

<multiplying operator> ::= * | / | DIV | MOD | AND

Operator	Operation	Type of Operands	Type of Result
*	Multiplication Set intersection	INTEGER, REAL set type T	INTEGER, REAL T
/	Division	REAL, INTEGER	REAL
DIV	division with truncation	INTEGER	INTEGER
MOD	modulus	INTEGER	INTEGER
AND	logical and	BOOLEAN	BOOLEAN

10.2.3 Adding Operators

<adding operator> ::= + | - | OR

Operator	Operation	Type of Operands	Type of Result
binary +	Addition	INTEGER, REAL	INTEGER, REAL
	Set Union	set type T	T
binary -	Subtraction	INTEGER, REAL	INTEGER, REAL
	Set difference	set type T	T
OR	logical or	BOOLEAN	BOOLEAN
unary -	negation	INTEGER, REAL	INTEGER, REAL
unary +	identity	INTEGER, REAL	INTEGER, REAL

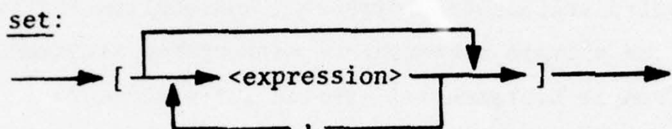
10.2.4 Relational Operators

<relational operator> ::= = | <> | < | <= | >= | > | IN

Operator	Operation	Type of Operands	Type of Result
=	equal	passive	Boolean
	equivalence	Boolean	
<>	unequal	passive	
	exclusive or	Boolean	
>	greater	enumeration, REAL	Boolean
<	less	string	
<=	less or equal	enumeration, string, REAL	Boolean
	contained in	set type T	
	implication	Boolean	
>=	greater or equal	enumeration, string, REAL	Boolean
	contains	set type T	
IN	membership	Left operand is of enumeration type and the right operand is of set type whose membership type is compatible with the left operand.	Boolean

10.3 Sets

Set values are constructed from one or more expressions enclosed in square brackets and separated by commas. The value is the set consisting of the expression values. The set expressions must be of compatible enumeration types.



The empty set is denoted by [ ].

11. STATEMENTS

Statements denote operations on constants and variables. They may be prefixed by a label which can be referenced by a GOTO statement. Simple statements cannot be divided into smaller statements. Structured statements are composed of other statements.

<statement> ::= <unlabelled statement> | <label> : <unlabelled statement>

<unlabelled statement> ::= <simple statement> | <structured statement>

<simple statement> ::= <empty statement> | <assignment statement> |
<procedure call> | <event call> | <goto statement> | <init statement>
<terminate statement>

<structured statement> ::= <compound statement> | <basic structured statement>

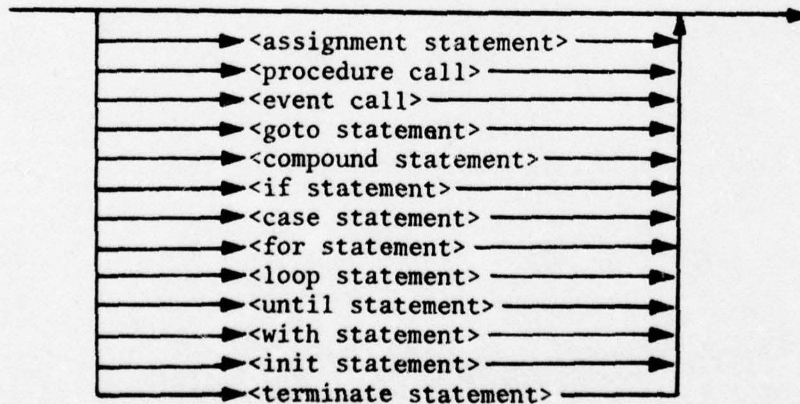
<basic structured statement> ::= <until statement> | <conditional statement>
<repetitive statement> | <with statement>

<conditional statement> ::= <if statement> | <case statement>

<repetitive statement> ::= <for statement> | <loop statement>

<empty statement> ::= <empty>

statement:



11.1 Assignment Statement

The assignment statement serves to replace the current value of a variable by a new value specified by an expression. The type of the variable and the expression must be compatible.

The variable must be of passive type and may not be a constant parameter. Assignment to a function identifier must occur within the block of the function's declaration. There must occur one or more explicit assignment statements of which at least one must be executed.

<assignment statement> ::= <variable> := <expression> | <function identifier>

:= <expression>

assignment statement:

Examples:

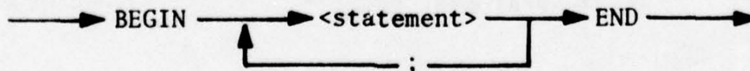
```
I := 1
M[I, J+K] := SQR(J) - I*J
P1.NAME := 'PASCAL'
HUE := [RED, SUCC(YELLOW)]
```

11.2 Compound Statement

A compound statement defines a sequence of statements to be executed sequentially in the same order as they are written. The sequence of statements are separated by the statement separator ; (which does not act as a statement terminator).

<compound statement> ::= BEGIN <statement>{;<statement>} END

compound statement:



Example:

```
BEGIN TEMP := X; X := Y; Y := TEMP END
```

11.3 GOTO Statement

A GOTO statement breaks the normal sequential execution of statements by defining its successor explicitly by a label, i.e., the next statement executed is the one labeled with the specified label. The label of a GOTO statement must be declared in the block containing the GOTO statement, and a statement within such block must be marked by the label. That is, the scope of a label is the block within which it is defined.

<goto statement> ::= GOTO <label>

11.4 IF Statement

The IF statement selects for execution one of two statements depending on the value of a Boolean expression. If the Boolean expression is <u>true</u> then the first statement is executed, else the second is executed. The second statement is optional.

<if statement> ::= IF <expression> THEN <statement component> FI |

IF <expression> THEN <statement component> ELSE <statement component>

FI

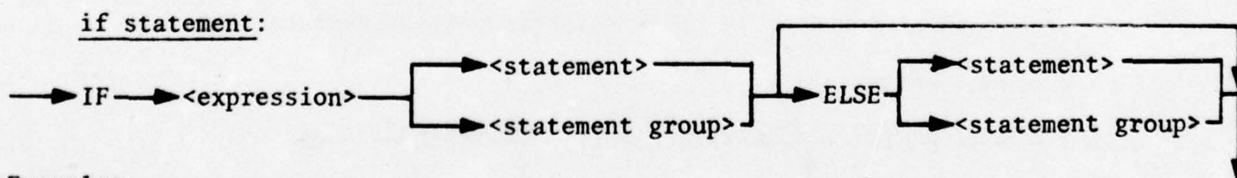
<statement component> ::= <statement> | <statement group>

<statement group> ::= <basic statement>{;<basic statement>}

<basic statement> ::= <simple statement> | <basic structured statement>

The THEN-ELSE, ELSE-FI or THEN-FI act as delemeters around the statement component so that the delimiters BEGIN and END are not needed. A semicolon may never preceed an ELSE or FI. The syntactic ambiguity arising from nested IF statements is resolved by associating an ELSE with the first THEN preceding it.

if statement:



Examples:

IF I>N THEN N := I; M[N] := 0 FI

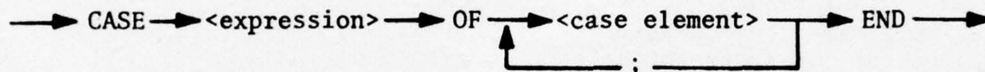
IF X>=Y THEN MAX := X; MIN := Y ELSE MAX := Y; MIN := X FI

11.5 CASE Statement

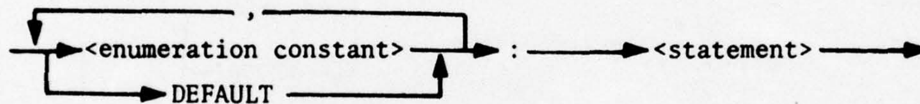
The CASE statement selects one of several statements for execution based on the value of an enumeration expression. Each statement is labeled by one or more unique enumeration constants of the same type as the enumeration expression. The statement labeled with the current value of the expression is executed. If no such label exists, the one statement with the label DEFAULT is executed. If no DEFAULT label exists, none of the statements will be executed.

```
<case statement> ::= CASE <expression> OF <case element>{;case element} END
<case element> ::= <case label list> : <statement>
<case label list> ::= <case label>{,<case label>}
<case label> ::= <enumeration constant> | DEFAULT
```

case statement:



case element:



Example:

```
CASE OPCODE OF
    ADD:    X := X+Y;
    SUB:    X := X-Y;
    MPY:    X := X*Y;
    DIV:    X := X/Y;
    DEFAULT:  ERROR('ILLEGAL OPCODE')
    END;
```

11.6 FOR Statement

A FOR statement specifies that a statement is to be repeatedly executed for a subrange of enumeration values that are assigned to the control variable. The control variable may not be a constant parameter, a record field, a function identifier or an array element.

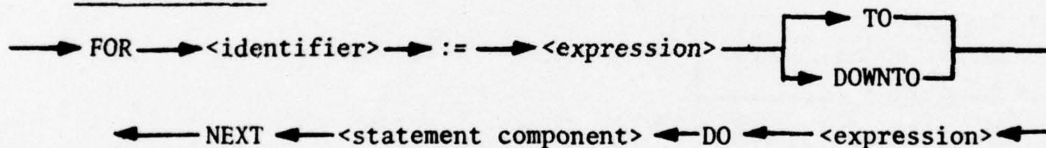
The repeated statement may not change the value of the control variable.

The subrange of enumeration values is specified by expressions defining the initial and final values which are evaluated *only once*. The control variable, the initial value, and the final value must be of compatible enumeration type.

The control variable can either be incremented from its initial value TO its final value or decremented from its initial value DOWNTO its final value. The statement is not executed if the initial value is greater (less) than the final value in the case of TO (DOWNTO). The final value of the control variable is undefined upon normal exit from the for statement.

<for statement> ::= FOR <control variable> := <for list> DO
<statement component> NEXT
<for list> ::= <initial value> TO <final value> | <initial value>
DOWNTO <final value>
<control variable> ::= <identifier>
<initial value> ::= <expression>
<final value> ::= <expression>

for statement



A FOR statement of the form:

```
FOR ID := E1 TO E2 DO SC NEXT
```

is equivalent to the sequence of statements:

```
IV := E1; FV := E2;
IF IV <= FV THEN
    ID := IV; SC;
    ID := SUCC(ID); SC;
    :
    :
    ID := FV; SC
    FI
```

A FOR statement of the form:

```
     FOR ID := E1 DOWNTO E2 DO SC
```

is equivalent to the sequence of statements:

```
  IV := E1; FV := E2;
  IF IV >= FV THEN
     ID := IV; SC;
     ID := PRED(ID); SC;
      :
      .
     ID := FV; SC
     FI
```

A semicolon may never preceed a NEXT.

Examples:

```
     FOR K := 1 TO 10 DO
         X := X + M[I,K]
         NEXT

     FOR WORKWEEK := MON TO FRI DO
         IF LATEDAYS[WORKDAY] THEN
            PAY := PAY - DAYWAGE FI
         NEXT
```

11.7 LOOP-WHILE Statement

The LOOP-WHILE statement specifies that certain statements are to be executed repeatedly while a Boolean expression remains true.

<loop statement> ::= LOOP <statement component> WHILE <expression> :
<statement component> REPEAT

loop statement:

→ LOOP → <statement component> → WHILE → <expression> → : →
← REPEAT ← <statement component> ←

Two important cases occur when either statement component is empty. If the first statement component is empty, the form is:

```
     LOOP WHILE B:
          SC
          REPEAT
```

This is the "while B do S" statement found in other languages. If the Boolean expression is initially false the statement component is not executed; otherwise it is executed repeatedly while the Boolean expression remains true.

If the second statement component is empty, the form is:

```
     LOOP
          SC
          WHILE B: REPEAT
```

This is the "repeat S until¬B" statement found in other languages. The statement component is executed at least once. The statement component is executed until the Boolean expression becomes false.

A semicolon may never preceed a WHILE or REPEAT.

Example:

```
     {Quicksort of array elements A[m] to A[n]}
     I := M; J := N; V := A[N];
     LOOP
          LOOP WHILE A[I]<V : I := I+1 REPEAT;
          LOOP WHILE A[J]>V : J := J-1 REPEAT;
     WHILE I<J :
          A[I] := A[J];
          I := I+1; J := J-1
     REPEAT
```

11.8 UNTIL Statement

The UNTIL statement specifies that a structured statement is to be executed until one of the designated events bound to it is invoked within the structured statement. When the event is invoked (cf. Section 11.10.3) control leaves the structured statement and the next statement is executed, i.e., execution of the structured statement is terminated. If no situation is invoked, execution of the structured statement terminates in the normal manner.

Events provide a mechanism for exiting from nested structured statements, in particular, a multilevel loop.

```
<until statement> ::= UNTIL <event identifier>{,<event identifier>} :
     <structured statement>
<event identifier> ::= <identifier>
```

until statement:

Examples:

```
UNTIL ERROR :
     BEGIN
      ...
     IF AVAIL=NIL THEN ERROR('AVAILABLE SPACE EXHAUSTED') FI;
      ...
     END

UNTIL EXITFOR :
     FOR I := 1 TO N DO
          FOR J := 1 TO M DO
               ...
               IF M[I, J] = Ø THEN EXITFOR FI; {both for loops are terminated}
               ...
          NEXT
     NEXT
```

11.9 WITH Statement

The WITH statement permits record fields and monitor entry routines (outside the monitor type in which they are declared) to be used as variable identifiers within the qualified statement, i.e., it is unnecessary to qualify them with the identifiers of the record or monitor variable. No assignments may be made in the qualified statement to any elements of the with variable list. However, assignments are possible to the components of these variables.

```
<with statement> ::= WITH <with variable list> DO <qualified statement>
<with variable list> ::= <with variable>{,<with variable>}
<with variable> ::= <record variable> | <monitor variable>
<qualified statement> ::= <statement>
```

with statement

```
——> WITH ——> <variable> ——> DO ——> <statement> ——>
         ^                |
         |_______ , ______|
```

The use of more than one <with variable> is equivalent to the use of nested WITH statements, i.e.,

```
     WITH V1, V2, .., Vn DO S
```

is equivalent to

```
     WITH V1 DO
          WITH V2 DO
               ....
               WITH Vn DO
```

Example:

```
     WITH P1, NAME, BIRTH DO
          BEGIN LAST := 'PASCAL';
               SOCIALSECURITY := 200308394;
               MONTH := JAN;
               S := FACULTY;
               POSITION := DEAN
          END
```

11.10 Routine Calls

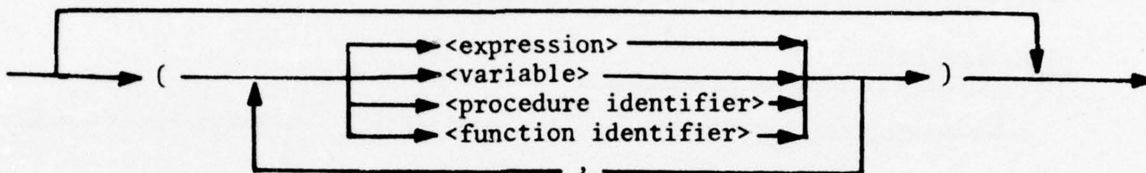
A routine call specifies the execution of a procedure, function or event with a list of actual parameters known as arguments. The arguments are variables, expressions, procedure identifiers and function identifiers, and are substituted for the corresponding (formal) parameters defined in the routine declaration (cf. Section 12). The arguments are substituted for the parameters before the routine is executed. The number of arguments must equal the number of parameters specified in the routine declaration, and the correspondence is established by position. Arguments corresponding to variable parameters must be variables while arguments corresponding to constant parameters must be expressions. Components of a packed structure may not be arguments corresponding to variable parameters. Arguments corresponding to procedure/function parameters must be procedure/function identifiers.

An argument type must be compatible with the corresponding parameter types with the following exceptions:

1) an argument corresponding to a constant non-universal string parameter may be a string of any length,
2) an argument corresponding to a universal parameter may be any passive type (except a pointer type) that occupies the same number of store locations as the parameter type.

<arguments> ::= <empty> | (<argument list>)
<argument list> ::= <argument>{,<argument>}
<argument> ::= <expression> | <variable> | <procedure identifier> |
<function identifier>

arguments:



A monitor type may not call its own entry routines but it may call an entry routine declared within another monitor type. A monitor type may call one of its own simple routines.

An entry routine declared in a monitor type can be called simultaneously by one or more processes or monitor types. The calls will be executed strictly one at a time if the monitor entry routines operate on the same parameters and variables of the monitor.

There are three kinds of routine calls: procedure call, function call and event call.

<routine call> ::= <procedure call> | <function call> | <event call>

11.10.1 Procedure Call

A procedure call is a statement that specifies the execution of a procedure.

A simple procedure is denoted by its identifier. An entry procedure is denoted by qualifying the procedure identifier with the identifier of a variable or the monitor type defining the procedure.

The use of the procedure identifier in a procedure call within its declaration implies recursive execution of the procedure.

```
<procedure call> ::= <simple procedure call> | <entry procedure call>
<simple procedure call> ::= <simple procedure identifier><arguments>
<entry procedure call> ::= <monitor variable>.<entry procedure identifier>
    <arguments>
<simple procedure identifier> ::= <identifier>
<monitor variable> ::= <identifier>
<entry procedure identifier> ::= <identifier>
```

procedure call:

Examples:

```
MATRIXMUL(M1, M2, M3);
INSERT(PT1, I*J);
```

11.10.2 Function Call

A function call is a factor in an expression (cf. Section 10). The remarks on a procedure call apply to a function call as well. Just substitute the word "function" for the word "procedure" throughout the text.

Examples:

```
X := SIMPSON(0, PI/2, G);
I := A(5,2);
```

11.10.3 Event Call

An event call is a statement that specifies an event is to be invoked. Normally a routine returns to the point after the call. However, an event returns to the statement immediately succeeding the structured statement to which it is bound. An event may not be called recursively.

```
<event call> ::= <event identifier><arguments>
<event identifier> ::= <identifier>
```

Example:

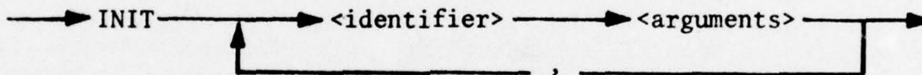
```
ERROR('AVAILABLE SPACE EXHAUSTED');
```

11.11 INIT Statement

The INIT statement initializes a process or variable of type monitor. For a process, an instance is created and its statements executed sequentially. The process is executed concurrently with all other processes, including the one that initialized it. For a monitor, its initial statement is executed as a nameless routine. A monitor can only be initialized once. This must be done within the process in which it is declared. In both cases, the rules for arguments are the same as for arguments in routine calls (cf. Section 11.10).

```
<init statement> ::= INIT <system identifier><arguments>{,<system
    identifier><arguments>}
<system identifier> ::= <process identifier> | <monitor variable>
<process identifier> ::= <identifier>
<monitor variable> ::= <identifier>
```

init statement:



Example:

```
INIT  PRODUCER, PRODUCER, PRODUCER, CONSUMER, CONSUMER, BOUNDED_BUFFER
{initializes three PRODUCER processes, two CONSUMER processes and the
 BOUNDED_BUFFER monitor}
```

11.12 TERMINATE Statement

The TERMINATE statement terminates the execution of the specified process instance(s).

```
<terminate statement> ::= TERMINATE <process reference>{,<process reference>}
<process reference> ::= <variable>
```

A process reference must be a variable of type REF (cf. Section 8.1.4) whose value was set by the standard procedure PROSID (cf. Section 13). After execution of the statement, the value of the process reference is zero.

12. ROUTINE DECLARATIONS

A routine declaration defines a list of (formal) parameters, if any, and a compound statement that operates on them. Execution of the compound statement can be invoked by a routine call (cf. Section 11.10). The parameter list defines the type of parameters on which a routine can operate. Each parameter is specified by its name and type. There are five kinds of parameters: variable, constant, universal, function and procedure.

A variable (call by reference) parameter represents a variable which within the routine may be assigned a value. It is prefixed with the word VAR.

A constant (call by value) parameter represents an expression that is evaluated when the routine is called. Its value cannot be changed by the routine. It is not prefixed with any word.

A universal parameter causes compatibility checking of parameter and argument types in routine calls to be suppressed. (cf. Section 11.10). Its type identifier is prefixed with the word UNIV. Inside the given routine the parameter is considered to be of its specified non-universal type, and outside the routine call the argument is considered to be its declared non-universal type. Universal parameters must be any passive type except a pointer type.

A procedure parameter represents the name of a procedure that may be used as a procedure call. Specification of a procedure parameter also includes the kinds of its parameters.

A function parameter represents the name of a function that may be used as a function call. Specification of a function parameter also includes the kind of its parameters and the type of the function's value. The result type must be an enumeration or pointer type.

The parameters and variable declared within a routine are temporary variables, i.e., they exist only while the routine is being executed.

<parameters> ::= <empty> | (<parameter list>)

<parameter list> ::= <parameter description>{;<parameter description>}

<parameter description> ::= <parameter group> | VAR <parameter group> |
PROCEDURE <identifiers><parameter declarers> |
FUNCTION <identifiers><parameter declarers> : <result type>

<parameter group> ::= <identifiers> : <type identifier> | <identifiers> :
UNIV <type identifier>

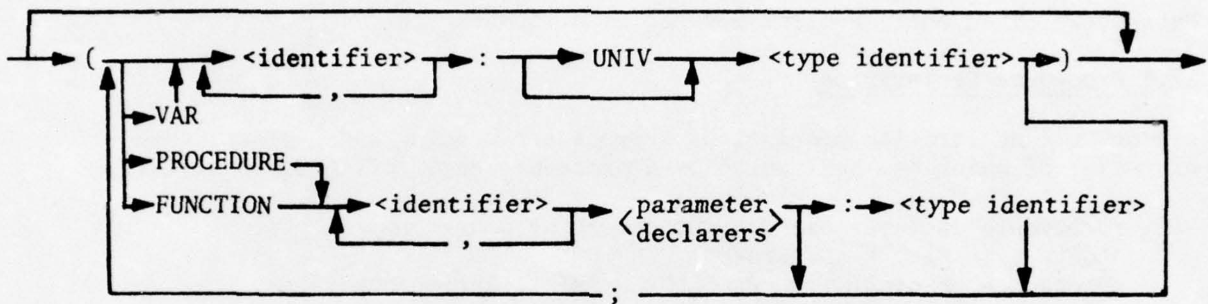
<identifiers> ::= <identifier>{,<identifier>}

<parameter declarers> ::= <empty> | (<parameter declarer list>)

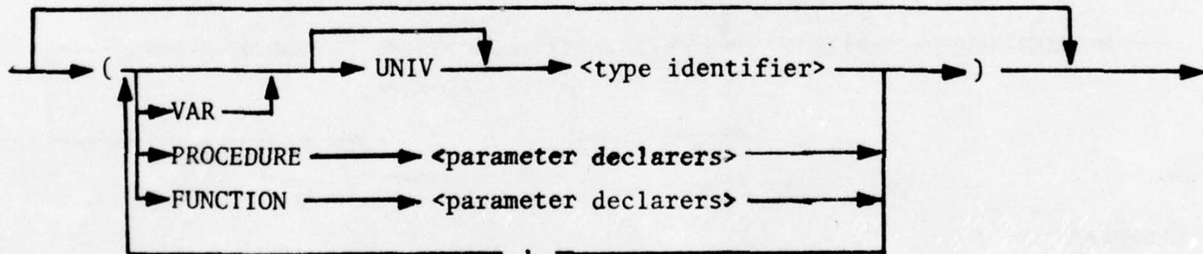
<parameter declarer list> ::= <parameter declarer>{,<parameter declarer>}

<parameter declarer> ::= <type identifier> | VAR <type identifier> |
UNIV <type identifier> | VAR UNIV <type identifier> |
PROCEDURE <parameter declarers> | FUNCTION <parameter declarers>

parameters:



parameter declarers:



There are three kinds of routine declarations: procedure declaration, function declaration and event declaration.

<routine declaration> ::= <procedure declaration> | <function declaration> |
<event declaration>

If a routine is referenced before it has been declared, then its heading followed by the symbol FORWARD must be introduced first. The routine can be completed later by repeating its heading, without the parameter list, followed by the block.

A procedure/function declaration may be an entry routine in which case it is prefixed with the symbol ENTRY, otherwise the declaration is known as a simple procedure/function. An entry routine declaration may only occur within a monitor type and cannot be nested within another routine declaration.

A monitor entry routine is an entry routine declared within a monitor type. It can be called simultaneously by one or more system types (cf. Section 11.10). A monitor entry routine has exclusive access to permanent monitor variables while it is being executed.

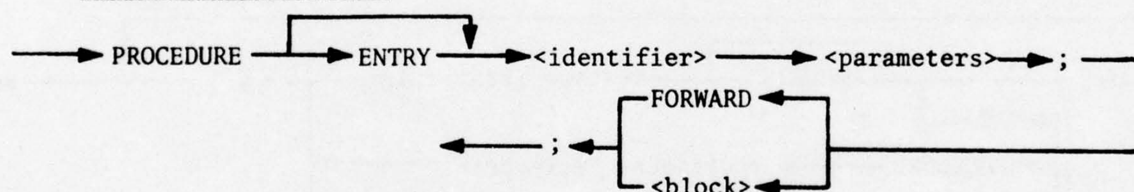
Parameters of an entry routine may not be of queue type.

12.1 Procedure Declaration

A procedure declaration consists of a procedure heading and a block, the execution of which can be invoked by a procedure call (cf. Section 11.10.1).

```
<procedure declaration> ::= <procedure heading><body>
<body> ::= <block> | FORWARD
<procedure heading> ::=  PROCEDURE <identifier><parameters>;
     PROCEDURE ENTRY <identifier><parameters>;
```

procedure declaration:



Examples

```
PROCEDURE MATRIXMUL(A, B : MATRIX; VAR C : MATRIX);
      VAR I, J, K : 0..10; SUM : REAL;
      BEGIN
      FOR I := 0 TO 10 DO
            FOR J := 0 TO 10 DO
                 SUM := 0.0;
                      FOR K := 0 TO 10 DO
                           SUM := SUM + A[I, K] * B[K, J]
                           NEXT {K};
                      C[I, J] := SUM
                 NEXT {J};
            NEXT {I};
      END {MATRIXMUL};
```

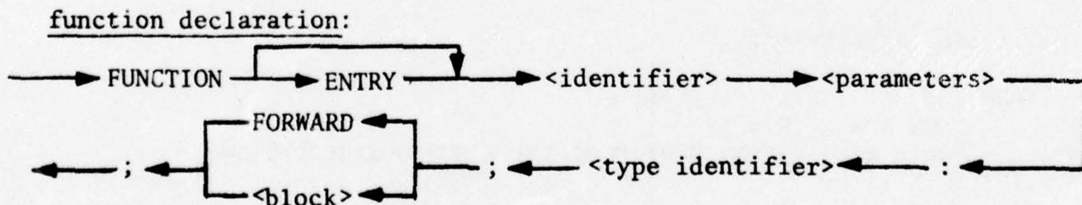
```
PROCEDURE INSERT (P : CHAIN; V : INTEGER);
     {insert a new node after node P in a linear linked list and initialize
      its VALUE field to V}
     VAR Q : CHAIN;
     BEGIN NEW(Q);
          WITH Q↑ DO
               BEGIN VALUE := V;
                    LINK := P↑.LINK
               END;
          P↑.LINK := Q
     END {insert};
```

12.2 Function Declaration

A function declaration consists of a function heading and a block, the execution of which can be invoked by a function call (cf. Section 11.10.2). The function heading specifies a result type which must be an enumeration or pointer type. The result of a function is defined by assigning a value to the function identifier within the function declaration. Function parameters must be constant parameters.

<function declaration> ::= <function heading><body>
<body> ::= <block> | FORWARD
<function heading> ::= FUNCTION <identifier><parameters> : <result type>; |
    FUNCTION ENTRY <identifier><parameter> : <result type>;
<result type> ::= <type identifier>

function declaration:



Examples:

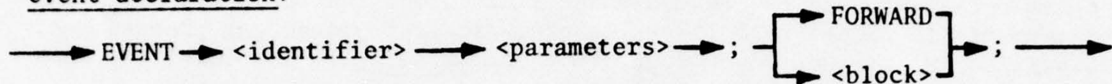
```
{cf. Wirth [35]}
FUNCTION  SIMPSON(A, B : REAL; FUNCTION F(REAL) : REAL) : REAL;
     CONS EPSILON = 0.00001;
     VAR I, N : INTEGER;
          S, SS, S1, S2, S4, H : REAL;
     {F(X) must be well-defined in the interval A<X < B}
BEGIN N := 2; H := (B-A)*0.5;
     S1 := H*(F(A) + F(B)); S2 := 0;
     S4 := 4*H*F(A + H); S := S1 + S2 + S4;
     LOOP SS := S; N := 2*N; H := H/2;
          S1 := 0.5*S1; S2 := 0.5*S2 + 0.25*S4;
          S4 := 0; I := 1;
          LOOP S4 := S4 + F(A + I * H); I := I + 2
               WHILE I<=N:
          REPEAT;
          S4 := 4*H*S4; S := S1 + S2 + S4
          WHILE ABS(S-SS)>= EPSILON :
     REPEAT;
     SIMPSON := S/3
 END {SIMPSON};

FUNCTION A(M, N : INTEGER) : INTEGER;
    {Ackermann's function}
    BEGIN
    IF  M = 0 THEN A := N + 1
          ELSE IF N = 0 THEN A := A(M-1, 1)
                    ELSE A := A(M-1, A(M, N-1)) FI;
          FI
    END {Ackermann};

FUNCTION G(X : REAL) : REAL;
     CONS A = 3; B = 5;
     BEGIN G := 1/SQRT(SQR(A*COS(X)) + SQR(B*SIN(X))) END;
```

12.3 Event Declaration

An event declaration consists of an event heading and a block, the execution of which can be invoked by an event call (cf. Section 11.10.3). The compound statement of the block represents the action to be taken.

<event declaration> ::= <event heading><action>
<action> ::= <block> | FORWARD
<event heading> ::= EVENT <identifier><parameters>;

event declaration:



Example

```
EVENT ERROR (S : STRING);
     BEGIN PRINT('UNRECOVERABLE ERROR:',S) END;

EVENT EXITFOR;
     BEGIN END;
     {Since the action is empty, invoking the event is equivalent to
      exiting from the structured statement the event identifier is
      bound to}.
```

13. PROCESS DECLARATION

A process declaration is the prototype for a class of sequential processes. It consists of a process heading and a block. A sequential process is an instance of a process declaration whose statements are executed sequentially. A sequential process is created and its execution initiated by the INIT statement (cf. Section 11.11). The executions of sequential process are overlaped in time and under control of the kernel (cf. Section 14). Such processes are said to be concurrent.

A process declaration may only be nested within another process declaration. The entire program is an implied process (cf. Section 12). The parameters must be constant parameters either of passive type or a monitor component.

Concurrent processes communicate only by means of monitors (cf. Section 8.2.5). One process cannot operate on the parameters or local variables of another process.

```
<process declaration> ::= <process heading><block>
<process heading> ::= PROCESS <identifier><parameters>;
```

process declaration

──► PROCESS ──► <identifier> ──► <parameters> ──► ; ──►

Examples: (cf. Hoare [26])

```
    ...
TYPE PORTION = RECORD ... END;
    ...
PROCESS PRODUCER;
    VAR  NOTFINISHED : BOOLEAN;
        INFO : PORTION;
    BEGIN  NOTFINISHED := TRUE;
        LOOP WHILE  NOTFINISHED :
            {produce the next portion}
            BOUNDED_BUFFER.APPEND(INFO); {add portion to buffer}
            REPEAT;
    END;
```

```
PROCESS CONSUMER;
     VAR  NOTFINISHED : BOOLEAN;
          INFO : PORTION;
     BEGIN  NOTFINISHED := TRUE;
          LOOP WHILE NOTFINISHED:
               BOUNDED_BUFFER.REMOVE(INFO);  {take portion from buffer}
               {process portion taken}
               REPEAT;
     END;
```

(cf. Section 8.2.5 for examples defining the monitor BOUNDED_BUFFER which contains the routines APPEND and REMOVE).

14. CONCURRENT PROGRAM

A concurrent program consists of a library prelude followed by a block. The block is an anonymous paramterless process called the *initial process*. An instance of this process is automatically initialized after program loading. *Program loading* consists of loading the compiled code for a concurrent program along with the library routines mentioned on its *library prelude*. The library prelude consists of constant type and routine definitions. The library prelude routines consist only of procedure and/or function headings. The library routines are defined within the *library*. The library is a separately compiled program that consists only of constant, type and routine declarations. The order of the routine definitions must be the same as the corresponding declarations in the library.

<program> ::= <library prelude><program heading><block>.

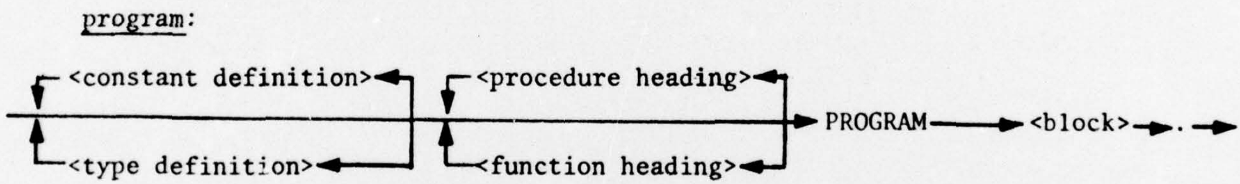
<library prelude> ::= {<constant or type definition >}{<library routine definition >}

<constant or type definition> ::= <constant definition> | <type definition>

<library routine definition> ::= <procedure heading> | <function heading>

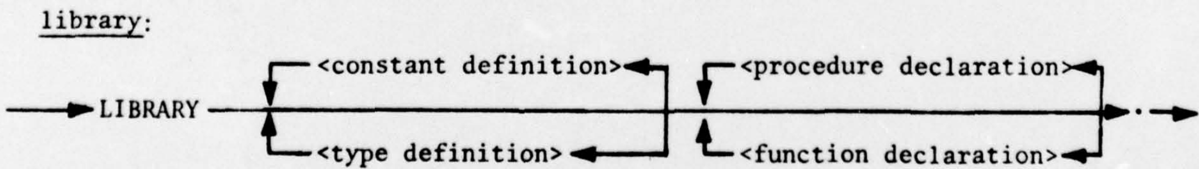
<program heading> ::= PROGRAM

program:



<library> ::= <library heading><library declarations>.
<library heading> ::= LIBRARY
<library declarations> ::= {<constant or type definition>}{library routine declaration>}
<library routine declaration> ::= <procedure declaration> | <function declaration>

library:



A program and the library are the basic units of compilation.

Execution of a compiled concurrent program does not require an operating system for support. The minimal run-time support needed is contained in the kernel, a program written in the machine language of the object machine. The kernel permits a user to construct the requisite operating system facilities and policies needed to support his stand alone system. In essence, the kernel creates a virtual machine that a concurrent program interfaces to.

The kernel performs the following functions:

1) Process management
   The kernel controls the creation, execution and termination of processes. When a process is created, the kernel allocates a processor to it. If there are more processes than processors, the kernel multiplexes processors among them. Selection of a process to be executed is based on priority with top priority given to processes executing monitor code. When the process terminates, its processor is deallocated.

2) Memory management
   When a process is created, storage for its stack and heap are allocated. During execution, storage for a procedure is automatically allocated and deallocated from its stack upon procedure entry and exit. The execution stack is located after a procedure's activation record. Allocation of storage from the heap is controlled by standard library routines. When a process terminates, its storage is deallocated. How and when deallocation is managed is implementation dependent.

3) Gives exclusive access to monitors
   The kernel handles the short-term scheduling of simultaneous calls on monitor entry routines and guarantees that the calls will be executed one at a time. A user can, via a monitor, choose his own strategy of medium-term process scheduling.

4) I/O management
   The kernel makes peripheral devices appear uniform with respect to simple I/O and exception conditions. Simple I/O is performed via a standard routine which starts a data transfer. The process calling the I/O routine is suspended until resumed by the I/O interrupt (cf. Section 16). A user must guarantee that only one process at a time uses a peripheral and perform error recovery.

5) Handles interrupts

When an interrupt occurs, the kernel resumes the process which is associated with the interrupt.

6) Provides real-time control

The kernel maintains the time of day clock and the real-time clock. Standard routines are used to obtain the time of day, and delay a process for a period of time.

15. SCOPE RULES

The scope of an identifier is that region of the program text where it is known with a single meaning. A scope is either a program, routine, monitor type, record type or WITH statement.

In order to be known, all identifiers must be introduced either by a declaration or a qualification (the singular exception to this rule is a pointer type which may refer to a type not yet defined). A declaration associates an identifier with a particular variable, constant, type or routine. Qualification associates a field or entry identifier with a particular record variable or monitor variable respectively. A qualification is either the variable name followed by a period or a WITH statement (cf. Sections 9.2.3, 11.8, 11.10.1).

Two scopes are either disjoint or one is embedded inside the other. When a scope is embedded within another scope, the inner scope is nested in the outer scope.

An identifier can only be introduced with one meaning in a scope. However, it may be introduced with another meaning in another disjoint or inner scope.

Within a program are known:

a) any standard identifier,
b) constant, type, and routine identifiers introduced within and after the library prelude (cf. Section 14),
c) labels declared after the library prelude.

Within a monitor type are known:

a) any standard identifier,
b) all identifier and labels introduced within the monitor type itself except its entry routine identifiers,
c) all constant and type identifiers introduced within a program different from those in (b).

Within a routine are known:

a) any standard identifier,
b) all identifiers introduced within the routine, including the routine identifier,
c) all identifiers introduced in its outer scopes different from those in (b),
d) all labels declared in the routine.

Within a record are known:

a) all standard type identifiers,
b) all type identifiers introduced in its outer scopes.

Within the WITH statement are known:

a) any standard identifier,

b) all identifiers introduced by the WITH statement itself and by its outer WITH statements,

c) all identifiers introduced in its outer (non-WITH statement) scopes different from those in (b).

16. INPUT/OUTPUT

Basic I/O is handled by the standard procedure IO, a primitive operation upon which a user may construct various device handlers, interrupt handlers, I/O exception handling routines, generalized I/O control routines, and file systems. The parameters of IO have data types that completely characterize the peripheral devices available to a user on a particular machine. These types are available to a user on a particular machine. These types are recognized by the kernel (cf. Section 14) which treats the devices in a uniform manner. The kernel suspends the process calling IO, initiates the I/O operation, processes the I/O interrupt, returns I/O status information (through one of the parameters to IO) and resumes the delayed process. The kernel assumes that a device is used by only one process at a time. It is the user's responsibility to guarantee this assumption is satisfied. Also, the user must perform error recovery. Since the process calling IO is suspended until the I/O operation is completed, I/O requests should be processed by a process different from the one making the request if the I/O transfer is to proceed in parallel with the execution of the requesting process.

The form of the standard procedure IO is:

```
IO(IOVAR, IOPAR, IODEVICE)
```

Calling IO is a request that peripheral device IODEVICE perform the I/O operation specified in IOPAR on variable IOVAR.

IOVAR and IOPAR are variable parameters of arbitrary passive types. The type of IOVAR depends on the device, e.g., for a terminal device, the unit of data transmitted may be a single character so the type of the argument corresponding to IOVAR must be CHAR, or for devices like a disk, printer, card reader, card punch and magnetic tape the unit of data transmitted is a string of characters representing respectively a disk record, print line, card image, punched card and tape block so the type of the argument corresponding to IOVAR must be STRING with an appropriate length.

IOPAR is of type RECORD. This record contains fields representing the I/O operation, I/O result and an I/O argument whose values vary from device to device. The I/O operation field is an enumeration type whose values are the possible I/O operations, i.e., whether to send or receive data and/or control information. The I/O result field is of enumeration type whose values are the possible outcomes of the I/O operation, e.g., operation completed successfully, operation failed due to a transmission error, end-of-file mark sensed, etc.. The I/O argument field provides additional information pertinent to the device, e.g., an integer representing a disk record, or an enumeration constant specifying further the kind of I/O move operation for a tape unit-i.e., whether to skip forward or backward a record, rewind the tape, etc..

IODEVICE is a constant parameter of arbitrary enumeration type whose values represent the various available I/O devices.

The IO procedure translates characters on input from their internal representation on the object machine to the internal representation (ordinal value) defined by the compiler, and vice versa for output.

Example:

Assume the kernel recognizes the following types:

```
{type of IODEVICE}
TYPE PERIPHERALS = (READER, PRINTER, DISK, TAPE, TERMINAL);
     ...
TYPE IOOPERATION = (INPUT, OUTPUT, MOVE, CONTROL);
     IORESULT = (COMPLETE, TRANSMISSION_ERROR, ENDFILE);
     ...
{type of IOPAR}
TYPE IOPARM =
     RECORD
          OPERATION : IOOPERATION;
          STATUS : IORESULT;
          CASE ARG : PERIPHERALS OF
              DISK : (PAGE_INDEX : INTEGER);
              TAPE : (MOVE_OPERATION : (SKIP_FORWARD, BACKSPACE, REWIND,
                     OUTEOF));
          PRINTER : (LAYOUT_OPERATION : (SINGLE_SPACE, DOUBLE_SPACE,
                     NEW_PAGE))
        END;
```

Then PASCAL's READ(INPUT, V1) procedure (Jensen & Wirth [15]), where the INPUT file is the card reader, V1 is a variable of type integer, real or character, would be programmed as follows:

```
CONS EOL = (:177:); NO_FILES = ...;
TYPE TEXT = PACKED ARRAY [1..81] OF CHAR;
    DATATYPE = (INT, REEL, CHR);
    INVAR = RECORD
                 CASE VTYPE : DATATYPE OF
                      INT : (VI : INTEGER);
                      REEL : (VR : REAL);
                      CHR : (VC : CHAR)
            END;
    FILE = 1..NO_FILES;
    BUFFER_TYPE = RECORD
                       PT : 0..80;
                       DATA : TEXT;
                       EOLINE, EOFILE : BOOLEAN {initially TRUE, FALSE}
                  END;
```

```
VAR F : FILE;
     INBUF : BUFFER_TYPE;

PROCEDURE GET(VAR BUF : BUFFER_TYPE);
{advance the current buffer position to the next character}.
VAR READPAR : IOPARM;
BEGIN WITH BUF DO
     BEGIN
     IF EOFILE THEN GOTO 1 FI;
     IF EOLINE THEN
          READPAR.OPERATION := INPUT;
          IO(BUF, READPAR, READER);
          IF READPAR.STATUS = ENDFILE THEN
               EOFILE := TRUE;
               GOTO 1
               FI;
          PT := 0; EOLINE := FALSE; DATA[81] := EOL
          FI;
     PT := PT + 1;
     IF DATA[PT] = EOL THEN
          DATA[PT] := ' ';
          EOLINE := TRUE
          FI;
     END
1 : END {GET};

PROCEDURE READI(VAR BUFFER : BUFFER_TYPE, VAR V1 : INVAR);
{scan integer in BUFFER and place in V1}
VAR I : INTEGER;
     SIGN : 0..1;
BEGIN WITH BUFFER DO
     BEGIN {note: DATA[PT] is the next character to be read}
     LOOP WHILE DATA[PT] = ' ' : GET(BUFFER) REPEAT;
     SIGN := 0;
     IF DATA[PT] = '+' THEN GET(BUFFER)
          ELSE IF DATA[PT] = '-' THEN
               SIGN = 1; GET(BUFFER) FI
          FI;
     LOOP WHILE DATA[PT] >= '0' AND DATA[PT] <= '9':
          I := 10*I+ (DATA[PT] - '0');
          GET(BUFFER)
          REPEAT;
     IF SIGN = 1 THEN I := -I FI;
     V1.VI := I
     END
END {READI};
```

```
PROCEDURE READ(F : FILE, VAR V1 : INVAR);
{read from file F the value for V1}
BEGIN
CASE F OF
     1 {INPUT} :  CASE V1.VTYPE OF
                       INT : READI(INBUF, V1);
                       REEL : READR(INBUF, V1);
                       CHR : READC(INBUF, V1)
                  END;
     2 {OUTPUT} : ERROR ('READ ON OUTPUT FILE');
           :
           :
           END
     END;
```

Similarly for READR and READC.

17. REAL-TIME CONTROL

For real-time applications a user may want to delay a process some specified amount of time or measure the real time taken by a process to perform some action. These real-time controls are provided by the following standard routines:

WAIT(T) The calling process is delayed T units of time where a unit of time is implementation dependent. If the call occurs in a monitor, other calls on this monitor will be delayed.

TIME The result is an integer defining the real time in seconds since system initialization.

It is assumed the object machine has one or more real-time clocks in order to implement the above routines.

GLOSSARY

active type: a type containing monitor types, queue types and reference types.

argument: a variable, expression, procedure identifier or function identifier passed in an argument list, i.e., the actual parameter to a routine.

array type: defines a composite structure with indexable components of homogeneous type.

concurrent process: a sequential process whose execution is overlapped in in time with other sequential processes.

constant parameter: a parameter defined without the prefix VAR. Its value cannot be changed.

data type: the definition of a data structure and the operations that may be performed on it.

entry routine: a procedure or function prefixed with the ENTRY keyword. Its scope is that of the monitor type in which it is declared.

enumeration type: a symbolic scalar including Boolean, integer or character type, or subrange thereof.

event: a parameter mechanism which when invoked causes an action to be performed and an exit from an arbitrary nest of control to be taken.

initial process: the block of a concurrent program.

initial statement: the statement of a monitor type that will be executed when a variable of the monitor type is initialized.

kernel: the minimum run-time support provided a user. It controls the allocation of concurrent processes to processors, the exclusive access of concurrent processes to shared data, the peripheral devices, the interrupts and the allocation of storage for processes.

monitor type: defines a shared data structure and the operations through which the data structure may be exclusively accessed by concurrent processes.

parameter: an identifier declared in a parameter list.

passive type: a type not containing a monitor type, queue type or reference type.

permanent variable: the variables declared within a monitor type. They exist forever after initialization of a variable of type monitor.

pointer type: defines a set of values referencing components of a given type.

queue type: defines a queue that may be used by a monitor entry routine to schedule processes.

record type: defines a composite data structure with labeled components of heterogeneous type.

reference type: defines a set of values for referencing an instance of a process.

routine: a procedure, function or event.

scalar type: defines an ordered set of values by enumeration of the identifiers which denote these values.

set type: defines the set of all subsets of values of an enumeration type, including the empty set.

sequential process: an instance of a process declaration whose statements are executed sequentially.

simple routine: a routine that is not an entry routine.

simple type: an enumeration, real, queue or reference type.

string type: a one-dimensional array of characters.

structured type: an array, record, set, pointer or monitor type.

subrange type: an enumeration type that is defined as a subrange of another enumeration type by specifying its minimum and maximum values.

temporary variable: parameters and variables declared within a routine that exist only while the routine is being executed.

type compatibility: two types are compatible if:
1) they are defined by the same type definition, or
2) they are subranges of a single enumeration type, or
3) they are string types of the same length, or
4) they are set types whose members are of compatible base types, or
5) one is of type integer or a subrange thereof and the other is of type real.

universal type: a parameter type defined with the UNIV keyword. Type compatibility between a universal parameter type and the corresponding argument type is supressed in a routine call.

variable parameter: a parameter defined with the prefix VAR. It represents a variable whose value may be changed within the routine.

DISTRIBUTION LIST

Defense Documentation Center 12
Cameron Station
Alexandria, VA 22314

Naval Training Equipment Center 24
Orlando, FL 32813

Naval Air Systems Command 2
NAIR 340
Washington, D.C. 20360

Naval Air Systems Command 2
Library, NAIR-50174
Washington, D.C. 20360

Naval Air Systems Command 1
NAIR-413
Washington, D.C. 20360