PER BRINCH HANSEN

Information Science
California Institute of Technology

June 1975

CONCURRENT PASCAL REPORT

CONCURRENT PASCAL REPORT

Per Brinch Hansen

Information Science
California Institute of Technology

June 1975

Abstract

This report defines Concurrent Pascal - an abstract programming language for structured programming of computer operating systems. It extends the sequential programming language Pascal with concurrent processes, monitors, and classes. Concurrent Pascal has been implemented for the PDP 11/45 computer at Caltech.

Key Words and Phrases: Concurrent Pascal, structured multiprogramming, programming languages, hierarchical operating systems, concurrent processes, monitors, classes, abstract data types, access rights, scheduling, job control.
CR Categories: 4.2, 4.3

The development of Concurrent Pascal has been supported by the National Science Foundation under grant number DCR74-17331.

CONTENTS

1. INTRODUCTION

This report defines Concurrent Pascal - an abstract programming language for structured programming of computer operating systems. It extends the sequential programming language Pascal with concurrent processes, monitors, and classes.

This is a brief, concise definition of Concurrent Pascal. A more informal introduction to sequential and Concurrent Pascal by means of examples is provided by the following reports:

Jensen, K., and Wirth, N. Pascal - user manual and report. Lecture Notes in Computer Science 18, Springer-Verlag, 1974.

Brinch Hansen, P. The programming language Concurrent Pascal. IEEE Transactions on Software Engineering 1, 2 (June 1975).

There are minor differences between the sequential programming concepts defined in the sequential and Concurrent Pascal reports.

The central part of this report is a chapter on data types. It is based on the assumption that data and operations on them are inseparable aspects of computing that should not be dealt with separately. For each data type we define the constants that represent its values and the operators and statements that apply to these values.

Concurrent Pascal has been implemented for the PDP 11/45 computer at Caltech. An appendix defines the additional restrictions and extensions of this implementation.

2. SYNTAX GRAPHS

The language syntax is defined by means of syntax graphs of the form:

while statement

$$\begin{tikzcd} {} \arrow[r] & \text{WHILE} \arrow[r] & \text{expr} \arrow[r] & \text{DO} \arrow[r] & \text{statement} \arrow[r] & {} \end{tikzcd}$$

A syntax graph defines the name and syntax of a language construct. Basic symbols are represented by capitals and special characters, for example

WHILE   DO   +   ;

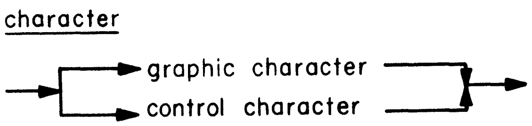
Constructs defined by other graphs are represented by their names written in small letters, for example

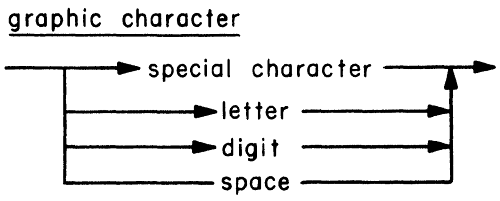
expr   statement

Correct sequences of basic symbols and constructs are represented by arrows.

3. CHARACTER SET

Concurrent programs are written in a subset of the ASCII character set:





A graphic character is a printable character.

The special characters are

!	"	#	$	%	&	'	(	)	*	+
,	-	.	/	:	;	<	=	>	?	@

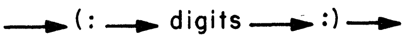
The letters are

A	B	C	D	E	F	G	H	I	J	K
L	M	N	O	P	Q	R	S	T	U	V
W	X	Y	Z	_						

The digits are

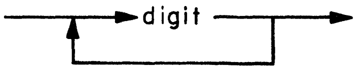
0 1 2 3 4 5 6 7 8 9

control character



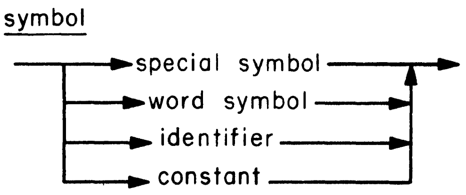
A control character is an unprintable character. It is represented by an integer constant called its ordinal value (Appendix B). The ordinal value must be in the range 0..127.

digits



4. BASIC SYMBOLS

A program consists of symbols and separators.



The special symbols are

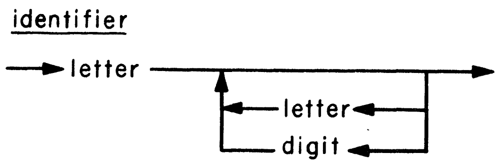
+ - * / & = <> < > <= >=
( ) (. .) := . , ; : ' ..

They have fixed meanings (except within string constants and comments).

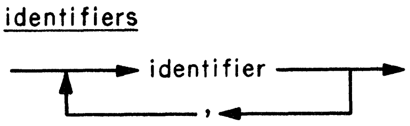
The word symbols are

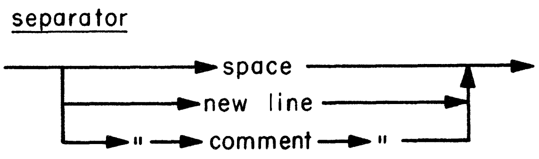
ARRAY	BEGIN	CASE	CLASS	CONST
CYCLE	DIV	DO	DOWNTO	ELSE
END	ENTRY	FOR	FUNCTION	IF
IN	INIT	MOD	MONITOR	NOT
OF	OR	PROCEDURE	PROCESS	PROGRAM
RECORD	REPEAT	SET	THEN	TO
TYPE	UNIV	UNTIL	VAR	WHILE
WITH				

They have fixed meanings (except within string constants and comments). Word symbols cannot be used as identifiers.



An identifier is introduced by a programmer as the name of a constant, type, variable, or routine.





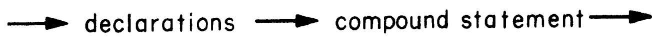
Two constants, identifiers, or word symbols must be separated by at least one separator or special symbol. There may be an arbitrary number of separators between two symbols, but separators may not occur within symbols.

A comment is any sequence of graphic characters (except ") enclosed in quotes. It has no effect on the execution of a program.

5. BLOCKS

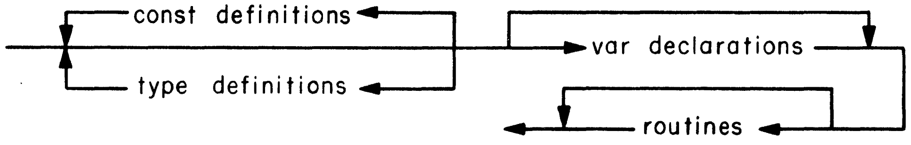
The basic program unit is a block:

block



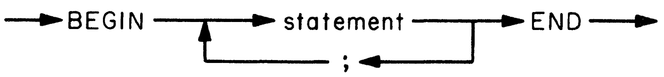
It consists of declarations of computational objects and a compound statement that operates on them.

declarations



A declaration defines a constant, type, variable, or routine and introduces an identifier as its name.

compound statement

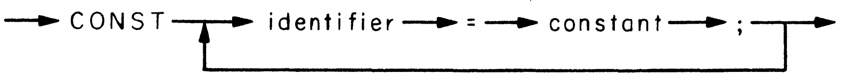


A compound statement defines a sequence of statements to be executed one at a time from left to right.

6. CONSTANTS

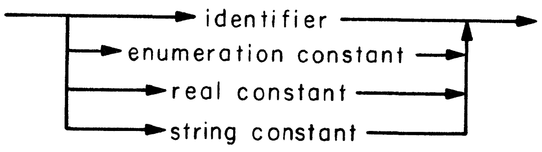
A constant represents a value that can be used as an operand in an expression.

constant definitions



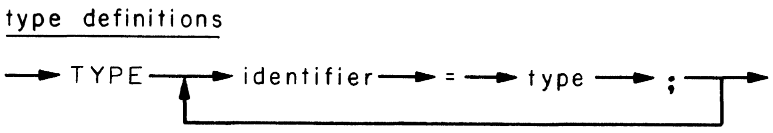
A constant definition introduces an identifier as the name of a constant.

constant

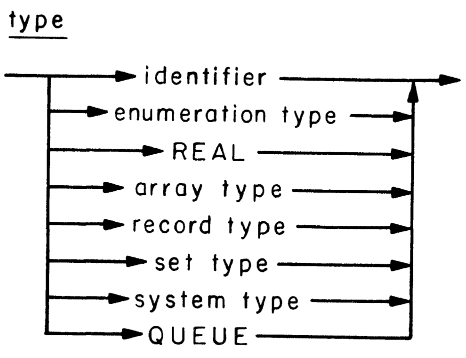


7. TYPES

A data type defines a set of values which a variable or expression may assume.



A type definition introduces an identifier as the name of a data type. A data type cannot refer to its own type identifier.



Enumeration types, reals, and queues can only be operated upon as a whole. They are simple types.

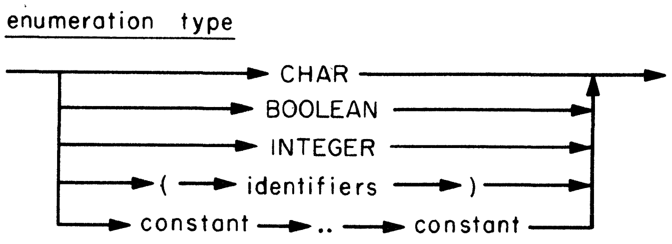
Arrays, records, sets, and system types are defined in terms of other types. They are structured types containing component types.

A data type that neither contains system types nor queues is a passive type. All other types are active types.

An operation can only be performed on two operands if their data types are compatible (Section 9).

7.1. Enumeration types

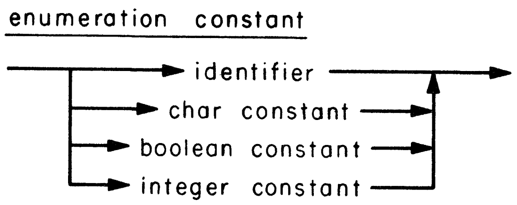
An enumeration type consists of a finite, ordered set of values.



The types char, boolean, and integer are standard enumeration types.

A non-standard enumeration type is defined by listing the identifiers that denote its values in increasing order.

An enumeration type can also be defined as a subrange of another enumeration type by specifying its min and max values (separated by a double period). The min value must not exceed the max value, and they must be compatible enumeration constants (Section 9).

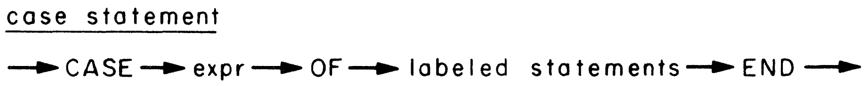


The basic operators for enumerations are:

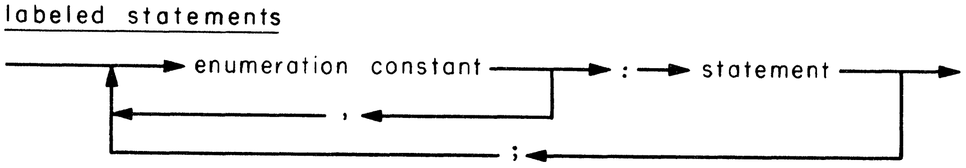
:=	(assignment)
<	(less)
=	(equal)
>	(greater)
<=	(less or equal)
<>	(not equal)
>=	(greater or equal)

The result of a relation is a boolean value.

An enumeration value can be used to select one of several statements for execution:



A case statement defines an enumeration expression and a set of statements. Each statement is labeled by one or more constants of the same type as the expression. A case statement executes the statement which is labeled with the current value of the expression. (If no such label exists, the effect is unknown.)



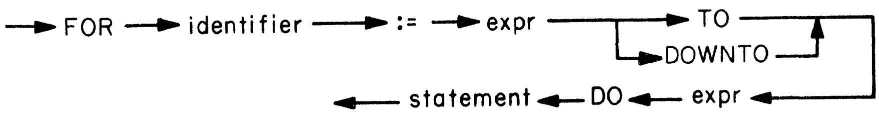
The case expression and the labels must be of compatible enumeration types, and the labels must be unique.

The following standard functions apply to enumerations:

succ(x)	The result is the successor value of x (if it exists).
pred(x)	The result is the predecessor value of x (if it exists).

An enumeration type can be used to execute a statement repeatedly for all the enumeration values:

for statement

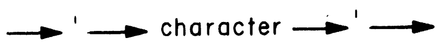


A for statement consists of an identifier of a control variable, two expressions defining a subrange, and a statement to be executed repeatedly for successive values in the subrange. The control variable can either be incremented from its min value TO its max value or decremented from its max value DOWNTO its min value. If the min value is greater than the max value, the statement is not executed. The value of the control variable is undefined after completion of the for statement.
The control variable and the expressions must be of compatible enumeration types. The control variable may not be a constant parameter, a record field, a function identifier, or a variable entry referenced by selection (Sections 7.4, 8.2, 11). The repeated statement may not change the value of the control variable.

7.1.1. Characters

The type CHAR is a standard enumeration type. Its values are the set of ASCII characters represented by char constants:

char constant



The following standard function applies to characters:

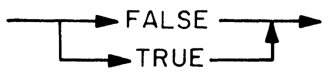
ord(x)	The result (of type integer) is the ordinal value of the character x.

The ordering of characters is defined by their ordinal values (Appendix B).

7.1.2. Booleans

The type BOOLEAN is a standard enumeration type. Its values are represented by boolean constants:

boolean constant



where FALSE < TRUE.

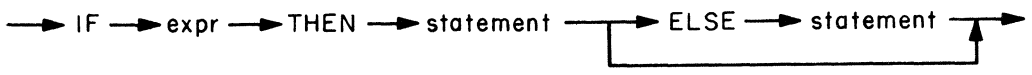
The following operators are defined for booleans:

&	(and)
or	
not	

The result is a boolean value.

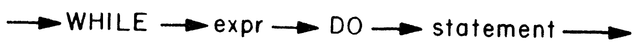
A boolean value can be used to select one of two statements for execution. It can also be used to repeat the execution of a statement while a condition is true (or until it becomes true).

<u>if statement</u>

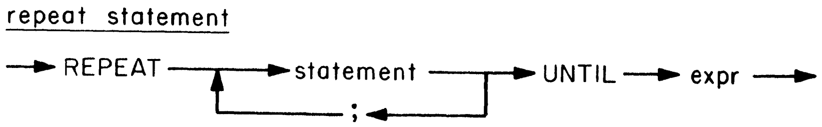


An if statement defines a boolean expression and two statements. If the expression is true then the first statement is executed, else the second statement is executed. The second statement may be omitted in which case it has no effect.
The expression value must be a boolean.

<u>while statement</u>



A while statement defines a boolean expression and a statement. If the expression is false the statement is not executed; otherwise, it is executed repeatedly until the expression becomes false.
The expression value must be a boolean.

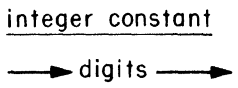


A repeat statement defines a sequence of statements and a boolean expression. The statements are executed at least once. If the expression is false, they are executed repeatedly until it becomes true.

The expression value must be a boolean.

7.1.3. Integers

The type INTEGER is a standard enumeration type. Its values are a finite set of successive, whole numbers represented by integer constants:



The following operators are defined for integers:

+	(plus sign or add)
-	(minus sign or subtract)
*	(multiply)
div	(divide)
mod	(modulo)

The result is an integer value.

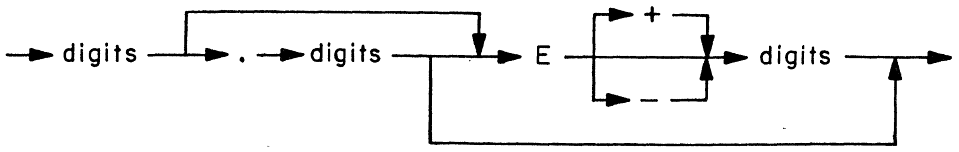
The following standard functions apply to integers:

abs(x)	The result (of type integer) is the absolute value of the integer x.
chr(x)	The result (of type char) is the character with the ordinal value x.
conv(x)	The result is the real value corresponding to the integer x.

7.2. Reals

The standard type REAL consists of a finite subset of the real numbers represented by real constants:

real constant



The letter E represents the scale factor 10.
The following operators are defined for reals:

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

+	(plus sign or add)
-	(minus sign or subtract)
*	(multiply)
/	(divide)

The result of a relation is a boolean value. The result of an arithmetic operation is a real value.

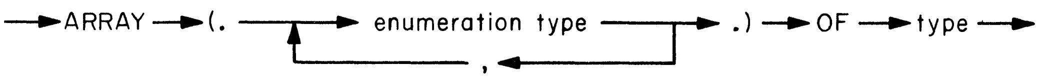
The following standard functions apply to reals:

abs(x)	The result (of type real) is the absolute value of the real x.
trunc(x)	The result is the (truncated) integer value corresponding to the real x.

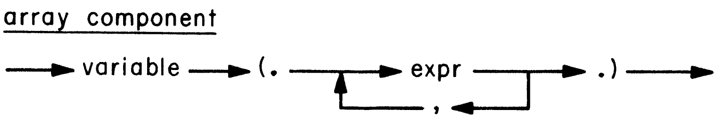
7.3. Array types

An array consists of a fixed number of components of the same type. An array component is selected by one or more index expressions.

array type



The index types must be enumeration types. The component type can be any type. The number of index types is called the dimension of the array.



A component of an n-dimensional array variable is selected by means of its variable identifier followed by n index expressions (enclosed in brackets and separated by commaes).

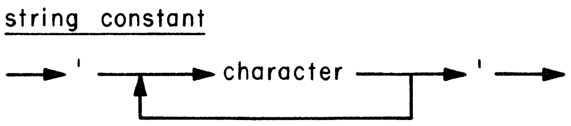
The number of index expressions must equal the number of index types in the array type definition, and the expressions must be compatible with the corresponding types.

The basic operators for arrays are:

:=	(assignment)
=	(equal)
<>	(not equal)

The operands must be passive, compatible arrays. The result of a relation is a boolean value.

A one-dimensional array of m characters is called a string type of length m. Its values are the string constants of length m:



The ordering of characters defines the ordering of strings.

The following operators are defined for strings (in addition to those defined for all array types);

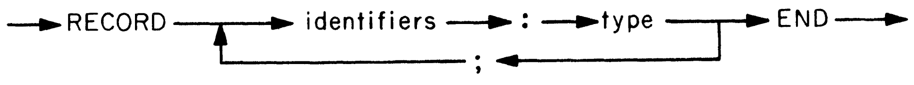
<	(less)
>	(greater)
<=	(less or equal)
>=	(greater or equal)

The operands must be strings of the same length. The result of a relation is a boolean value.

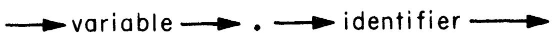
7.4. Record types

A record consists of a fixed number of components of (possibly) different types:

record type



record component



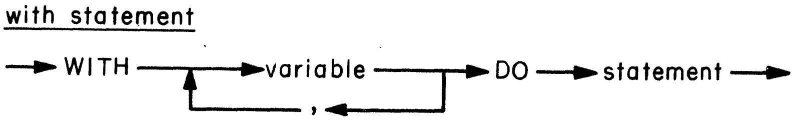
The components of a record type are called its fields. A field of a record variable is selected by means of its variable identifier followed by the field identifier (separated by a period).

The basic operators for records are:

:=	(assignment)
=	(equal)
<>	(not equal)

The operands must be passive, compatible records. The result of a relation is a boolean value.

A with statement can be used to operate on the fields of a record variable:



A with statement consists of one or more record variables and a statement. This statement can refer to the record fields by their identifiers only (without qualifying them with the identifiers of the record variables).

The statement

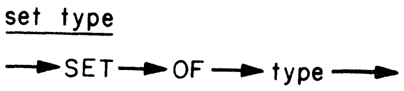
```
with v1, v2, ... , vn do S
```

is equivalent to

```
with v1 do
  with v2, ... , vn do S
```

7.5. Set types

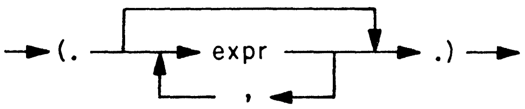
The set type of an enumeration type consists of all the subsets that can be formed of the enumeration values:



The component type of a set type is called its base type. It must be an enumeration type.

Set values can be constructed as follows:

set constructor



A set constructor consists of one or more expressions enclosed in brackets and separated by commaes. It computes the set consisting of the expression values. The set expressions must be of compatible enumeration types.

The empty set is denoted

(..)

The basic operators for sets are:

:=	(assignment)
<=	(contained in)
>=	(contains)
-	(difference)
&	(intersection)
or	(union)

The operands must be compatible sets. The result of a relation is a boolean value. The result of the other operators is a set value that is compatible with the operands.

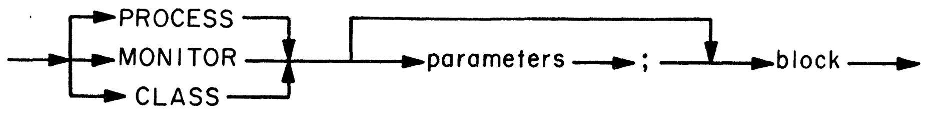
in	(membership)

The first operand must be an enumeration type and the second one must be its set type. The result is a boolean value.

7.6. System types

A concurrent program consists of three kinds of system types:

system type



A process type defines a data structure and a sequential statement that can operate on it.

A monitor type defines a data structure and the operations that can be performed on it by concurrent processes. These operations can synchronize processes and exchange data among them.

A class type defines a data structure and the operations that can be performed on it by a single process or monitor. These operations provide controlled access to the data.

A system type consists of the following components:

Parameters that represent constants and other system types on which the system type can operate. They are called the access rights of the system type.

Constants, data types, variables, and routines that are accessible within the system type (but generally not outside it). (The variable entries defined in Section 8.2 are the only exception to this rule.)

Routine entries that are accessible outside the system type (but not within it). These routines define meaningful operations on the system type that can be performed by other system types.

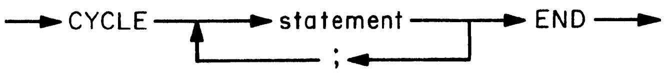
An initial statement to be executed when a variable of the system type is initialized.

In general, a system type parameter must be a constant parameter of type enumeration, real, set, or monitor (Section 11). In addition, a class type can be a parameter of another class type.

A system type can only be declared within another system type (but not within a record type or routine).

A process type can repeat the execution of a set of statements forever. This is done by means of the cycle statement:

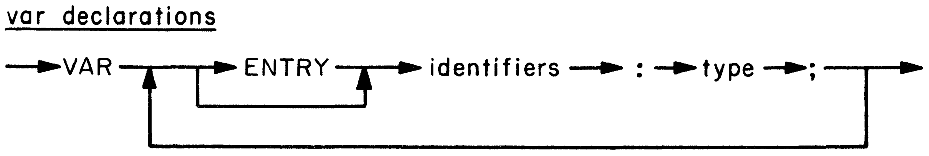
cycle statement



A cycle statement defines a sequence of statements to be executed repeatedly forever.

8. VARIABLES

A variable is a named store location that can assume 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.

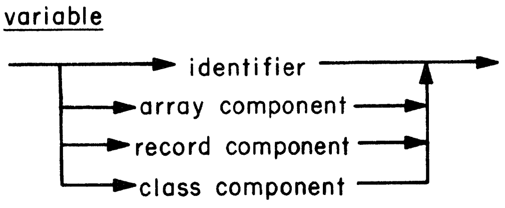


A variable declaration defines the identifier and type of a variable.
The meaning of a variable entry is defined in Section 8.2.
The declaration

```
var v1, v2, ... , vn: T;
```

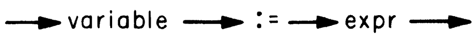
is equivalent to

```
var v1: T; v2: T; ... ; vn: T;
```



A variable is referenced by means of its identifier. A variable component is selected by means of index expressions or field identifiers (Sections 7.3, 7.4, 8.2).

assignment



An assignment defines the assignment of an expression value to a variable. The variable and the expression must be compatible.

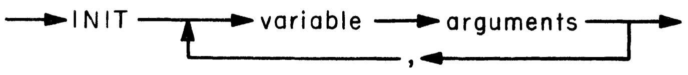
The variable must be of passive type. It may not be a constant parameter or a variable entry referenced by selection (Sections 7, 8.2, 11).

8.1. System components

A variable of system type is called a system component. It is either a process, a monitor, or a class.

System components are initialized by means of init statements:

init statement



An init statement defines the access rights of a system component (by means of arguments), allocates space for its variables, and executes its initial statement.

The statement

```
init v1, v2, ... , vn
```

is equivalent to

```
init v1; init v2, ... , vn
```

The initial statement of a class or monitor is executed as a nameless routine. The initial statement of a process is executed as a sequential process. This process is executed concurrently with all other processes (including the one that initialized it).

The parameters and variables of a system component exist forever after initialization. They are permanent variables.
A system component must be declared as a permanent variable within a system type. It cannot be declared as a temporary variable within a routine.

A system component can only be initialized once. This must be done in the initial statement of the system type in which it is declared.

8.2. Variable entries

A variable prefixed with the word ENTRY is a variable entry:

```
var entry f: T
```

It must be declared as a permanent variable of passive type within a class type.

A class type can refer to one of its own variable entries by means of its identifier f:

```
f
```

Outside the class type, a variable entry f of a class variable v can be selected either by means of the class identifier v followed by the entry identifier f (separated by a period):

```
v.f
```

or by means of a with statement

```
with v do begin ... f ... end
```

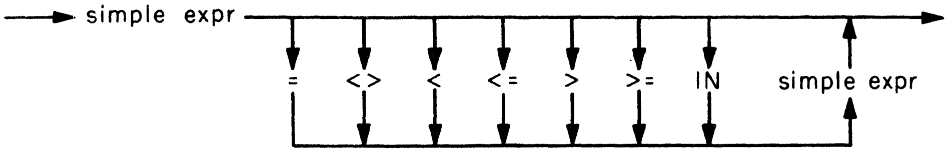
A class type can make assignment to its variable entries, but outside it they can only be referenced (but not changed) by selection. So a variable entry is similar to a function entry (Section 11).

9. EXPRESSIONS

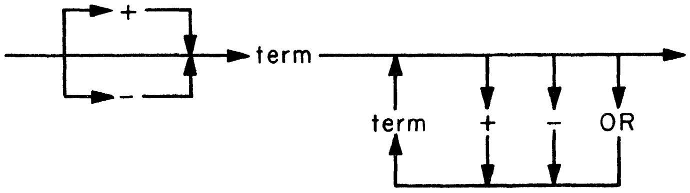
An expression defines a computation of a value by application of operators to operands. It is evaluated from left to right using the following priority rules:

First, factors are evaluated.
Secondly, terms are evaluated.
Thirdly, simple expressions are evaluated.
Fourthly, complete expressions are evaluated.

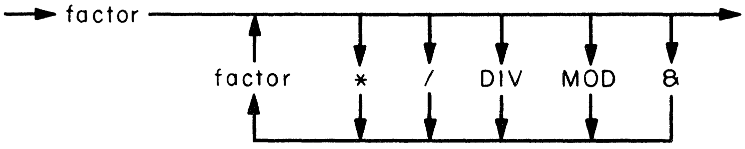
expr



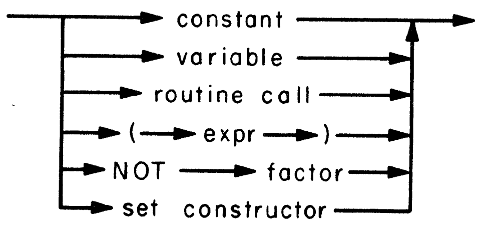
simple expr



term



factor



Type compatibility

An operation can only be performed on two operands if their data types are compatible. They are compatible if one of the following conditions is satisfied:

1) Both types are defined by the same type definition or variable declaration (Sections 7, 8).

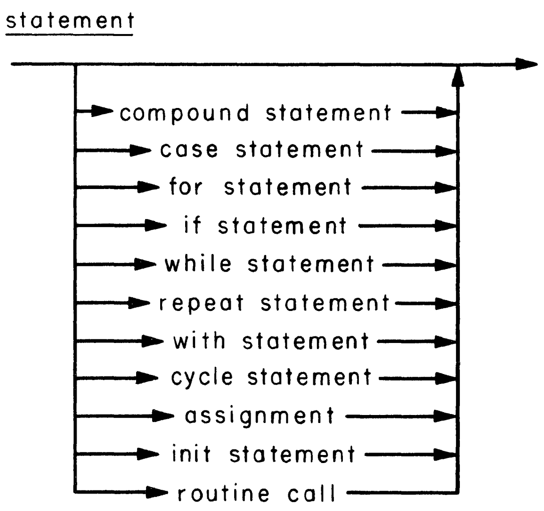
2) Both types are subranges of a single enumeration type (Section 7.1).

3) Both types are strings of the same length (Section 7.3).

4) Both types are sets of compatible base types. The empty set is compatible with any set (Section 7.5).

10. STATEMENTS

Statements define operations on constants and variables:



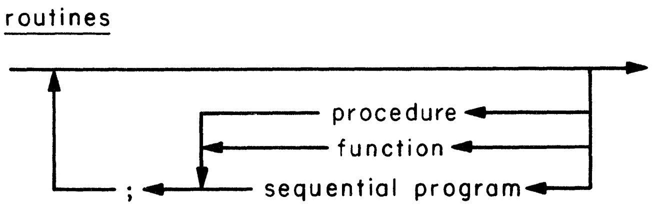
Section
5
7.1
7.1
7.1.2
7.1.2
7.1.2
7.4
7.6
8
8.1
11

Empty statements, assignments, and routine calls cannot be divided into smaller statements. They are simple statements. All other statements are structured statements formed by combinations of statements.

An empty statement has no effect.

11. ROUTINES

A routine defines a set of parameters and a compound statement that operates on them:



A routine can only be defined within a system type (but not within another routine).

A system component cannot reference the variables of another system component (except if they are variable entries of a class as defined in Section 8.2).

A system component can, however, call routine entries declared within other system types. There are four kinds of routine entries:

A process entry is a routine entry declared within a process type. It can only be called by sequential programs executed by a process of that type (but it cannot be called by system components).

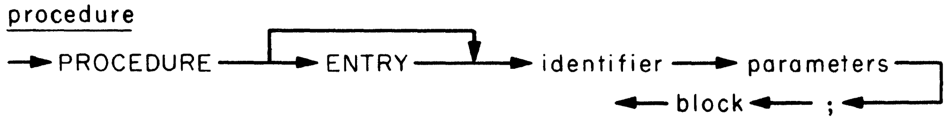
A monitor entry is a routine entry declared within a monitor type. It can be called simultaneously by one or more system components that wish to operate on a monitor of that type. A monitor entry has exclusive access to permanent monitor variables while it is being executed. If concurrent processes simultaneously call monitor routines that operate on the same permanent variables, the calls will be executed strictly one at a time.

A class entry is a routine entry declared within a class type. It can only be called by one system component at a time. So a class entry has also exclusive access to permanent class variables while it is being executed. But, in contrast to a monitor entry, the exclusive access of a class entry can be ensured during compilation (and not during execution).

An <u>initial statement</u> of a system type is a nameless routine entry called by means of the init statement (Section 8.1).

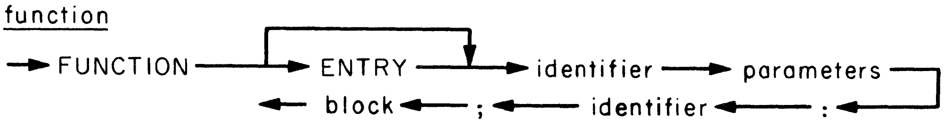
There are three kinds of routines: procedures, functions, and sequential programs.

<u>procedure</u>



A procedure consists of a procedure identifier, a parameter list, and a block to be executed when the procedure is called.

<u>function</u>



A function consists of a function identifier, a parameter list, a function type identifier, and a block to be executed when the function is called.

A function computes a value. The value e of a function f is defined by an <u>assignment</u>

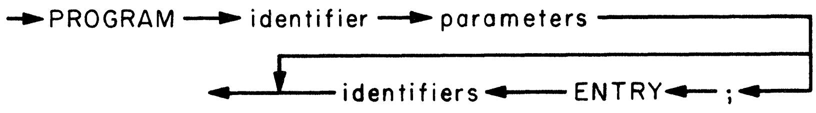
f:= e

within the function block.

The function and its value must be of compatible enumeration types.

A process that controls the execution of a compiled sequential program is called a job process. The process definition must include a declaration of the sequential program:

sequential program

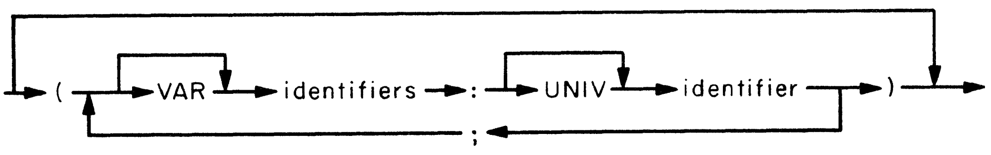


A program declaration consists of a program identifier, a parameter list, and a set of access rights.

Program parameters must be of passive types. The right-most parameter represents the variable in which the code of the sequential program is stored. It cannot be referenced by the sequential program itself.

The access rights of a program is a list of identifiers of routine entries defined within the job process that contains the program declaration. The sequential program may call these routines during its execution.

parameters



A parameter list defines the type of parameters on which a routine can operate. Each parameter is specified by its parameter and type identifiers (separated by a colon).

A variable parameter represents a variable to which the routine may assign a value. It is prefixed with the word VAR. The parameter declaration

```
var v1, v2, ... , vn: T
```

is equivalent to

```
var v1: T; var v2, ... , vn: T
```

A constant parameter represents an expression that is evaluated when the routine is called. Its value cannot be changed by the routine. A constant parameter is not prefixed with the word VAR.
The parameter declaration

```
v1, v2, ... , vn: T
```

is equivalent to

```
v1: T; v2, ... , vn: T
```

A parameter is of universal type if its type identifier is prefixed with the word UNIV. The meaning of universal types is explained later.

The parameters and variables declared within a routine exist only while it is being executed. They are temporary variables.

The permanent parameters of a system type define all other system types with which it can interact. A system type interacts with another system type when it calls a routine entry declared within the other system type.

Permanent parameters of system types must be constant parameters of type enumeration, real, set, or monitor. In addition, a class type can be a parameter of another class type.

Parameters of routine entries may not contain queues as components.

Function parameters must be constant.

Program parameters and parameters of universal type must be passive (Section 7).

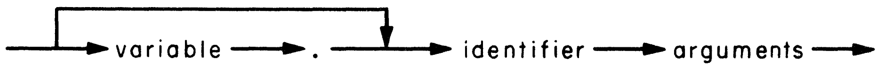
Universal parameters

The prefix UNIV suppresses compatibility checking of parameter and argument types in routine calls (Sections 9, 11).

An argument of type T1 is compatible with a parameter of universal type T2 if both types are passive and represented by the same number of store locations.

The type checking is only suppressed in routine calls. Inside the given routine the parameter is considered to be of non-universal type T2, and outside the routine call the argument is considered to be of non-universal type T1.

routine call



A routine call specifies the execution of a routine with a set of arguments. It can either be a function call, a procedure call, or a program call.

A routine that is not prefixed with the word ENTRY is a simple routine. A system type can call one of its own simple routines by means of its identifier P followed by a list of arguments a1, ... , an:

P(a1, ... , an)

A system type can call a routine entry declared within another system type T by qualifying the call with the identifier v of a variable of type T:

v.P(a1, ... , an)

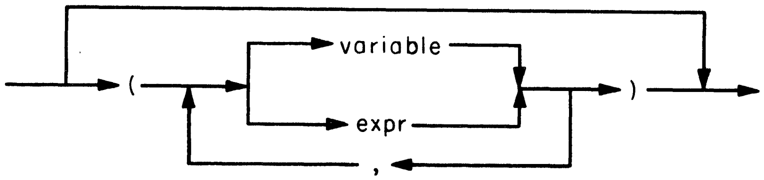
or by a with statement

with v do begin ... P(a1, ... , an) ... end

A routine may not call itself, and a system type may not call its own routine entries.

A routine call used as a factor in an expression must be a function call. A routine call used as a statement must be a procedure call (Sections 9, 10).

arguments



An argument list defines the arguments used in a routine call. The number of arguments must equal the number of parameters specified in the routine. The arguments are substituted for the parameters before the routine is executed.

Arguments corresponding to variable and constant parameters must be variables and expressions, respectively. The selection of variable arguments and the evaluation of constant arguments are done once only (before the routine is executed).

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

An argument corresponding to a constant string parameter may be a string of any length.

An argument corresponding to a universal parameter may be of any passive type that occupies the same number of store locations as the parameter type.

12. QUEUES

The standard type QUEUE may be used within a monitor type to delay and resume the execution of a calling process within a routine entry (Sections 7.6, 11).

At most one process at a time can wait in a single queue. A queue is either empty or non-empty. Initially, it is empty.

A variable of type queue can only be declared as a permanent variable within 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)	The calling process 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.
continue(x)	The calling process returns from the monitor routine that performs the continue operation. If another process is waiting in the queue x that process will immediately resume its execution of the monitor routine that delayed it. The resumed process now again has exclusive access to the monitor variables.

13. SCOPE RULES

A scope is a region of program text in which an identifier is used with a single meaning. An identifier must be introduced before it is used. (The only exception to this rule is a sequential program declaration within a process type: it may refer to routine entries defined later in the same process type. This allows one to call sequential programs recursively.)

A scope is either a system type, a routine, or a with statement. A system type or routine introduces identifiers by declaration; a with statement does it by selection (Sections 5, 7.4, 7.6, 8.2, 11).

When a scope is defined within another scope we have an outer scope and an inner scope that are nested. An identifier can only be introduced with one meaning in a scope. It can, however, be introduced with another meaning in an inner scope. In that case, the inner meaning applies in the inner scope and the outer meaning applies in the outer scope.

System types can be nested, but routines cannot. Within a routine, with statements can be nested. This leads to the following hierarchy of scopes:

```
(nested system types
  (non-nested routines
    (nested with statements)))
```

A system type can use

(1) all constant and type identifiers introduced in its outer scopes.

(2) all identifiers introduced within the system type itself (except its routine entry identifiers).

A routine can use

(1), (2) defined above and

(3) all identifiers introduced within the routine itself (except the routine identifier).

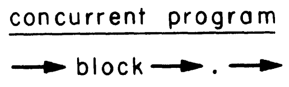
A <u>with statement</u> can use

(1), (2), (3) defined above and
(4) all identifiers introduced by the with statement itself and by its outer with statements.

The phrase "all identifiers introduced in its outer scopes" should be qualified with the phrase "unless these identifiers are used with different meanings in these scopes. In that case, the innermost meaning of each identifier applies in the given scope."

14. CONCURRENT PROGRAMS

The outermost scope of a concurrent program is an anonymous, parameterless process type, called the <u>initial process</u>:



An instance of this process is automatically initialized after program loading. Its purpose is to initialize other system components.

A. PDP 11/45 SYSTEM

This appendix defines additional restrictions and extensions of Concurrent Pascal for the PDP 11/45 computer.

A.1. Language restrictions

A non-standard enumeration type can at most consist of 128 constant identifiers.

The range of integers is -32768..32767.

Integer case labels must be in the range 0..127.

The range of reals is approximately $-10^{38}..10^{38}$. The smallest absolute real value that is non-zero is approximately 10^{-38}. The relative precision of a real is approximately 10^{-16}.

A string must contain an even number of characters.

Enumeration types and system types cannot be defined within record types.

A set of integers can only include members in the range 0..127.

A process component can only be declared within the initial process.

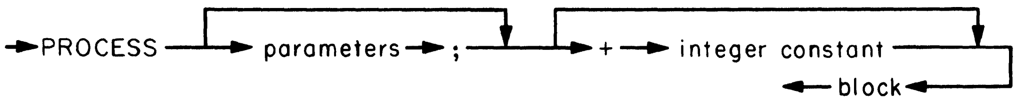
The standard procedure continue can only be called within a routine entry of a monitor type.

A.2. Store allocation

The compiler determines the store requirements of system components under the assumption that routine calls are not recursive. The scope rules prevent recursion within concurrent programs, but not within sequential programs.

The programmer must estimate an additional data space needed to execute sequential programs within a job process. The data space of a sequential program (in bytes) is defined by an integer constant after the process parameters:

process type



A.3. Process attributes

The standard function

attribute(x)

defines an attribute x of the calling process. The index and value of the attribute are universal enumerations.

At present, the attribute index is of the following type:

```
type attrindex =
  (caller, heaptop, progline, progresult, runtime)
```

The meaning of these attributes is defined in the sequel.

The attribute function can be used to identify the calling process:

attribute(caller)	The result is an integer that identifies the calling process. The machine associates consecutive integers 1, 2, .. with processes during their initialization starting with the initial process.

A.4. Heap control

Associated with every process is a heap in which sequential Pascal programs can allocate semi-permanent data structures (by means of a standard procedure new that is not available in Concurrent Pascal).

A process can measure the extent of its heap by means of the standard function attribute:

attribute(heaptop)	The result is an integer defining the top address of the heap.

The heap top can be reset to a previous value by means of the standard procedure

setheap(x)	The top address of the heap is set equal to the integer x (defined by a previous call of attribute): x:= attribute(heaptop)

This crude mechanism is intended mainly to enable a job process to measure the initial extent of its heap before it executes a sequential program and to reset the heap when the program terminates.

A.5. Program termination

When a sequential program terminates its job process can call the standard function attribute to determine the line number in which the program terminated and its result:

```
attribute(progline)
attribute(progresult)
```

The line attribute is an integer and the progresult is of the following type:

```
type resulttype =
  (terminated, overflow, pointererror, rangeerror,
   varianterror, heaplimit, stacklimit)
```

The result values have the following meaning:

terminated	Correct termination.
overflow	An integer or real is out of range.
pointererror	A variable is referenced by means of a pointer with the value <u>nil</u>.
rangeerror	An enumeration value is out of range.
varianterror	A reference to a field of a variant record is incompatible with its tag value.
heaplimit	The heap capacity is exceeded.
stacklimit	The stack capacity is exceeded.

The above are <u>standard results</u> of sequential Pascal programs generated by the machine. A concurrent program may, however, extend the result type with <u>non-standard values</u>, for example:

```
type resulttype =
  (terminated, overflow, pointererror, rangeerror,
   varianterror, heaplimit, stacklimit, codelimit,
   timelimit, printlimit)
```

Non-standard results can be used as arguments to the standard procedure <u>stop</u> (defined below).

The following <u>standard procedures</u> control <u>program preemption</u>:

start	Prevents preemption of a sequential program to be executed by the calling process.
stop(x, y)	Preempts a sequential program called by process x with the result y. The process identity must have been defined earlier by a call of attribute x:= attribute(caller)

<u>Start</u> should be called before a sequential program is executed. If <u>stop</u> is called while a sequential program is executing a routine entry within its job process, preemption is delayed until the routine call has been completed.

A.6. Real time control

The standard routines for real time control are:

wait	The calling process is delayed until the next 1-second signal from a clock. (If the waiting is done within a monitor this will delay other calls of the same monitor).
realtime	The result is an integer defining the real time (in seconds) since system initialization.

The standard function attribute can be used to define the run time of the calling process:

attribute(runtime)	The result is an integer defining the processor time (in seconds) used by the calling process since its initialization. (This is only accurate on a machine with a readable clock.)

A.7. Input/output

Input/output is handled by means of the following standard procedure:

io(x, y, z)	Peripheral device z performs the operation y on variable x. The calling process is delayed until the operation is completed. (If the io is done within a monitor, it will delay other calls of the same monitor.) x and y are variable parameters of arbitrary passive types. z is a constant parameter of arbitrary enumeration type.

At present, the machine assumes that the io device z and the io parameter y are of the following types:

```
type iodevice =
  (terminal1, disk1, tape1, printer1, reader1)

type ioparam = record
                 operation: iooperation;
                 result: ioresult;
                 arg: ioarg
               end
```

where

```
type iooperation = (input, output, move, control)

type ioresult =
  (complete, intervention, transmission, failure,
   endfile, endmedium, startmedium)
```

The io results have the following meaning:

complete	The operation succeeded.
intervention	The operation failed; the device requires manual intervention before the operation can be repeated.
transmission	The operation failed; the transmission error can probably be corrected by repeating the operation immediately.
failure	The operation failed; the device failure cannot be corrected by repeating the operation.
endfile	An end of file mark was sensed.
endmedium	The end of medium mark was sensed.
startmedium	The start of medium mark was sensed.

The types of the io block x and the io argument within the io parameter vary from device to device.

A concurrent program must ensure that a device is used by at most one process at a time (wherever this rule applies).

A.7.1. Terminal

device	terminal1
block	A single character (ASCII representation).
input	Inputs a single character and echoes it back as output. The character CR is input as LF and echoed as CR, LF. The character BEL cannot be input (see below).
output	Outputs a single character. The character LF is output as CR, LF.
control	Delays the calling process until the BEL key is depressed. The BEL key can be depressed at any time (whether the terminal is passive, inputting, or outputting); it has no effect unless one or more processes are waiting for it.
result	complete

One or more control operations can be executed simultaneously with a single input/output operation. A BEL signal continues the execution of all processes waiting for it.

A.7.2. Disk

device	disk1
block	A universal string of 512 characters (called a disk page).
argument	An integer in the range 0..4799 (called a page index).
input	Inputs a disk page with a given page index.
output	Outputs a disk page with a given page index.
control	Starts execution of a concurrent program stored on consecutive disk pages identified by the first page index.
result	complete, intervention, transmission, or failure.

A disk can only perform one operation at a time.

The system uses the following algorithm to convert a page index to a physical disk address consisting of a surface number, cylinder number, and sector number:

```
     surface:= page_index div 12 mod 2;
     cylinder:= page_index div 24;
     sector:= page_index mod 12;
```

A.7.3. Magnetic tape

device	tape1
block	A universal string of 512 characters (called a tape block).
argument	An enumeration constant defining one of the following move operations: (outeof, rewind, upspace, backspace)
input	Inputs the next block from tape (if any).
output	Outputs the next block on tape (if there is room for it).
move	Moves the tape as defined by the argument:
outeof	Outputs an end of file mark (if there is room for it).
rewind	Rewinds the tape.
upspace	Moves the tape forward one block (or file mark), whichever occurs first.
backspace	Moves the tape backwards one block (or file mark), whichever occurs first.
result	complete, intervention, transmission, failure, endfile, endmedium, or startmedium.

A tape station can only perform one operation at a time.

A.7.4. Line Printer

device	printer1
block	A string of 132 characters (ASCII representation) (called a printer line).
output	Outputs a line of 132 characters (or less). A line of less than 132 characters must be terminated by a CR, LF, or FF character.
result	complete or intervention.

A line printer can only perform one operation at a time.

A.7.5. Card Reader

device	reader1
block	A string of 80 characters (ASCII representation) (called a punched card).
input	Inputs a card of 80 characters. Characters that have no graphic representation on a key punch are input as SUB characters.
result	complete, intervention, transmission, or failure.

A card reader can only perform one operation at a time.

A.8. Compiler characteristics

The compiler consists of 7 passes. It requires a code space of 9 K words and a data space of 7 K words. After a basic loading time of 7 sec the compilation speed is 240 char/sec (or about 9 - 10 lines/sec).

The programmer may prefix a program with compiler options enclosed in parantheses and separated by commaes:

(number, check, test)

The options have the following effect:

number	The generated code will only identify line numbers of the program text at the beginning of routines. (This reduces the code by about 25 per cent, but makes error location more difficult.)
check	The code will not make range checks of constant enumeration arguments.
test	The compiler will print the intermediate output of all passes. (This facility should be used as a diagnostic aid to locate compiler errors.)

A.9. Program characteristics

The following is the execution times of operand references, operators, and statements in usec (measured on a PDP 11/45 with 850 nsec core store). They exceed the figures stated in the computer programming manual by 25 per cent.

	enumeration	real	set (n members)	structure (n words)
constant c	7	39	53 + 32 n	17
variable v	10	32	46	10
field v.f	27	40	54	18
indexed v(.e.)	40 + e	53 + e	67 + e	31 + e
:=	8	0	0	10 + 5 n
= <>	12	32	67	16 + 6 n
< > <= >=	12	32	74	16 + 11 n
in			31	
succ pred	7			
&	10		82	
or	8		58	
not	10			
+ -	9	38	58	
*	16	45		
div mod /	20	46		
abs	7	17		
conv	21			
trunc		22		

	(n iterations)
case e of ... c: S; ... end	28 + e + S
for v:= 1 to n do S	82 + (69 + S) n
if B then S else S	16 + B + S
while B do S	(20 + B + S) n
repeat S until B	(13 + B + S) n
with v do S	16 + S
cycle S end	(7 + S) n
simple routine call	58
process entry call	75
class entry call	80
monitor entry call	200
empty	10
delay, continue (processor switching)	600
clock interrupt (every 17 msec)	900
io	1500

The compiler generates about 5 words of <u>code</u> per program line (including line numbers and range checks).

The store requirements of <u>data</u> types are:

enumeration	1 word(s)
real	4
set	8
string (m characters)	m/2

B. ASCII CHARACTER SET

0	nul	32		64	@	96	
1	soh	33	!	65	A	97	a
2	stx	34	"	66	B	98	b
3	etx	35	#	67	C	99	c
4	eot	36	$	68	D	100	d
5	enq	37	%	69	E	101	e
6	ack	38	&	70	F	102	f
7	bel	39	'	71	G	103	g
8	bs	40	(	72	H	104	h
9	ht	41	)	73	I	105	i
10	lf	42	*	74	J	106	j
11	vt	43	+	75	K	107	k
12	ff	44	,	76	L	108	l
13	cr	45	-	77	M	109	m
14	so	46	.	78	N	110	n
15	si	47	/	79	O	111	o
16	dle	48	0	80	P	112	p
17	dc1	49	1	81	Q	113	q
18	dc2	50	2	82	R	114	r
19	dc3	51	3	83	S	115	s
20	dc4	52	4	84	T	116	t
21	nak	53	5	85	U	117	u
22	syn	54	6	86	V	118	v
23	etb	55	7	87	W	119	w
24	can	56	8	88	X	120	x
25	em	57	9	89	Y	121	y
26	sub	58	:	90	Z	122	z
27	esc	59	;	91	[	123	{
28	fs	60	<	92	\	124	|
29	gs	61	=	93	]	125	}
30	rs	62	>	94	^	126	¬
31	us	63	?	95	_	127	del

INDEX

BIBLIOGRAPHIC DATA SHEET	1. Report No. NSF-OCA-DCR74-17331-CP1	2.	3. Recipient's Accession No.
4. Title and Subtitle Concurrent Pascal Report			5. Report Date June 1975
			6.
7. Author(s) Per Brinch Hansen			8. Performing Organization Rept. No. CIT-IS-TR 17
9. Performing Organization Name and Address Information Science 286-80 California Institute of Technology Pasadena, California 91125			10. Project/Task/Work Unit No.
			11. Contract/Grant No. NSF DCR74-17331
12. Sponsoring Organization Name and Address National Science Foundation Office of Computing Activities Washington, D. C. 20550			13. Type of Report & Period Covered 1. Edition
			14.

15. Supplementary Notes

16. Abstracts

This report defines Concurrent Pascal - an abstract programming language for structured programming of computer operating systems. It extends the sequential programming language Pascal with concurrent processes, monitors, and classes. Concurrent Pascal has been implemented for the PDP 11/45 computer at Caltech.

17. Key Words and Document Analysis. 17a. Descriptors

Concurrent Pascal, structured multiprogramming, programming languages, hierarchical operating systems, concurrent processes, monitors, classes, abstract data types, access rights, scheduling, job control.

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group

18. Availability Statement	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 60
	20. Security Class (This Page UNCLASSIFIED	22. Price