PER BRINCH HANSEN
ALFRED C. HARTMANN

Information Science
California Institute of Technology

July 1975

SEQUENTIAL PASCAL REPORT

SEQUENTIAL PASCAL REPORT

Per Brinch Hansen
Alfred C. Hartmann

Information Science
California Institute of Technology

July 1975

Abstract

This report defines the sequential programming language Pascal as implemented for the PDP-11/45 computer.

Key Words and Phrases: Pascal, programming languages.

CR Categories: 4.2

CONTENTS

1. Introduction

This report defines the sequential programming language Pascal implemented on the PDP-11/45 computer. Pascal is a general purpose language for structured programming invented by Niklaus Wirth.

This is a brief concise definition of Pascal. A more informal introduction to Pascal is provided by the following reports:

Wirth, N. Systematic Programming, Prentice-Hall, 1973.

Jensen, K. and Wirth, N. Pascal-User Manual and Report,
Lecture Notes in Computer Science 18, Springer-Verlag, 1974.

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.

Sequential 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

2. Syntax Graphs

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

while statement

→ WHILE → expr → DO → statement →

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

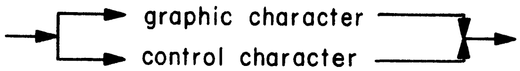
Correct *sequences* of basic symbols and constructs are represented by arrows.

3. CHARACTER SET

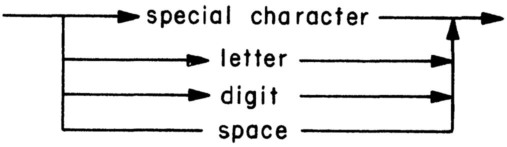
3. Character Set

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

character



graphic character



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

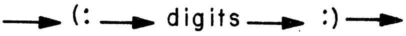
W X Y Z _

The digits are

0 1 2 3 4 5 6 7 8 9

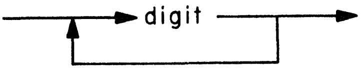
3. CHARACTER SET

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.

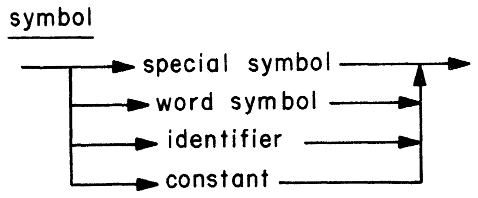
digits



4. BASIC SYMBOLS

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

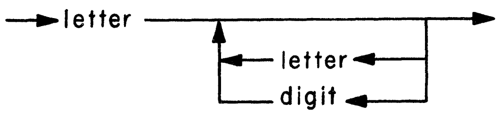
ARRAY	BEGIN	CASE	CONST	DIV
DO	DOWNTO	ELSE	END	FOR
FORWARD	FUNCTION	IF	IN	MOD
NOT	OF	OR	PROCEDURE	PROGRAM
RECORD	REPEAT	SET	THEN	TO
TYPE	UNIV	UNTIL	VAR	WHILE
WITH				

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

Word symbols cannot be used as identifiers.

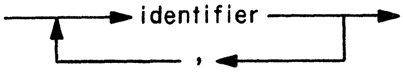
4. BASIC SYMBOLS

identifier

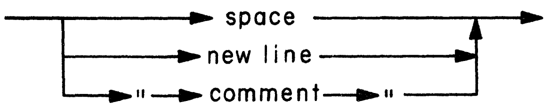


An identifier is introduced by a programmer as the name of a constant, type, variable, or routine.

identifiers



separator



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.

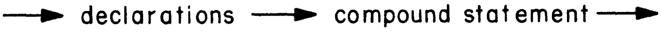
A comment is any sequence of graphic characters (except ") enclosed in quotes. It has no effect on the execution of a program.

5. BLOCKS

5. Blocks

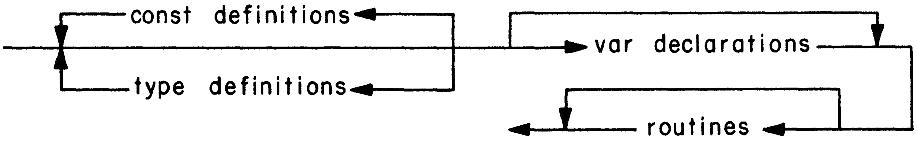
The basic program unit is a block:

block



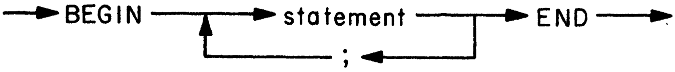
It consists of declarations of computational objects and a compound statement that operates on them.

declarations



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

compound statement

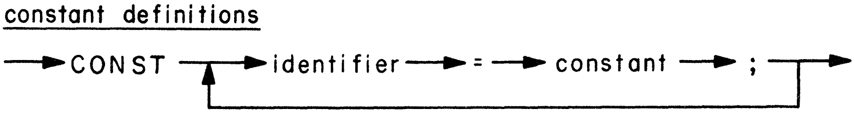


A compound statement defines a sequence of statements to be executed one at a time from left to right.

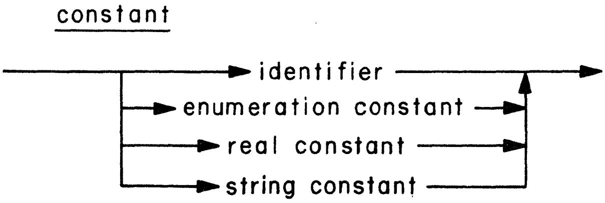
6. CONSTANTS

6. Constants

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



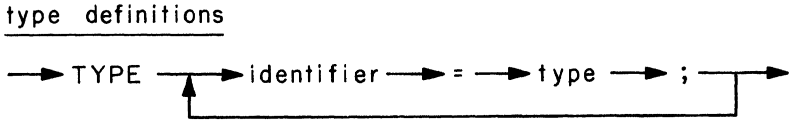
A constant definition introduces an identifier as the name of a constant.



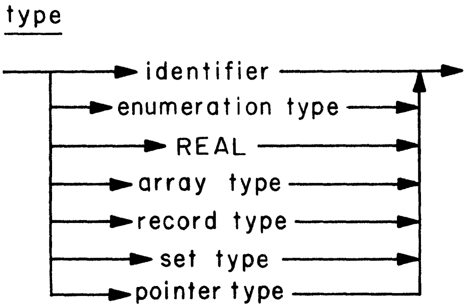
7. TYPES

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. In general, a data type cannot refer to its own type identifier. A pointer type may however refer to a data type before it has been defined.



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

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

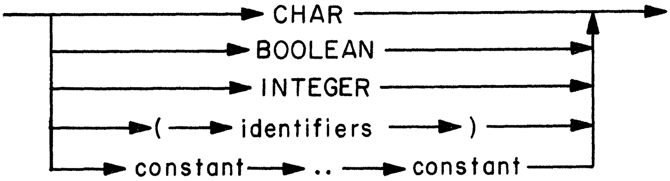
A data type that contains a pointer type is a list type. All other types are nonlist types.

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.

enumeration type

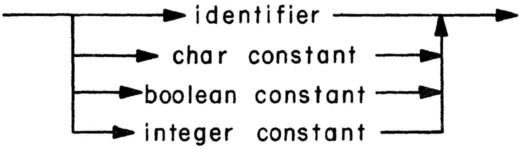


The types char, boolean, and integer are standard enumeration types.

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).

enumeration constant



7.1. ENUMERATION TYPES

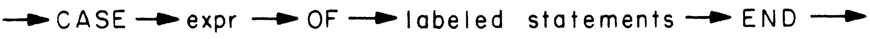
The basic operators for enumerations are:

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

The result of a relation is a boolean value.

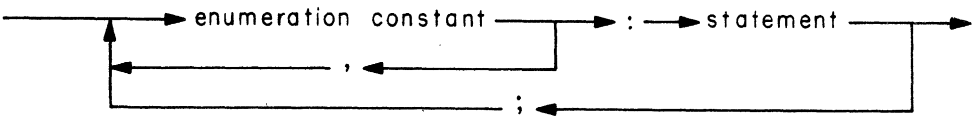
An enumeration value can be used to select one of several statements for execution:

case statement



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).

labeled statements



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

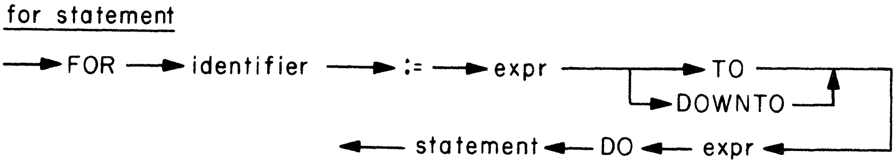
7.1. ENUMERATION TYPES

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:



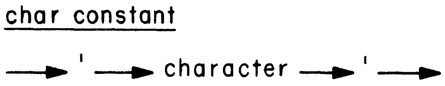
A for statement consists of an identifier of a control variable, two expressions defining a subrange, and a statement to be executed repeatedly for successive values in the subrange.

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 an array element (Sections 7.3, 7.4, 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:



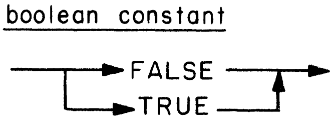
The following standard function applies to characters:

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:



where FALSE<TRUE.

The following operators are defined for booleans:

& (and)

or

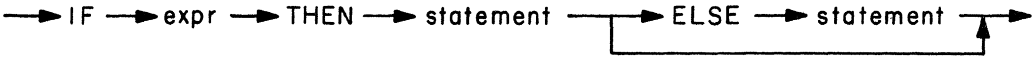
not

The result is a boolean value.

7.1.2. BOOLEANS

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).

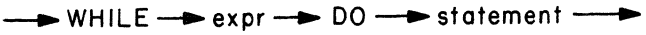
if statement



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.

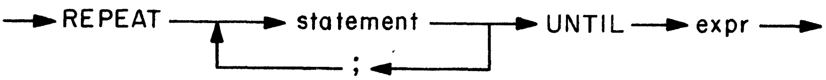
while statement



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.

repeat statement



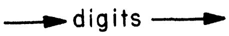
A repeat statement defines a sequence of statements and a boolean expression. The statements are executed at least once. If the expression is false, they are executed repeatedly until it becomes true.

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:

integer constant



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.

7.2 REALS

The following standard functions apply to integers:

abs(x) The result (of type integer) is the absolute value of the integer x.

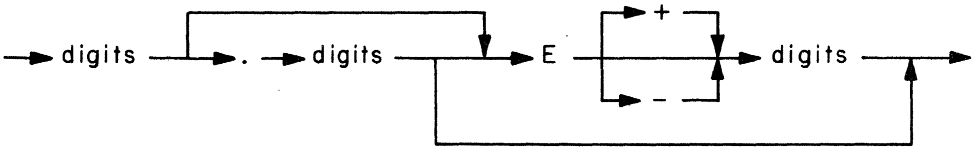
chr(x) The result (of type char) is the character with the ordinal value x.

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.

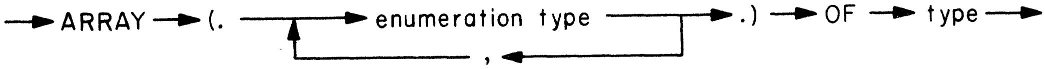
The following *standard functions* apply to reals:

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*.

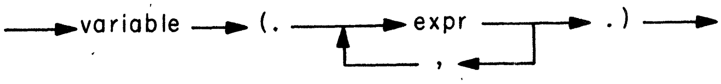
array type



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.

7.3. ARRAY TYPES

array component



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 commas).

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.

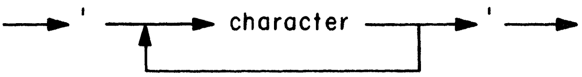
The basic operators for arrays are:

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

The operands must be 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:

string constant



The ordering of characters defines the ordering of strings.

7.4. RECORD TYPES

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

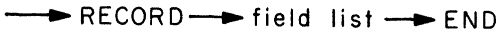
<	(less)
>	(greater)
<=	(less or equal)
>=	(greater or equal)

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

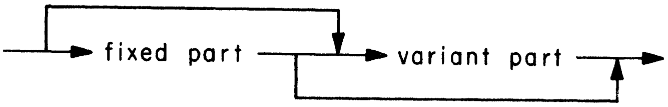
7.4. Record Types

A record consists of a fixed part and a variant part. One of these (but not both) can be missing.

record type

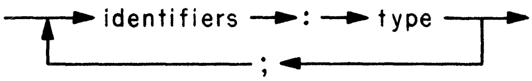


field list

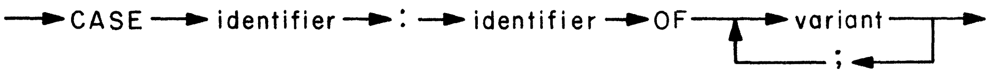


The fixed part consists of fields of fixed types.

fixed part



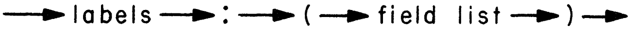
variant part



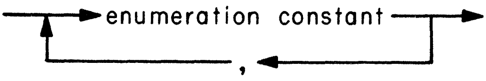
7.4. RECORD TYPES

The variant part defines a tag field and one or more different sets of fields (called variants). Each possible variant is labeled by one or more constants. A record of this type can represent any one of the variants. The value of the tag field defines the chosen variant.

variant



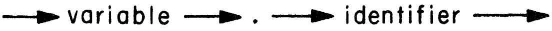
labels



The tag field and the labels must be of compatible enumeration types, and the labels must be unique.

A field of a record variable is selected by means of its variable identifier followed by the field identifier (separated by a period).

record component



A variant field can only be selected if the value of the tag field is equal to one of the labels of that variant.

The basic operators for records are:

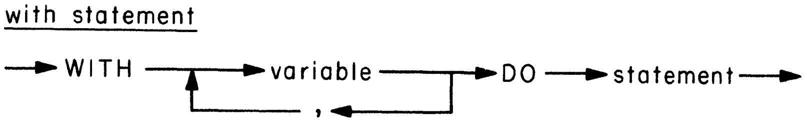
:= (assignment)

= (equal)

<> (not equal)

The operands must be 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

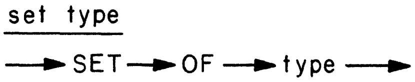
```
with vl, v2, ... , vn do S
```

is equivalent to

```
with vl 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:

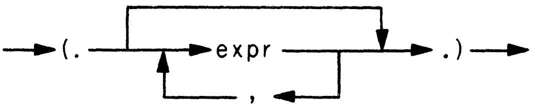


7.5. SET TYPES

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 commas. 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. Pointer Types

A pointer type is a reference to another type:

pointer type

→ @ → type →

pointer component

→ variable → @ →

The type referenced by a pointer is its *component type*. The component of a pointer variable is *selected* by means of its variable identifier followed by the symbol @.

The basic *operators* for pointers are:

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

The operands must be pointers to compatible components.

An assignment associates the component of one pointer variable with another pointer variable as well.

Two pointers are equal if both are associated with the same component. The result of a pointer comparison is a boolean.

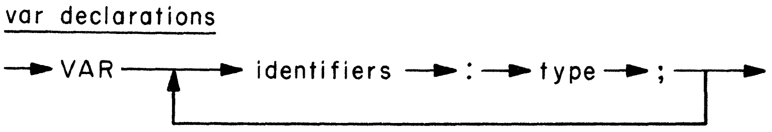
The pointer constant NIL denotes an undefined component. Initially all pointer variables have the value NIL. They may get a new value by assignment or by the standard procedure:

new(p) Associates a new component with the pointer variable p.

8. VARIABLES

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.

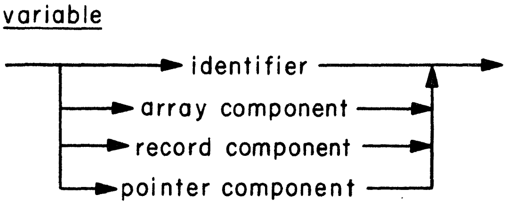


A variable declaration defines the identifier and type of a variable. The declaration

var vl, v2, ... , vn: T;

is equivalent to

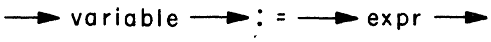
var vl: T; v2: T; ... ; vn: T;



A variable is referenced by means of its identifier. A variable component is selected by means of index expressions, field identifiers, or pointer references (Sections 7.3, 7.4, 7.6).

8. VARIABLES

assignment



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

The variable may not be a constant parameter (Section 11).

9. EXPRESSIONS

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:

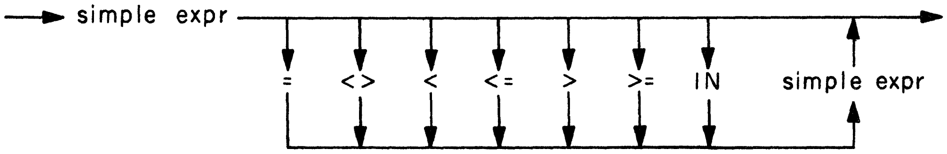
First, factors are evaluated.

Secondly, terms are evaluated.

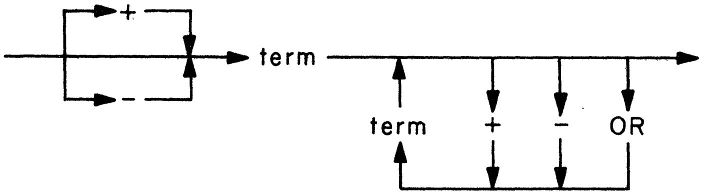
Thirdly, simple expressions are evaluated.

Fourthly, complete expressions are evaluated.

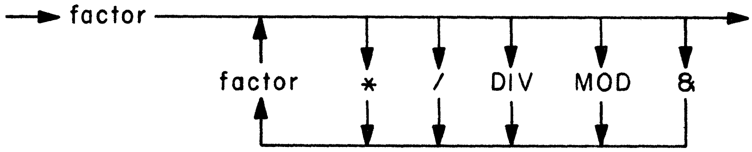
expr



simple expr

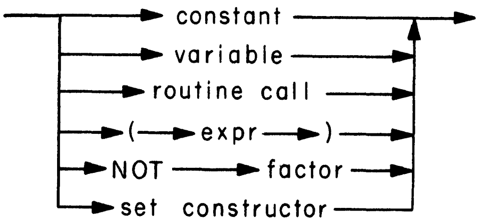


term



9. EXPRESSIONS

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).

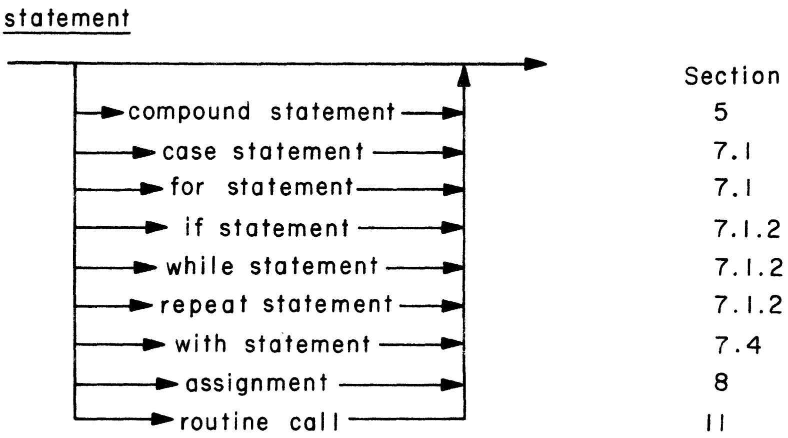
3) Both types are strings of the same length (Section 7.3).

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

10. STATEMENTS

10. Statements

Statements define operations on constants and variables:



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.

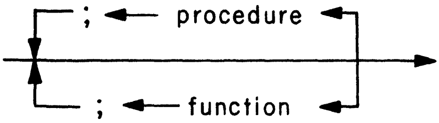
An empty statement has no effect.

11. ROUTINES

11. Routines

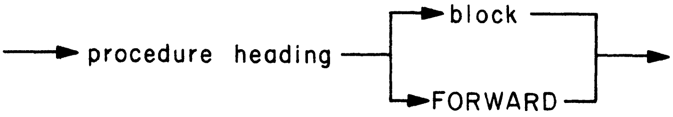
A routine defines a set of parameters and a block that operates on them. In the case of prefix routines (Section 13) and forward declarations, the block is omitted.

routines



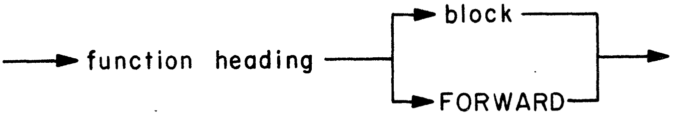
There are two kinds of routines, procedures and functions. A procedure consists of a procedure heading and a block to be executed when the procedure is called:

procedure



A function consists of a function heading and a block to be executed when the function is called:

function

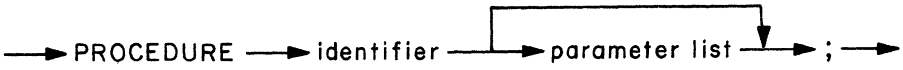


11. ROUTINES

If a routine is referenced before its block is defined, it must be introduced first by means of its heading followed by the symbol FORWARD. The routine can then be completed later by repeating its heading (without the parameter list) following by the block.

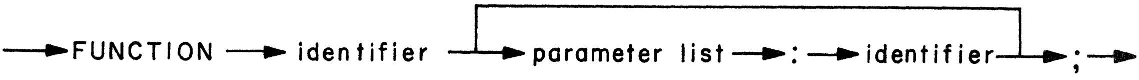
A procedure heading defines the procedure identifier and its parameter list.

procedure heading



A function heading gives the function identifier, its parameter list, and the function type.

function heading



A function computes a value. The value e of a function f is defined by an assignment

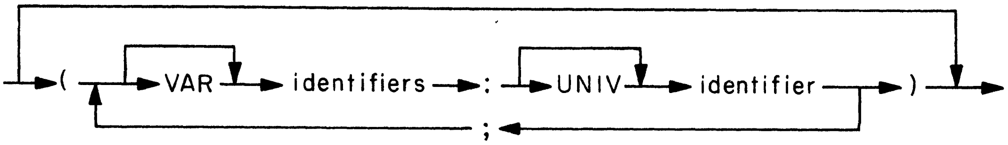
f: = e

within the function block.

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

11. ROUTINES

parameter list



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 vl, v2, ... , vn: T

is equivalent to

var vl: 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

vl, v2, ... , vn: T

is equivalent to

vl: 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.

11. ROUTINES

Function parameters must be constant.

Universal types must be nonlist types.

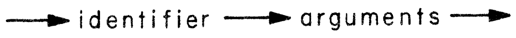
Universal Parameters

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 nonlist types 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.

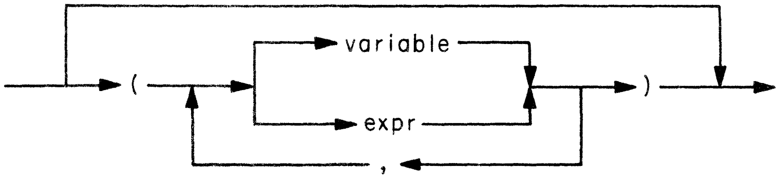
routine call



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

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



11. ROUTINES

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 nonlist type that occupies the same number of store locations as the parameter type.

12. SCOPE RULES

12. 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 pointer type: it may refer to a type that has not yet been defined).

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

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

(program

(non-nested routines

(nested with statements)))

A program can use

(1) any standard identifier.

(2) constant, type, and routine identifiers introduced within and after the prefix (Section 13).

A routine can use

(1), (2) defined above and

(3) all identifiers introduced within the routine itself.

12. SCOPE RULES

A with statement can use

(1), (2), (3) defined above and

(4) all identifiers introduced by the with statement itself and by its outer with statements.

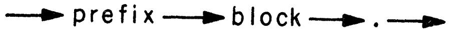
The phrase "all identifiers introduced in its outer scopes" should be qualified with the phrase "unless these identifiers are used with different meanings in these scopes. In that case, the innermost meaning of each identifier applies in the given scope."

13. SEQUENTIAL PROGRAMS

13. Sequential Programs

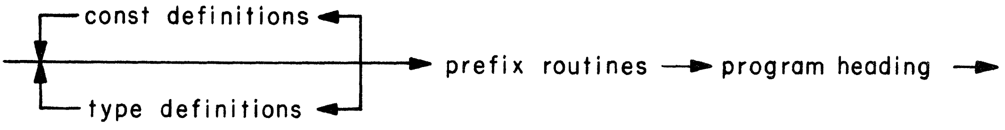
A sequential program consists of a prefix followed by a block:

program

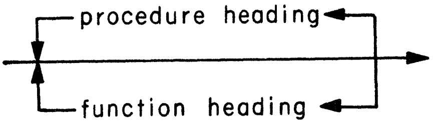


The prefix defines the program's interface to the operating system. This interface consists of constant, type, and routine definitions:

prefix



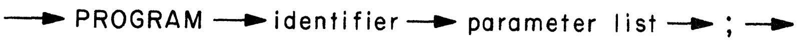
prefix routines



Prefix routines consist only of procedure or function headings. The prefix routine blocks are defined within the operating system. They can be called as other routines by the program (Section 11).

The program heading gives the program identifier and its parameter list:

program heading



A. PDP 11/45 SYSTEM

A. PDP 11/45 System

This appendix defines additional restrictions and extensions of Sequential 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 cannot be defined within record types.

A non-standard enumeration type used as a tag field type can contain at most 16 constant identifiers.

Integer variant labels must be in the range 0..15.

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

A. 2. Compiler Characteristics

The compiler consists of 7 passes. It requires a code space of 9 K words and a data space of 8 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 commas:

```
(number, check, test)
```

A.2. COMPILER CHARACTERISTICS

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 the following checks:

a) range checks of constant enumeration arguments;

b) pointer checks to insure that NIL-valued pointers are not used as references;

c) variant checks to insure that only currently defined variant fields are referenced.

The code will not initialize pointer variables to NIL.

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. 3. PROGRAM CHARACTERISTICS

A. 3.Program Characteristics

The following are 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 + 17 l	40 + 17 l	54 + 17 l	18 + 17 l
(l levels of variant nesting)				
indexed v(. e.)	40 + e	53 + e	67 + e	31 + e
pointer p@	26	40	54	18
:=	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		

A. 3. PROGRAM CHARACTERISTICS

	(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
routine call	58
prefix routine call	75
clock interrupt (every 17 msec)	900
new(p) -n words	39 + 4 n

The compiler generates about 5 words of code per program line (including line numbers and checks).

The store requirements of data 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

INDEX

INDEX