PASCAL USERS GROUP

PASCAL NEWS

NUMBER 18

COMMUNICATIONS ABOUT THE PROGRAMMING LANGUAGE PASCAL BY PASCALERS

MAY, 1980

POLICY: PASCAL NEWS (17-Mar-80)

* Pascal News is the official but informal publication of the User's Group.

* Pascal News contains all we (the editors) know about Pascal; we use it as the vehicle to answer all inquiries because our physical energy and resources for answering individual requests are finite. As PUG grows, we unfortunately succumb to the reality of:

  1. Having to insist that people who need to know "about Pascal" join PUG and read Pascal News - that is why we spend time to produce it!

  2. Refusing to return phone calls or answer letters full of questions - we will pass the questions on to the readership of Pascal News. Please understand what the collective effect of individual inquiries has at the "concentrators" (our phones and mailboxes). We are trying honestly to say: "We cannot promise more that we can do."

* Pascal News is produced 3 or 4 times during an academic year; usually in September, November, February, and May.

* ALL THE NEWS THAT'S FIT, WE PRINT. Please send material (brevity is a virtue) for Pascal News single-spaced and camera-ready (use dark ribbon and 18.5 cm lines!)

* Remember: ALL LETTERS TO US WILL BE PRINTED UNLESS THEY CONTAIN A REQUEST TO THE CONTRARY.

* Pascal News is divided into flexible sections:

  POLICY - explains the way we do things (ALL-PURPOSE COUPON, etc.)

  EDITOR'S CONTRIBUTION - passes along the opinion and point of view of the editor together with changes in the mechanics of PUG operation, etc.

  HERE AND THERE WITH PASCAL - presents news from people, conference announcements and reports, new books and articles (including reviews), notices of Pascal in the news, history, membership rosters, etc.

  APPLICATIONS - presents and documents source programs written in Pascal for various algorithms, and software tools for a Pascal environment; news of significant applications programs. Also critiques regarding program/algorithm certification, performance, standards conformance, style, output convenience, and general design.

  ARTICLES - contains formal, submitted contributions (such as Pascal philosophy, use of Pascal as a teaching tool, use of Pascal at different computer installations, how to promote Pascal, etc.).

  OPEN FORUM FOR MEMBERS - contains short, informal correspondence among members which is of interest to the readership of Pascal News.

  IMPLEMENTATION NOTES - reports news of Pascal implementations: contacts for maintainers, implementors, distributors, and documentors of various implementations as well as where to send bug reports. Qualitative and quantitative descriptions and comparisons of various implementations are publicized. Sections contain information about Portable Pascals, Pascal Variants, Feature-Implementation Notes, and Machine-Dependent Implementations.

POLICY: PASCAL USERS GROUP (17-Mar-80)

Purpose: The Pascal User's Group (PUG) promotes the use of the programming language Pascal as well as the ideas behind Pascal through the vehicle of Pascal News. PUG is intentionally designed to be non political, and as such, it is not an "entity" which takes stands on issues or support causes or other efforts however well-intentioned. Informality is our guiding principle; there are no officers or meetings of PUG.

The increasing availability of Pascal makes it a viable alternative for software production and justifies its further use. We all strive to make using Pascal a respectable activity.

Membership: Anyone can join PUG, particularly the Pascal user, teacher, maintainer, implementor, distributor, or just plain fan. Memberships from libraries are also encouraged. See the ALL-PURPOSE COUPON for details.

Facts about Pascal, THE PROGRAMMING LANGUAGE:

Pascal is a small, practical, and general-purpose (but not all-purpose) programming language possessing algorithmic and data structures to aid systematic programming. Pascal was intended to be easy to learn and read by humans, and efficient to translate by computers.

Pascal has met these goals and is being used successfully for:

* teaching programming concepts
* developing reliable "production" software
* implementing software efficiently on today's machines
* writing portable software

Pascal implementations exist for more than 105 different computer systems, and this number increases every month. The "Implementation Notes" section of Pascal News describes how to obtain them.

The standard reference and tutorial manual for Pascal is:

Pascal - User Manual and Report (Second, study edition)
by Kathleen Jensen and Niklaus Wirth.
Springer-Verlag Publishers: New York, Heidelberg, Berlin
1978 (corrected printing), 167 pages, paperback, $7.90.

Introductory textbooks about Pascal are described in the "Here and There" section of Pascal News.

The programming language, Pascal, was named after the mathematician and religious fanatic Blaise Pascal (1623-1662). Pascal is not an acronym.

Remember, Pascal User's Group is each individual member's group. We currently have more than 3357 active members in more than 41 countries. this year Pascal News is averaging more than 120 pages per issue.

Policy

JOINING PASCAL USER'S GROUP?

- Membership is open to anyone: Particularly the Pascal user, teacher, maintainer, implementor, distributor, or just plain fan.
- Please enclose the proper prepayment (check payable to "Pascal User's Group"); we will not bill you.
- Please do not send us purchase orders; we cannot endure the paper work!
- When you join PUG any time within an academic year: July 1 to June 30, you will receive all issues of Pascal News for that year.
- We produce Pascal News as a means toward the end of promoting Pascal and communicating news of events surrounding Pascal to persons interested in Pascal. We are simply interested in the news ourselves and prefer to share it through Pascal News. We desire to minimize paperwork, because we have other work to do.

- American Region (North and South America): Send $6.00 per year to the address on the reverse side. International telephone: 1-404-252-2600.
- European Region (Europe, North Africa, Western and Central Asia): Join through PUG (UK). Send £4.00 per year to: Pascal Users Group, c/o Computer Studies Group, Mathematics Department, The University, Southampton SO9 5NH, United Kingdom; or pay by direct transfer into our Post Giro account (28 513 4000); International telephone: 44-703-559122 x700.
- Australasian Region (Australia, East Asia - incl. Japan): PUG(AUS). Send $A8.00 per year to: Pascal Users Group, c/o Arthur Sale, Department of Information Science, University of Tasmania, Box 252C GPO, Hobart, Tasmania 7001, Australia. International telephone: 61-02-23 0561 x435

PUG(USA) produces Pascal News and keeps all mailing addresses on a common list. Regional representatives collect memberships from their regions as a service, and they reprint and distribute Pascal News using a proof copy and mailing labels sent from PUG(USA). Persons in the Australasian and European Regions must join through their regional representatives. People in other places can join through PUG(USA).

RENEWING?

- Please renew early (before August) and please write us a line or two to tell us what you are doing with Pascal, and tell us what you think of PUG and Pascal News. Renewing for more than one year saves us time.

ORDERING BACK ISSUES OR EXTRA ISSUES?

- Our unusual policy of automatically sending all issues of Pascal News to anyone who joins within a academic year (July 1 to June 30) means that we eliminate many requests for backissues ahead of time, and we don't have to reprint important information in every issue--especially about Pascal implementations!
- Issues 1 .. 8 (January, 1974 - May 1977) are out of print.
  (A few copies of issue 8 remain at PUG(UK) available for £2 each.)
- Issues 9 .. 12 (September, 1977 - June, 1978) are available from PUG(USA) all for $10.00 and from PUG(AUS) all for $A10.
- Issues 13 .. 16 are available from PUG(UK) all for £6; from PUG(AUS) all for $A10; and from PUG(USA) all for $10.00.
- Extra single copies of new issues (current academic year) are: $3.00 each - PUG(USA); £2 each - PUG(UK); and $A3 each - PUG(AUS).

SENDING MATERIAL FOR PUBLICATION?

- Your experiences with Pascal (teaching and otherwise), ideas, letters, opinions, notices, news, articles, conference announcements, reports, implementation information, applications, etc. are welcome. Please send material single-spaced and in camera-ready (use a dark ribbon and lines 18.5 cm wide) form.
- All letters will be printed unless they contain a request to the contrary.

APPLICATION FOR LICENSE TO USE VALIDATION SUITE FOR PASCAL

Name and address of requestor:
(Company name if requestor is a company) _____

_____

_____

Phone Number: _____

Name and address to which information should be addressed (Write "as above" if the same) _____

_____

_____

Signature of requestor: _____

Date: _____

In making this application, which should be signed by a responsible person in the case of a company, the requestor agrees that:

a) The Validation Suite is recognized as being the copyrighted, proprietary property of R. A. Freak and A.H.J. Sale, and

b) The requestor will not distribute or otherwise make available machine-readable copies of the Validation Suite, modified or unmodified, to any third party without written permission of the copyright holders.

In return, the copyright holders grant full permission to use the programs and documentation contained in the Validation Suite for the purpose of compiler validation, acceptance tests, benchmarking, preparation of comparative reports, and similar purposes, and to make available the listings of the results of compilation and execution of the programs to third parties in the course of the above activities. In such documents, reference shall be made to the original copyright notice and its source.

✗ Distribution charge: $50.00

✗ Make checks payable to ANPA/RI in US dollars drawn on a US bank. Remittance must accompany application.

Mail request to:

ANPA/RI
P.O. Box 598
Easton, Pa. 18042
USA
Attn: R.J. Cichelli

Source Code Delivery Medium Specification:
9-track, 800 bpi, NRZI, Odd Parity, 600' Magnetic Tape

( ) ANSI-Standard

a) Select character code set:
( ) ASCII ( ) EBCDIC

b) Each logical record is an 80 character card image.
Select block size in logical records per block.
( ) 40 ( ) 20 ( ) 10

( ) Special DEC System Alternates:
( ) RSX-IAS PIP Format
( ) DOS-RSTS FLX Format

Office use only

Signed _____
Date _____

Richard J. Cichelli
On behalf of A.H.J. Sale & R.A. Freak

- - - - - - ALL-PURPOSE COUPON - - - - - - (17-Mar-80)

Pascal User's Group, c/o Rick Shaw
Digital Equipment Corporation
5775 Peachtree Dunwoody Road
Atlanta, Georgia 30342 USA

NOTE

- Membership is for an academic year (ending June 30th).
- Membership fee and All Purpose Coupon is sent to your Regional Representative.
- SEE THE POLICY SECTION ON THE REVERSE SIDE FOR PRICES AND ALTERNATE ADDRESS if you are located in the European or Australasian Regions.
- Membership and Renewal are the same price.
- The U. S. Postal Service does not forward Pascal News.

- -

[ ] Enter me as a new member for:
[ ] Renew my subscription for:

[ ] 1 year ending June 30, 1980
[ ] 2 years ending June 30, 1981
[ ] 3 years ending June 30, 1982

[ ] Send Back Issue(s) _____

[ ] My new/correct address/phone is listed below

[ ] Enclosed please find a contribution, idea, article or opinion which is submitted for publication in the Pascal News.

[ ] Comments: _____

_____

_____

ENCLOSED PLEASE FIND: $ / A$ / £ _____.____

NAME _____

ADDRESS _____

_____

_____

PHONE _____

COMPUTER _____

DATE _____

Contributors to this issue (#18) were:

EDITOR	Rick Shaw
Here & There	John Eisenberg
Books & Articles	Rich Stevens
Applications	Rich Cichelli, Andy Mickel
Standards	Jim Miner, Tony Addyman
Implementation Notes	Bob Dietrich
Administration	Moe Ford, Kathy Ford, Jennie Sinclair

Editor's Contribution

Wow! Bet ya didn't expect to see another edition of Pascal News so soon! Actually, PN #17 was so late that we printed both editions at the same time.

ABOUT THIS ISSUE

In an effort to keep our members up to date with activity on the standards front, we have devoted this whole issue to the proposed ISO draft standard.

It is very important that our members review this proposal and comment if they feel it necessary. The national standards body in your country, or a member of the standards committee is the best person to send any comments. (See also Tony Addyman's comments on returning comments.)

ON BEING ON TIME

As you have probably noticed, Pascal News is still not back on a proper schedule. So, what are we doing about it? Well I'll tell you. We are working very hard. Honest! The plan is to publish PN #19 in June. This would get us almost up to date for this year. Then work through the summer to get PN #20 out by September. This would make us almost on schedule, right? The reason that we can be so optimistic is that all the set up work for an operation in Atlanta (as opposed to Minneapolis) has been completed. Now all we have to do is crank out the news!

THE BAD NEWS

Inflation has hit PUG. As of 1-July-80 the membership fee for Pascal Users Group will have to be raised. It will not be much, but at least enough to cover the cost of printing and mailing. We are loosing money every issue now. In the U.S. at the moment it is only a few cents a copy. But at $1.43 a copy for returned issues by the Post Office we are getting killed. Note that you members can help with this problem, by always informing us of your new address when you move.

THANKS

We all owe Tony Addyman a debt of gratitude, for the years (literally!) of work that has gone into the proposal for an ISO standard for the language Pascal. Without his drive and enthusiasm, the standard for Pascal would still be just a good idea.

A Draft Propasal for Pascal

A.M.Addyman
Dept of Computer Science
University of Manchester
Oxford Road
Manchester, M13 9PL, United Kingdom

CONTENTS

0. FOREWORD TO THE DRAFT

The language Pascal was designed by Professor Niklaus Wirth to satisfy two principal aims:

(a) to make available a language suitable for teaching programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language.

(b) to define a language whose implementations could be both reliable and efficient on then available computers.

However, it has become apparent that Pascal has attributes which go far beyond these original goals. It is now being increasingly used commercially in the writing of both system and application software. This standard is primarily a consequence of the growing commercial interest in Pascal and the need to promote the portability of Pascal programs between data processing systems.

1. SCOPE OF THIS STANDARD

1.1 This Standard specifies requirements for
(a) the syntax of Pascal;
(b) the semantic rules for interpreting the meaning of a program written in Pascal;
(c) the form of input data to be processed by a program written in Pascal;
(d) the form of output data produced by a program written in Pascal.

1.2 This standard does not specify
(a) the size or complexity of a program and its data that will exceed the capacity of any specific data processing system or the capacity of a particular processor;
(b) the minimal requirements of a data processing system that is capable of supporting an implementation of a processor for Pascal;
(c) the set of commands used to control the environment in which a Pascal program is transformed and executed;
(d) the mechanism by which programs written in Pascal are transformed for use by a data processing system.

2. REFERENCES

None.

3. DEFINITIONS

(a) error. A violation by a program of the requirements of this standard.
(b) implementation-defined. Those parts of the language which may differ between processors, but which will be defined for any particular processor.
(c) implementation-dependent. Those parts of the language which may differ between processors, and for which there need not be a definition for a particular processor.
(d) processor. A compiler, interpreter, or other mechanism which accepts the program as input and either executes it, prepares it for execution, or both.
(e) scope. The text for which the declaration or definition of an identifier or label is valid.
(f) totally-undefined. If a variable is of a structured-type, the state of the variable when every component of the variable is totally-undefined. Totally-undefined is synonymous with undefined if the variable is not of a structured-type.

(a) accept all the features of the language specified in clause 6 with the meanings defined in clause 6;
(b) be accompanied by a document that provides a definition of all implementation-defined features;
(c) treat each occurrence of an error in at least one of the following ways:
1) there shall be a statement in an accompanying document that the error is not reported;
2) the processor shall have reported a prior warning that an occurrence of that error was possible;
3) the processor shall report the error during preparation of the program for execution;
4) the processor shall report the error during execution of the program.
The method for reporting errors or warnings shall be implementation-dependent.
(d) be accompanied by a document that separately describes any features accepted by the processor that are not specified in clause 6. Such extensions shall be described as being 'extensions to Pascal specified by ISO....: 198-'.
(e) be able to process in a manner similar to that specified for errors any use of any such extension;
(f) be able to process in a manner similar to that specified for errors any use of an implementation-dependent feature.

5.2 Programs
A program complying with the requirements of this standard shall:
(a) use only those features of the language specified in clause 6;
(b) not rely on any particular interpretation of implementation-dependent features.

6. REQUIREMENTS

6.1 Lexical tokens

NOTE. The syntax given in this sub-clause (6.1) describes the formation of lexical tokens from characters and the separation of these tokens, and therefore does not adhere to the same rules as the syntax in the rest of this standard.

6.1.1 General. The lexical tokens used to construct Pascal programs shall be classified into special-symbols, identifiers, directives, unsigned-numbers, labels and character-strings. The case of any letter occurring anywhere outside of a character-string (see 6.1.7) shall be insignificant in that occurrence to the meaning of the program.

```
letter = "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" .

digit = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9" .
```

6.1.2 Special-symbols. The special-symbols are tokens having special meanings and shall be used to delimit the syntactic units of the language.

```
special-symbol = "+"|"-"|"*"|"/"|"="|"<"|">"|"["|"]"|
                 "."|","|":"|";"|"^"|"("|")"|
                 "<>"|"<="|">="|":="|".." | word-symbol .

word-symbol =    "and"|"array"|"begin"|"case"|"const"|"div"|
                 "do"|"downto"|"else"|"end"|"file"|"for"|
                 "function"|"goto"|"if"|"in"|"label"|"mod"|
                 "nil"|"not"|"of"|"or"|"packed"|"procedure"|
                 "program"|"record"|"repeat"|"set"|"then"|
                 "to"|"type"|"until"|"var"|"while"|"with" .
```

6.1.3 Identifiers. Identifiers shall serve to denote constants, types, variables, procedures, functions parameters, bounds and programs, and fields and tag-fields in records. Identifiers may be of any length. No identifier shall have the same spelling as any word-symbol.

```
identifier = letter {(letter | digit)} .
```

Examples:

```
X          time       readinteger  sum    AlterHeatSetting
InquireWorkstationTransformation
InquireWorkstationIdentification
```

6.1.4 Directives. Directives shall only occur as a replacement for a procedure-block or function-block. The directive forward shall be the only standard directive (see 6.6.1 and 6.6.2). Other implementation-dependent directives may be defined. No directive shall have the same spelling as any word-symbol.

```
directive = letter {(letter | digit)} .
```

6.1.5 Numbers. Decimal notation shall be used for numbers that are the constants of integer-type and real-type (see 6.4.2.2). The letter "e" preceding a scale factor shall mean 'times ten to the power of'. The value of an unsigned-integer shall be in the closed interval 0 to maxint (see 6.4.2.2).

```
digit-sequence = digit {digit} .
unsigned-integer = digit-sequence .
unsigned-real =
    unsigned-integer "." digit-sequence ["e" scale-factor] |
    unsigned-integer "e" scale-factor .
unsigned-number = unsigned-integer | unsigned-real .
scale-factor = signed-integer .
sign = "+" | "-" .
signed-integer = [sign] unsigned-integer .
signed-real = [sign] unsigned-real .
signed-number = signed-integer | signed-real .
```

Examples:

```
1e10     1      +100      -0.1      5e-3      87.35E+8
```

6.1.6 Labels. Labels shall be digit-sequences and shall be distinguished by their apparent integral values, that shall be in the closed interval 0 to 9999.

... naturally reflected by the language.

```
label = digit-sequence .
```

6.1.7 Character-strings. A character-string consisting of a single string-element shall denote a constant of char-type (see 6.4.2.2). A character-string consisting of enclosed string-elements shall denote a constant of a string-type (see 6.4.3.2) with the same number of components as the character-string has string-elements. If the string of characters is to contain an apostrophe, this apostrophe shall be denoted by an apostrophe-image. Each string-character shall denote an implementation-defined value of char-type.

```
character-string = "'" string-element
    {string-element} "'" .
string-element = apostrophe-image | string-character .
apostrophe-image = "''" .
string-character =
    one-of-an-implementation-defined-set-of-characters .
```

Examples:

```
'A'      ';'      ''''
'Pascal'          'THIS IS A STRING'
```

6.1.8 Token separators. The construct

```
"{" any-sequence-of-characters-and-ends-of-lines-not-
containing-right-brace "}"
```

shall be a comment if the "{" does not occur within a character-string. The substitution of a space for a comment shall not alter the meaning of a program.

Comments, spaces (except in character-strings), and ends of lines shall be considered to be token separators. Zero or more token separators may occur between any two consecutive tokens, or before the first token of a program text. There shall be at least one separator between any pair of consecutive tokens made up of identifiers, word-symbols, or unsigned-numbers. No separators shall occur within tokens.

6.2 Blocks and scope

6.2.1 Block. A block shall consist of the definitions, declarations and statement-part that together form a part of a procedure-declaration, of a function-declaration or of a program.

```
block = label-declaration-part
            constant-definition-part
               type-definition-part
                  variable-declaration-part
                     procedure-and-function-declaration-part
                        statement-part .
```

The label-declaration-part shall specify all labels that prefix a statement in the corresponding statement-part. Each declared label shall prefix at most one statement in the statement-part. The

occurrence of a label as part of a label-declaration-part shall be its defining-point for the region that is the block immediately containing the label-declaration-part.

```
label-declaration-part = ["label" label {"," label} ";"] .

constant-definition-part = ["const" constant-definition ";"
                              {constant-definition ";"}] .

type-definition-part = ["type" type-definition ";"
                         {type-definition ";"}] .

variable-declaration-part = ["var" variable-declaration ";"
                              {variable-declaration ";"}] .

procedure-and-function-declaration-part =
      {(procedure-declaration | function-declaration) ";"} .
```

The statement-part shall specify the algorithmic actions to be executed upon an activation of the block.

```
statement-part = compound-statement .
```

All variables whose identifiers are declared in the variable-declaration-part of a block, except for those listed as program-parameters, shall be totally-undefined when execution of the statement-part of their block commences.

6.2.2 Scope

6.2.2.1 Each identifier or label within the block of a Pascal program shall have a defining-point.

6.2.2.2 Each defining-point shall have a region that is a part of the program text, and a scope that is a part or all of that region.

6.2.2.3 The region of each defining-point is defined elsewhere (see 6.2.1, 6.2.2.10, 6.3, 6.4.1, 6.4.2.3, 6.4.3.3, 6.5.1, 6.6.1, 6.6.2, 6.6.3.1, 6.8.3.10).

6.2.2.4 The scope of each defining-point shall be its region (including all regions enclosed by that region) subject to 6.2.2.5 and 6.2.2.6.

6.2.2.5 When an identifier or label that has a defining-point for region A has a further defining-point for some region B enclosed by A, then region B and all regions enclosed by B shall be excluded from the scope of the defining-point for region A.

6.2.2.6 The field-identifier of a field-designator (see 6.5.3.3) shall be one of the field-identifiers associated with the type of the record-variable.

6.2.2.7 The scope of a defining-point of an identifier or label shall include no other defining-point of the same identifier or label.

6.2.2.8 Within the scope of a defining-point of an identifier or label, all other occurrences of that identifier or label shall be designated corresponding occurrences. No occurrence outside that scope shall be a corresponding occurrence.

6.2.2.9 A defining-point of an identifier or label shall precede all corresponding occurrences of that identifier or label in the program-block with one exception, namely that a type-identifier T, that denotes the domain of a pointer-type ↑T, may have its defining-point anywhere within the type-definition-part in which ↑T occurs.

6.2.2.10 Identifiers that denote standard constants, types, procedures and functions shall be used as if their defining-points have a region enclosing the program.

6.2.2.11 Whatever an identifier or label denotes at its defining-point shall be denoted at all corresponding occurrences of that identifier or label.

6.3 Constant-definitions. A constant-definition shall introduce an identifier to denote a constant.

```
constant-definition = identifier "=" constant .
constant = [sign] (unsigned-number | constant-identifier)
         | character-string .
constant-identifier = identifier .
```

The occurrence of an identifier as the left-hand side of a constant-definition shall be its defining-point, at the end of the constant-definition, for the region that is the block immediately containing the constant-definition-part in which the constant-definition occurs. Each corresponding occurrence of that identifier shall be a constant-identifier and shall denote the constant of the constant-definition. A constant-identifier preceded by a sign shall have been defined to denote a value of real-type or of integer-type.

6.4 Type-definitions

6.4.1 General. A type shall be an attribute that is possessed by every value and every variable. Each occurrence of a new-type shall denote a distinct type. A type-definition shall introduce an identifier to denote a type.

```
type-definition = identifier "=" type-denoter .
type-denoter = type-identifier | new-type .
new-type = simple-type | structured-type | pointer-type .
```

The occurrence of an identifier as the left-hand side of a type-definition shall be its defining-point, at the end of the type-definition, for the region that is the block immediately containing the type-definition-part in which the type-definition occurs. Each corresponding occurrence of that identifier shall be a type-identifier and shall denote the same type as is denoted by its type-denoter.

Types shall be classified as simple, structured or pointer types according to the new-type with which they have been denoted. There shall be in addition certain predefined types which shall be denoted by predefined type-identifiers (see 6.4.2.2 and 6.4.3.5). A type-identifier shall be considered as a simple-type-identifier, a structured-type-identifier, or a pointer-type-identifier, according to the type that it denotes.

```
simple-type-identifier = type-identifier .
structured-type-identifier = type-identifier .
pointer-type-identifier = type-identifier .
type-identifier = identifier .
```

6.4.2 Simple-types

6.4.2.1 General. A simple-type shall determine an ordered set of values. The values of each ordinal-type shall have integer ordinal numbers.

```
simple-type = ordinal-type | real-type .
ordinal-type = enumerated-type | subrange-type |
               integer-type | Boolean-type | char-type |
               ordinal-type-identifier .
```

Where an appropriate word is substituted for x, an x-type-identifier shall be a type-identifier defined to denote an x-type.

6.4.2.2 Standard simple-types. The following types shall be standard:

integer-type — The predefined integer-type-identifier integer shall denote the integer-type. The values shall be a subset of the whole numbers, denoted as specified in 6.1.5 by the signed-integer values (see also 6.7.2.2). The ordinal number of a value of integer-type shall be the value itself.

real-type — The predefined real-type-identifier real shall denote the real-type. The values shall be an implementation-defined subset of the real numbers denoted as specified in 6.1.5 by the signed-real values.

Boolean-type — The predefined Boolean-type-identifier Boolean shall denote the Boolean-type. The values shall be the enumeration of truth values denoted by the predefined constant-identifiers false and true, such that false is the predecessor of true. The ordinal numbers of the truth values denoted by false and true shall be the integer values 0 and 1 respectively.

char-type — The predefined char-type-identifier char shall denote the char-type. The type shall be the enumeration of a set of implementation-defined characters, some possibly without graphic representations. The ordinal numbers of the character values shall be values of integer-type, that are implementation-defined, and that are determined by mapping the character values on to consecutive non-negative integer values starting at zero. The mapping shall be order preserving. The following relations shall hold:

(a) The subset of character values representing the digits 0 to 9 shall be numerically ordered and

contiguous.

(b) The subset of character values representing the upper-case letters A to Z, if available, shall be alphabetically ordered but not necessarily contiguous.

(c) The subset of character values representing the lower-case letters a to z, if available, shall be alphabetically ordered but not necessarily contiguous.

(d) The ordering relationship between any two character values shall be the same as between their ordinal numbers.

NOTE. Operators applicable to standard types are specified in 6.7.2.

6.4.2.3 Enumerated-types. An enumerated-type shall determine an ordered set of values by enumeration of the identifiers that denote those values. The ordering of these values shall be determined by the sequence in which their identifiers are enumerated, i.e. if x precedes y then x is less than y. The ordinal number of a value that is of an enumerated-type shall be determined by mapping all the values of the type as they occur in the identifier-list of the enumerated-type on to consecutive non-negative integer values starting from zero.

```
enumerated-type = "(" identifier-list ")" .
identifier-list = identifier { "," identifier } .
```

The occurrence of an identifier as part of the identifier-list of an enumerated-type shall be its defining-point as a constant-identifier for the region that is the block immediately containing the type-definition-part or variable-declaration-part in which the enumerated-type occurs.

Examples:

```
(red,yellow,green,blue,tartan)
(club,diamond,heart,spade)
(married,divorced,widowed,single)
(scanning,found,notpresent)
(Busy,InterruptEnable,ParityError,OutOfPaper,LineBreak)
```

6.4.2.4 Subrange-types. The definition of a type as a subrange of an ordinal-type shall include identification of the smallest and the largest value in the subrange. The first constant shall specify the smallest value which shall be less than or equal to the largest value. Both constants shall be of the same ordinal-type, and that ordinal-type shall be designated the host type of the subrange-type.

```
subrange-type = constant ".." constant .
```

Examples:

```
1..100
-10..+10
red..green
'0'..'9'
```

6.4.3 Structured-types

6.4.3.1 General. Structured-types shall be classified as array, record, set or file types according to the unpacked-structured-type immediately contained in their denotation. A component of a value of a structured-type shall be a value.

```
structured-type = ["packed"] unpacked-structured-type |
                  structured-type-identifier .
unpacked-structured-type = array-type | record-type | set-type |
                           file-type .
```

A structured-type which immediately contains an unpacked-structured-type shall be designated packed if and only if the token packed is immediately contained in the structured-type. The designation of a structured-type as packed shall indicate to the processor that data-storage should be economised, even if this causes operations on, or accesses to components of, variables of the type to be less efficient in terms of space or time.

The designation of a structured-type as packed shall affect only the representation in data-storage of that structured-type. If a component is itself structured, the component's representation in data-storage shall be packed only if the type of the component is designated packed.

> NOTE. Sections 6.4.3.2, 6.4.5, 6.6.3.3, and 6.6.5.4 specify the ways in which the treatment of entities of a type is affected by whether or not the type is designated packed.

6.4.3.2 Array-types. An array-type shall be structured as a mapping from each value of its index-type onto a distinct component. The index-type shall be an ordinal-type.

```
array-type = "array" "[" index-type { "," index-type } "]" "of"
             component-type .
index-type = ordinal-type .
component-type = type-denoter .
```

Examples:

```
array [1..100] of real
array [Boolean] of colour
```

An array-type that specifies a sequence of two or more index-types shall be an alternative notation for an array-type specified to have the index-type of the first index-type in the sequence, and to have a component-type that is an array-type specifying the sequence of index-types without the first and specifying the same component-type as the original specification. The component-type thus constructed

shall be designated packed if and only if the original array-type is designated packed.

NOTE. Each of the following two examples thus contains different ways of expressing its array-type.

```
Example 1.
  array[Boolean] of array[1..10] of array[size] of real
  array[Boolean] of array[1..10,size] of real
  array[Boolean,1..10,size] of real
  array[Boolean,1..10] of array[size] of real
Example 2.
  packed array[1..10,1..8] of Boolean
  packed array[1..10] of packed array[1..8] of Boolean
```

Let i denote a value of the index-type; let v[i] denote a value of that component of the array-type that corresponds to the value i by the structure of the array-type; let the smallest and largest values of the index-type be denoted by m and n; and let k = (ord(n)-ord(m)+1) denote the number of values of the index-type. Then the values of the array-type shall be the distinct k-tuples of the form:

```
  (v[m], ... ,v[n])
```

NOTE. A value of an array-type does not therefore exist unless all of its component values are defined. If the component-type has c values, then it follows that the cardinality of the set of values of the array-type is c raised to the power k.

Any type denoted by

```
  packed array[T1] of T2
```

where T1 is a subrange-type with a lower bound of 1 and T2 is the char-type, shall be designated a string-type.

NOTE. The values of a string-type possess additional properties which determine their correspondence with character-strings (see 6.1.7), allow writing them to textfiles (see 6.9.4.7) and define their use with relational-operations (see 6.7.2.5).

6.4.3.3 Record-types. A record-type shall be structured as a fixed number of components that shall be designated fields.

The occurrence of an identifier as a tag-field or as part of the identifier-list of a record-section shall be its defining-point as a field-identifier for the region that is the record-type immediately containing the tag-field or record-section. Each field-identifier shall be associated with a component of the specified type.

Let a variant-part contained in a field-list be considered as an additional field with appropriate values, and let Vi denote a value of the i-th field in a record-type definition with m fields. Then the record-type shall have a single null value if it has no fields; otherwise it shall have only the set of values:

V1, ... ,Vm

NOTE. If the number of values in each of the fields is F1,F2,...,Fm; then it follows that the cardinality of the set of values of the record-type is (F1*F2*...*Fm).

If the record-type contains a variant-part, the tag-type of that variant-part shall be an ordinal-type. All the case-constants of that variant-part shall be distinct and shall be of a type compatible with the tag-type (see 6.4.5). The set of values of all the case-constants shall be equal to the set of values of the tag-type.

Let each field-list immediately contained in a variant of a variant-part be considered to be a record-type with values as defined above. Then, if the variant-part contains a tag-field in its variant-selector or if its variants immediately contain no case-constant-lists with more than one case-constant, the variant-part shall have only the values:

k,Xk

where k denotes a value in the tag-type and Xk denotes a value of the variant associated with k. The occurrence of a case-constant in the case-constant-list of a variant shall associate the value of the case-constant with that variant.

NOTE. If there are n values in the tag-type, and the variant associated with the value i has Ti values, then it follows that the cardinality of the set of values of the variant-part is (T1+T2+...+Tn).

If a variant-part contains no tag-field in its variant-selector and it immediately contains case-constant-lists with more than one case-constant, then its values shall be determined as follows. Let f(i) denote an implicit function mapping values of the tag-type onto a new ordinal-type that shall have as many values as there are variants in the variant-part, and let the mapping be determined by associating with each variant in turn one value of this new type that is the result of applying f to each of the values of the tag-type in the case-constant-list associated with that variant. Then this case shall be equivalent to the one given before with the substitution of this new type for the tag-type and appropriate substitution of the case-constant-lists.

NOTE. A record-value exists only when none of its fields are undefined. A value of a variant-part exists when one and only one of its variants has a value.

The value of a tag-field shall determine which variant is active in determining the value of a variant-part. It shall be an error if any field-identifier defined within a variant is used in a field-designator (see 6.5.3.3) unless the value of the tag-field is associated with that variant. A variant-part that does not contain a tag-field in its variant-selector shall be assumed to have a virtual tag-field of the constructed ordinal-type described above and a