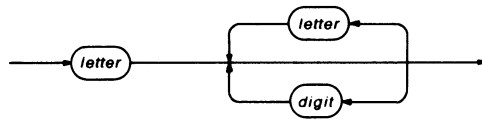
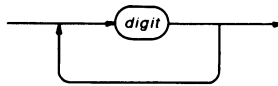


Appendix I Pascal Syntax Diagrams

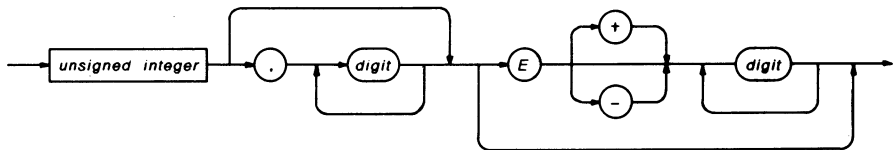
identifier



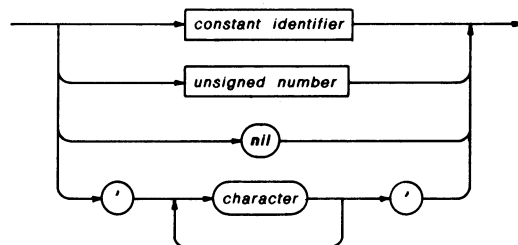
unsigned integer



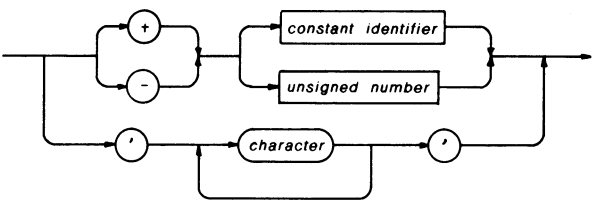
unsigned number



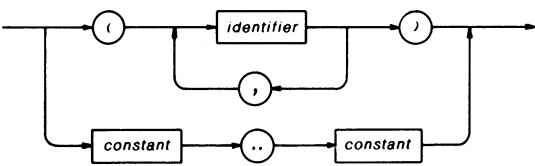
unsigned constant



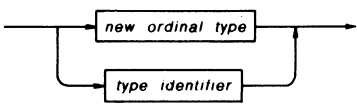
constant



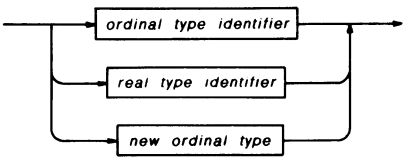
new ordinal type



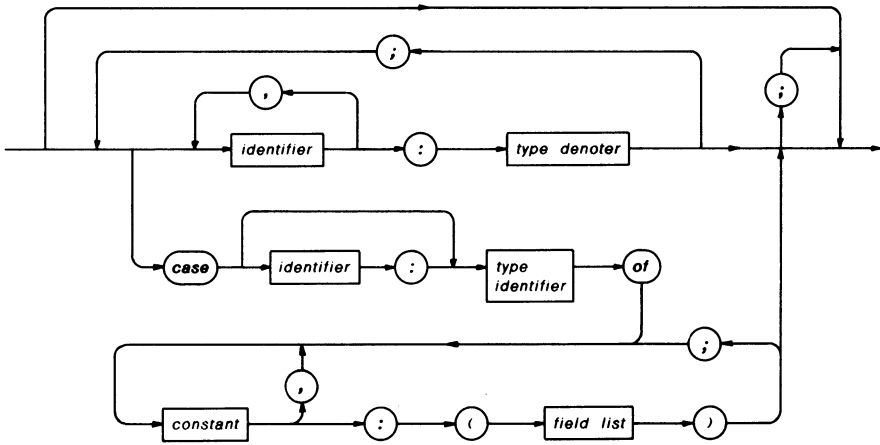
ordinal type



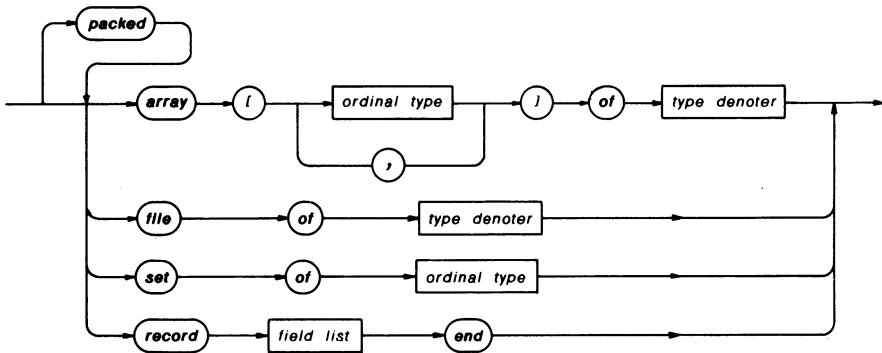
simple type



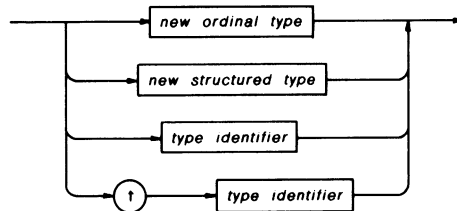
field list



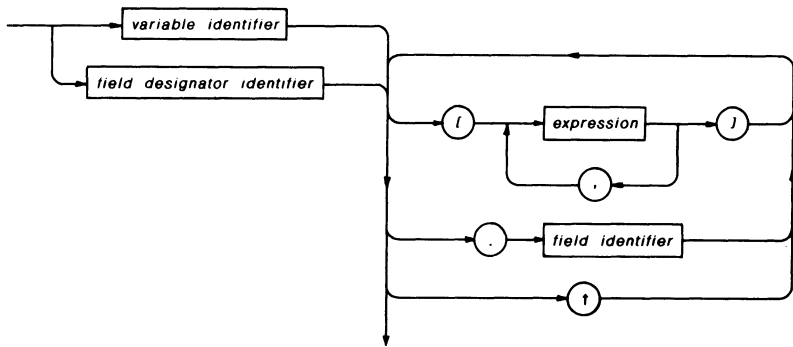
new structured type



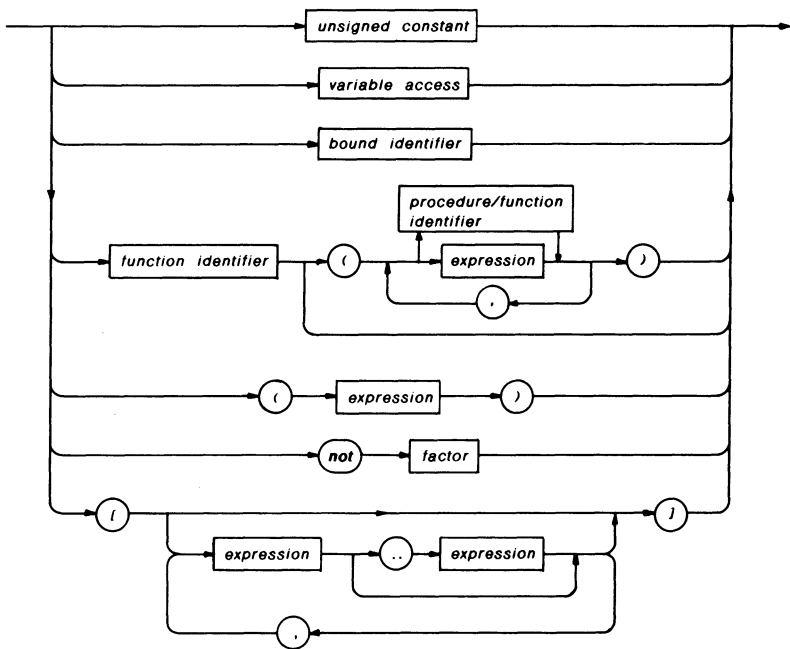
type denoter



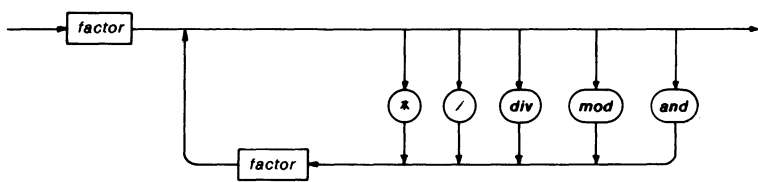
variable access



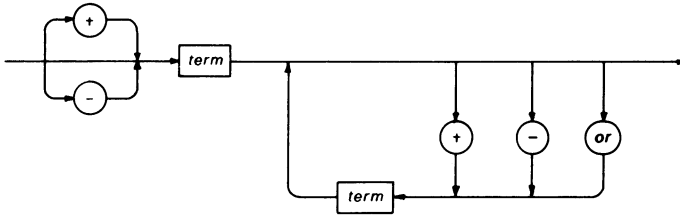
factor



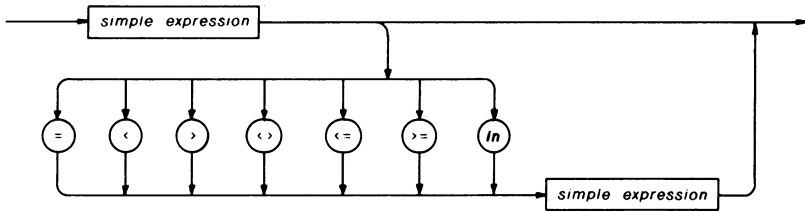
term



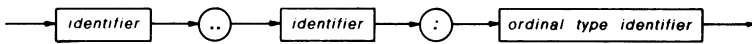
simple expression



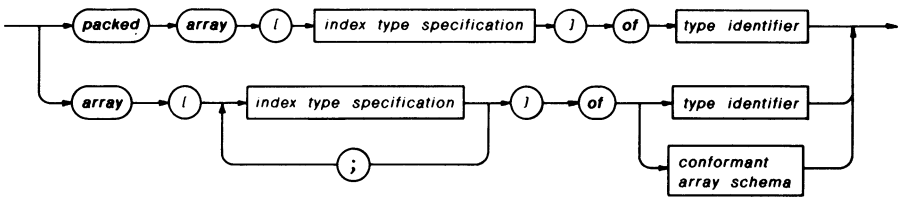
expression



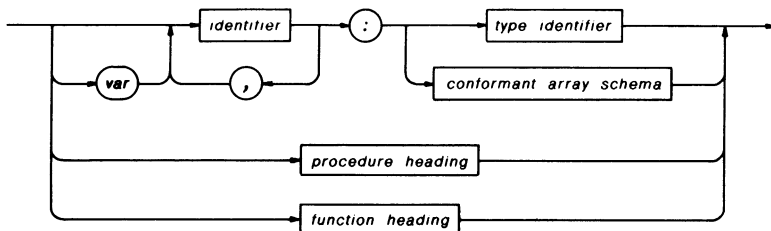
index type specification

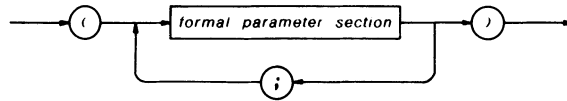
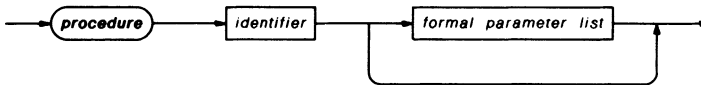
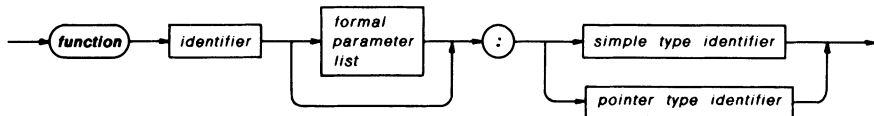
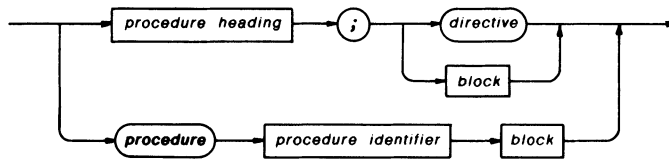
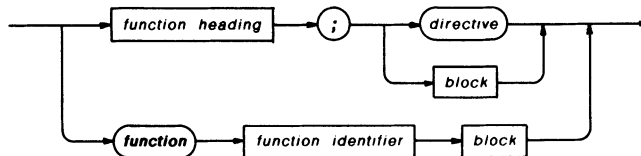


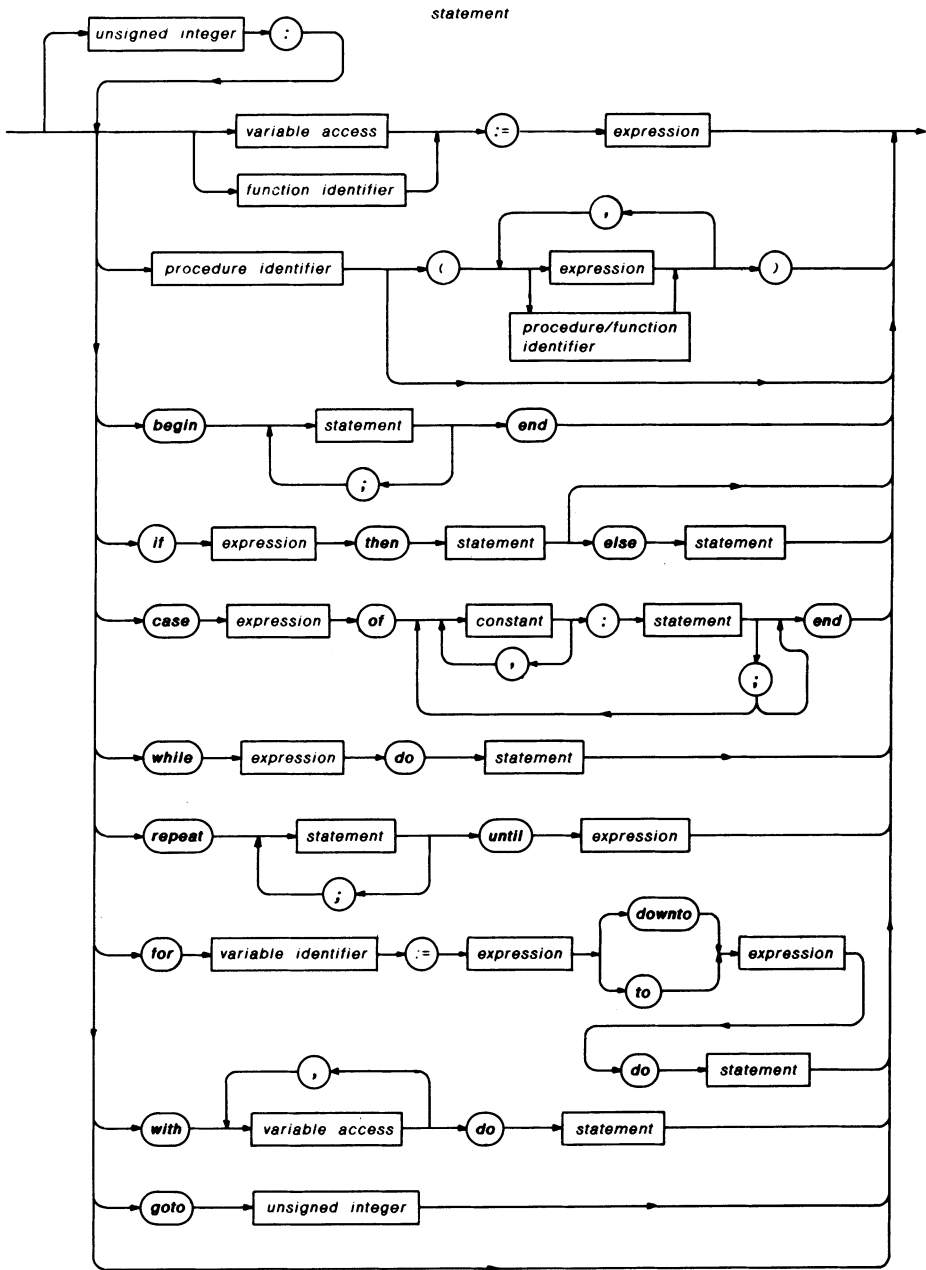
conformant array schema



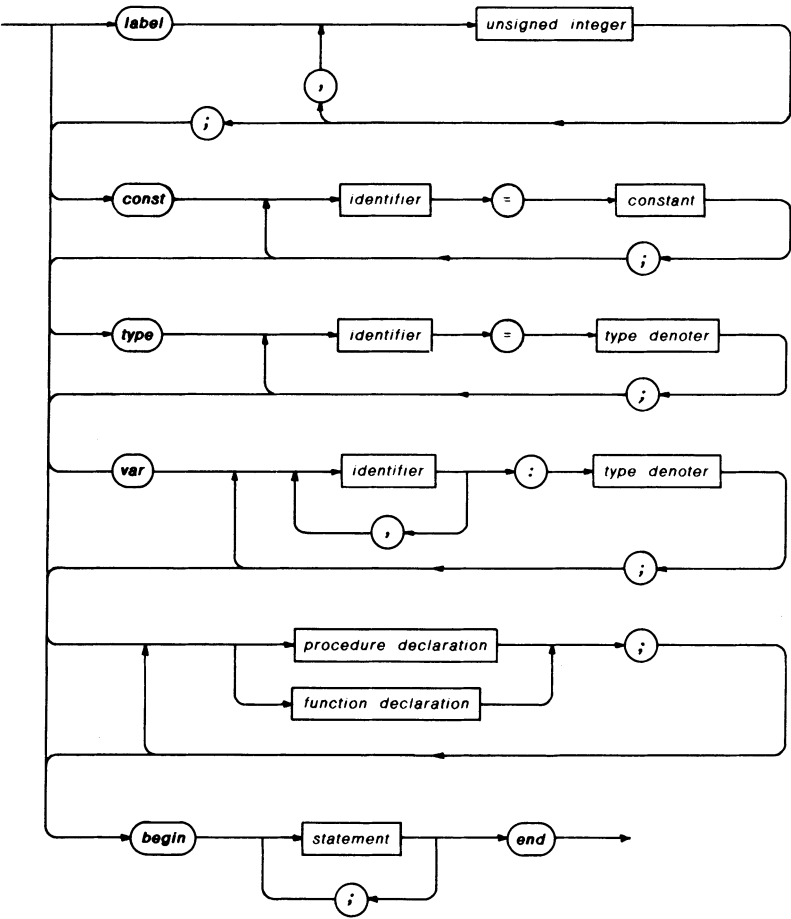
formal parameter section



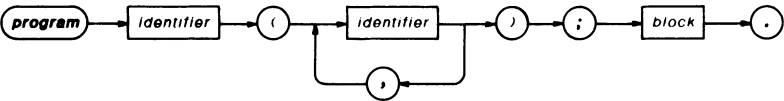
formal parameter list*procedure heading**function heading**procedure declaration**function declaration*



block



program



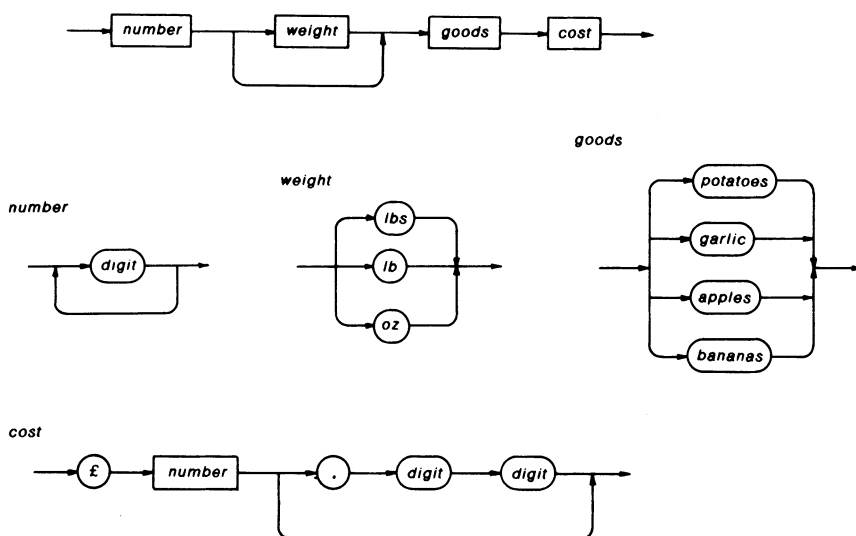
Appendix II List of Pascal Reserved Words

<i>and</i>	<i>nil</i>
<i>array</i>	<i>not</i>
<i>begin</i>	<i>of</i>
<i>case</i>	<i>or</i>
<i>const</i>	<i>packed</i>
<i>div</i>	<i>procedure</i>
<i>do</i>	<i>program</i>
<i>downto</i>	<i>record</i>
<i>else</i>	<i>repeat</i>
<i>end</i>	<i>set</i>
<i>file</i>	<i>then</i>
<i>for</i>	<i>to</i>
<i>function</i>	<i>type</i>
<i>goto</i>	<i>until</i>
<i>if</i>	<i>var</i>
<i>in</i>	<i>while</i>
<i>label</i>	<i>with</i>
<i>mod</i>	

Appendix III Answers to Exercises and Schema for Selected Problems

Chapter 1

shopping list item



Chapter 2

2.1

valid *x, x1, x1x, temperature, antidisestablishmentarianism*
 invalid *1x*—starts with digit
 John Brown—space
 re-activate— - ?

x(3)—brackets
begin, case—both are delimiters

2.2

line 1 ; missing
 line 2 *j* should be declared **real** as 3.4 is real.

	<i>k</i> should be declared
line 6	; missing
line 7	left side of assignment must be a variable
line 9	period missing

2.3 Lay out neatly—use meaningful identifiers—use comments—use *constants*—make output readable.

```
{ Program to print the sum of two integers }
program EX2A(input, output);
var number1, number2, sum : integer;
begin
  read(number1, number2);
  sum := number1 + number2;
  writeln('The total is ', sum)
end.
```

Chapter 3

3.1(a)

```
if pence = 0 then writeln('zero')
else writeln('non zero')
```

(b)

```
if i <= 200 then t := 0
else if i <= 400 then t := (i-200) * 0.3
  else if i <= 5000 then t := 200*0.3 + (i-400)*0.4
    else t := 200*0.3 + (i-400)*0.5
```

3.2

```
max := 0;
repeat
  read(num);
  if num > max then max := num
until num = 0;
writeln('Biggest is ', max)
```

3.3

```
for i := 1 to 200 do
  begin
    read(animal);
    case animal of
      1: writeln('lion');
      2: writeln('tiger');
      3: writeln('fly');
      .
      .
    end { of case }
  end
```

Problem 3.2

```

{ initialise count to 3 }
for m := 5 to 500 do
  begin
    { determine if m is prime
      by dividing by all numbers, n
      starting at m - 1 and until
      m divides exactly by n }
    if n = 1 then { increment count }
  end;
{ print answer }

```

Chapter 4

- 4.1(a) $a * \text{sqr}(x) + b * x + c$
 (b) $0.5 * (x * y + z / (3 * x))$
 (c) $\text{sqr}(\text{sqr}(a) + \text{sqr}(b))$

- 4.2(a) $1.0\text{E}-7$
 (b) $6.55260034\text{E}4$
 (c) $-7.5\text{E}-1$

- 4.3(a) 0.1666667 (b) 1.25 (c) 5 (d) 3 (e) 5

- 4.4(a) $i \bmod j = 0$
 (b) *not* (*odd*(*m*))
 (c) (*abs*(*y*) < 1) *or* (*abs*(*y*) > 2)

- 4.5 p *and* ($x \geq y$) *or* ($i < j$)

Chapter 5

5.1

```

for i := 1 to 20 do
  begin
    while not eoln do
      begin
        read(ch);
        if (ch = 'A') or (ch = 'E') or (ch = 'I')
          or (ch = 'O') or (ch = 'U') then write(ch)
        end;
      readln
    end
  end

```

5.2

EMPL NO.	INCOME	EXPENSES	DEPENDENTS	EMPLTYPE	TAX
I : 8	R:9:2	R:9:2	' ':10	I:1 ' ':8	I:1 R:8:2

where I = INTEGER and R = REAL

Chapter 6

- 6.1 line 4 — *procedure DrawALine(ch : char);*
 line 9 — *write(ch);*
 after line 14 — *DrawALine ('+');*
 after lines 16, 19 — *DrawALine ('-');*
- 6.2(a) line 2 — *function total : integer;*
 line 10 — *total := sum*
 indent body and change . to ;
- (b) line 2 — *function MaxPower(a,b:real):integer;*
 line 4 — delete *a, b,*
 line 7 — delete
 line 14 — *MaxPower := power*
 indent body and change . to ;
- (c) line 4 — *function av(n : integer) : real*
 line 4 — delete *,n*
 line 7 — delete *read(n);*
 line 14 — *av := average*
 indent body and change . to ;

6.3

```
{ Lengths arithmetic }
program EX6E(input, output);
```

```
var miles, yards, feet, inches, NumberOfInches : integer;
```

```
procedure convert
```

```
(var m, y, f, i : integer; ins : integer);
```

```
const
```

```
YardsInAMile = 1760;
```

```
InchesInAYard = 36;
```

```
InchesInAFoot = 12;
```

```
begin
```

```
m := ins div (YardsInAMile * InchesInAYard);
```

```
ins := ins mod (YardsInAMile * InchesInAYard);
```

```
y := ins div InchesInAYard;
```

```
ins := ins mod InchesInAYard;
```

```
f := ins div InchesInAFoot;
```

```
i := ins mod InchesInAFoot
```

```
end {convert};
```

```
function LengthInInches
```

```
(m, y, f, i : integer)
```

```
: integer;
```

```
const
```

```
YardsInAMile = 1760;
```

```
FeetInAYard = 3;
```

```
InchesInAFoot = 12;
```

```
begin
```

```
LengthInInches :=
```

```
((m*YardsInAMile) + y)*FeetInAYard + f)*InchesInAFoot + i
```

```
end {LengthInInches};
```

```
begin
```

```
read(miles, yards, feet, inches);
```

```
NumberOfInches := LengthInInches(miles, yards, feet, inches);
```

```
read(miles, yards, feet, inches);
```

```
NumberOfInches :=
```

```
NumberOfInches + LengthInInches(miles, yards, feet, inches);
```

```
convert(miles, yards, feet, inches, NumberOfInches);
```

```
writeln('Sum is ', NumberOfInches, ' inches - i.e.');
```

```
writeln(miles:4, ' miles,', yards:4, ' yards,',
```

```
feet:1, ' feet and ', inches:2, ' inches')
```

```
end.
```

Chapter 7

7.1 Implementation defined.

7.2 Implementation defined.

7.3 In the following examples the specified variables could have been declared as indicated.

- 3B power : 0..maxint;
- 3E a, b : 0..maxint;
- 3G CountOfPosNumbers, CountOfNegNumbers :
 0..maxint;
- 3H SumOfPosNumbers, ZeroCount : 0..maxint;
 SumOfNegNumbers : -maxint..0;
- 3I DayNumber : 1..7;
- 3J EmployeeType : 1..3;
 NumberOfEmployees, NumberOfDependents :
 0..maxint;
 i : 1..maxint;
- 5A NumberOfLines : 0..maxint;
- 5B TempInCentigrade : 0..99
- 6A i : 1..length;
- 6B NumberOfNumbers : 0..maxint;
 i : 1..maxint;
- 6C NumberOfPairs : 0..maxint;
 i : 1..maxint;
- 6D type natural = 0..maxint;
 InchRange = 0..11;
 FootRange = 0..2;
 YardRange = 0..1759;
 miles, ins and NumberOfInches are Natural
 yards and y are YardRange
 feet and f are FootRange
 inches and i are InchRange

- 6E similar to 6D
- 6F type Posint = 1..maxint;
 i (in DrawALine) : 1..width;
 i (in average) : Posint;
 ReadCount : Posint;
 i (in program) : Posint;
 count : Posint;
 n : 0..maxint;

Chapter 8

8.1

```
program LineCount(f, output);  
var  
    f : text;  
    LineCount : 0..maxint;  
begin  
    reset(f);  
    LineCount := 0;  
    while not eof(f) do  
        begin  
            readln(f);  
            LineCount := LineCount + 1  
        end;  
    writeln(LineCount)  
end.
```

8.2

```

program LineStats(f, output);
var
    f : text;
    minimum, maximum, count, total, LineCount : 0..maxint;
begin
    reset(f);
    minimum := maxint; maximum := 0; total := 0; LineCount := 0;
    while not eof(f) do
        begin
            count := 0;
            while not eoln(f) do
                begin
                    get(f);
                    count := count + 1
                end;
            readln(f);
            if count < minimum then minimum := count;
            if count > maximum then maximum := count;
            total := total + count;
            LineCount := LineCount + 1
        end;
    if LineCount <> 0 then writeln(minimum, maximum, total / LineCount)
    else writeln('empty file')
end.

```

8.3

```

program merge(f, g, out, output);
var
    ContinueMerge : boolean;
    f, g, out : file of integer;
begin
    reset(f); reset(g); rewrite(out);
    ContinueMerge := not(eof(f) or eof(g));
    while ContinueMerge do
        begin
            if ft > gt then
                begin
                    outt := gt; get(g);
                    ContinueMerge := not eof(g)
                end
            else
                begin
                    outt := ft; get(f);
                    ContinueMerge := not eof(f)
                end;
            put(out)
        end;
    while not eof(f) do
        begin
            outt := ft; put(out); get(f)
        end;
    while not eof(g) do
        begin
            outt := gt; put(out); get(g)
        end
    end.

```

8.4

```

repeat
  read(ch)
until (ch >= 'A') and (ch <= 'Z');

```

8.5

```

while (inputt < 'A') or (inputt > 'Z') do get(inputt)

```

8.6

```

procedure SkipBlanks(var f : text);
var state : (KeepGoing, EofDetected, NotASpace);
begin
  state := KeepGoing;
  repeat
    if eof(f) then state := EofDetected
    else if ft <> ' ' then state := NotASpace
    else get(f)
  until state <> KeepGoing
end; {SkipBlanks}

```

Problem A suitable scheme is

```

program currency(input, output, coins);
type natural = 0..maxint;
var
  coins : file of real;
  CoinValues : file of natural;
  smallest, pounds, value, ConversionRate : real;
  money, number, IntValue : natural;
begin
  { convert coins to multiples of least coin }
  { find the smallest coin - it is at the end }
  { create a file of integer multiples of the
    smallest coin }
  { read the conversion rate }
  { process the values to be converted }
  while not eof do
    begin
      { read the amount to be converted }
      { calculate the new amount as a multiple of the
        smallest denomination }
      { print the new amount correctly }
      { scan the coinvalues file for the necessary coins }
      while not eof(coinvalues) do
        begin
          { read a coin value }
          { calculate the number required }
          if number <> 0 then
            begin
              { print the number of coins and their correct value }
              { calculate the amount remaining }
            end
          end
        end
      end
    end
  end.

```

Chapter 9

9.1(a)

```

{ },
{ red }, { green }, { blue },
{ red, green }, { red, blue }, { green, blue },
{ red, green, blue }

```

(b)

```

{ },
{ 1 }, { 2 }, { 3 }, { 4 },
{ 1, 2 }, { 1, 3 }, { 1, 4 }, { 2, 3 }, { 2, 4 }, { 3, 4 },
{ 1, 2, 3 }, { 1, 2, 4 }, { 1, 3, 4 }, { 2, 3, 4 },
{ 1, 2, 3, 4 }

```

9.2(a) [3]

(b) [1 .. 5]

(c) [1, 2, 4]

9.3(a) *true* (b) *true* (c) *false* (d) *false*9.4(a) $[a, b, c] = [d]$ (b) $d \text{ in } [a, b, c]$ (c) $[a, b] = [c, d]$

Chapter 10

10.1

```

program FreqCount(f, output);
type frequencies = array['a'..'z'] of 0..maxint;
var
  ch : char;
  f : text;
  count : frequencies;
begin
  for ch := 'a' to 'z' do count[ch] := 0;
  reset(f);
  while not eof(f) do
    begin
      read(f, ch);
      if ch in ['a'..'z'] then
        count[ch] := count[ch] + 1
      end
    end
  end.

```

10.2 Assuming that most combinations will occur, a suitable type would be

```
type AllTwos = array[char, char] of boolean;
```

Problem 10.2 The correct data structure is

```
type PairFrequencies = array['a'..'z', 'a'..'z'] of 0..maxint;
```

A suitable scheme for the frequency calculation is

```
{ initialise the counts to zero }
reset(f);
if not eof(f) then { read the first character }
  while not eof(f) do
    begin
      { read a second character }
      if { both characters are letters } then
        { increment the frequency count }
        { make the second character the first one }
      end
    end.
```

Chapter 11

11.1

```
function difference(d1, d2 : dates) : integer;
function DaysFrom1900(d : dates) : integer;
var
  years : 0..100;
  days : 0..maxint;
  m : months;
  DayInYear : array[month] of 0..365;
begin { days from 1900 }
  DayInYear[jan] := 0; DayInYear[feb] := 31; DayInYear[mar] := 59;
  DayInYear[april] := 90; DayInYear[may] := 120; DayInYear[jun] := 151;
  DayInYear[july] := 181; DayInYear[aug] := 212; DayInYear[sep] := 243;
  DayInYear[oct] := 273; DayInYear[nov] := 304; DayInYear[dec] := 334;
  with d do
    begin
      years := y - 1900;
      days := years * 365;
      if m <= feb then years := years - 1;
      days := days + years div 4;
      days := days + DayInYear[m];
      DaysFrom1900 := days + d
    end
  end { DaysFrom1900 };
begin { difference }
  difference := DaysFrom1900(d2) - DaysFrom1900(d1)
end { difference };
```


11.2 Two changes are needed to the program. The first deals with the players' names. A simple modification to the loop which reads a player's name will allow the actual length of the name to be calculated. This information can be used to control the output when the name is printed. The second modification produces the longer description of the card. This requires the use of mapping functions or *case* statements, one for card suits and one for card faces. *Case* statements are preferable in both instances.

Problem A suitable data structure is:

```

type
  MorseTable =
    array(char) of
      record
        valid : boolean;
        length : 1..6;
        code : packed array[1..6] of char
      end;

```

This is filled with the information from the file to provide a mapping from characters to their Morse code equivalents.

Chapter 12

12.1

```

function perimeter(figure : GeometricFigure) : real;
const pi = 3.141596536;
begin
  with figure do
    case kind of
      rectangle :
        perimeter := 2 * (height + width);
      circle :
        perimeter := 2 * pi * radius;
      triangle :
        perimeter := side1 + side2 + sqrt(sqrt(side1) + sqrt(side2)
          - 2 * side1 * side2 * cos(angle))
    end
  end; { perimeter }

```

12.2(a)

```

program ANS122A(girls, output);
const n = 10;
type
  names = packed array[1..n] of char;
  FullName =
    record
      surname : names;
      NoOfNames : 1..3;
      forenames : array[1..3] of names
    end;
  FemaleStudent =
    record
      name : FullName;
      size : array[1..3] of 400..1299
    end;
var
  girls : file of FemaleStudent;
  AverageSize : array[1..3] of 0..maxint;
  i : 1..3;
  count : 0..maxint;
  person : FemaleStudent;
begin
  reset(girls); count := 0;
  for i := 1 to 3 do AverageSize[i] := 0;
  while not eof(girls) do
    begin
      read(girls, person);
      with person do
        for i := 1 to 3 do
          AverageSize[i] := AverageSize[i] + size[i];
        count := count + 1
      end;
  if count = 0 then write('empty file')
  else
    for i := 1 to 3 do write(AverageSize[i] div count);
  writeln
end.

```

(b)

```

program ANS122B(men, output);
const n = 10;
type
  names = packed array[1..n] of char;
  FullName =
    record
      surname : names;
      NoOfFNames : 1..3;
      forenames : array[1..3] of names
    end;
  MaleStudent =
    record
      name : FullName;
      money : real;
      car : boolean
    end;
var
  men : file of MaleStudent;
  NumberOfCarOwners : 0..maxint;
  person : MaleStudent;
  total : real;
  count : 0..maxint;
begin
  NumberOfCarOwners := 0; total := 0; count := 0;
  reset(men);
  while not eof(men) do
    begin
      read(men, person);
      with person do
        begin
          if car then
            NumberOfCarOwners := NumberOfCarOwners + 1;
            total := total + money;
            count := count + 1
          end
        end;
      if count = 0 then writeln('empty file')
      else writeln(NumberOfCarOwners, total / count)
    end.

```

(c) The better definition is

```

type
  FemaleStudent =
    record
      name : FullName;
      size :
        record
          bust, waist, hips : 400..1299
        end
    end;

```

12.3

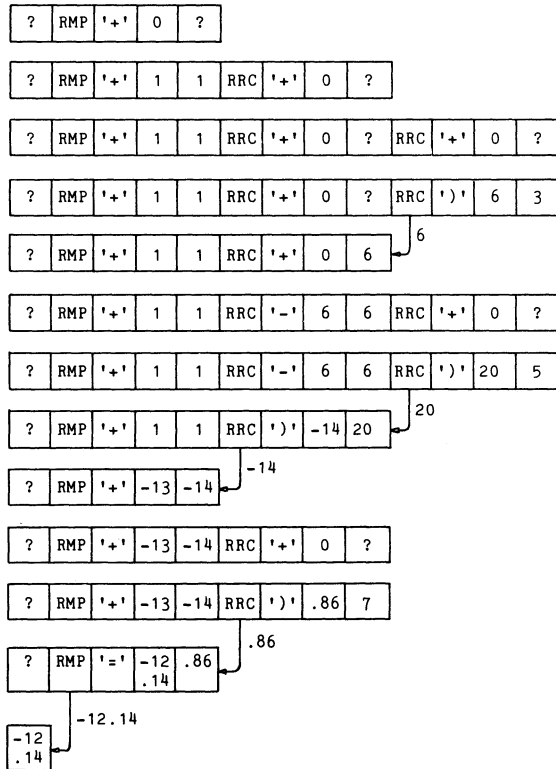
```

program ANS123(CSStudents, output);
const n = 10;
type
  names = packed array[1..n] of char;
  FullName =
    record
      surname : names;
      NoOfNames : 1..3;
      forenames : array[1..3] of names
    end;
  sexes = (female, male);
  student =
    record
      name : FullName;
      case sex : sexes of
        male : (money : real; car : boolean);
        female : (size : array[1..3] of 400..1299)
    end;
var
  MaleCount, FemaleCount, NumberOfCarOwners : 0..maxint;
  i : 1..3;
  total : real;
  AverageSize : array[1..3] of 0..maxint;
  person : student;
  CSStudents : file of student;
begin
  reset(CSStudents);
  MaleCount := 0; FemaleCount := 0; total := 0;
  NumberOfCarOwners := 0;
  for i := 1 to 3 do AverageSize[i] := 0;
  while not eof(CSStudents) do
    begin
      read(CSStudents, person);
      with person do
        case sex of
          male :
            begin
              if car then NumberOfCarOwners := NumberOfCarOwners + 1;
              total := total + money;
              MaleCount := MaleCount + 1
            end;
          female :
            begin
              for i := 1 to 3 do
                AverageSize[i] := AverageSize[i] + size[i];
              FemaleCount := FemaleCount + 1
            end
        end
      end;
    end;
  if MaleCount = 0 then writeln('no males')
  else writeln(NumberOfCarOwners, total / MaleCount);
  if FemaleCount = 0 then write('no females')
  else
    for i := 1 to 3 do
      write(AverageSize[i] div FemaleCount);
  writeln
end.

```

Chapter 13

13.1



where RMP means Return to Main Program
and RRC means Return to Recursive Call

13.2

```

function pos(function f(x : real) : real) : boolean;
  var x : real;
  begin
    x := -0.1;
    repeat
      x := x + 0.1
    until (x > 0.95) or ( f(x) < 0 );
    pos := f(x) < 0
  end {pos};

```

Problem

```

procedure try(i : integer);
  begin
    {initialise selection of positions
     for i-th queen};
    repeat
      MakeNextSelection;
      if safe then
        begin
          SetQueen;
          if i < 8 then
            begin
              try(i + 1);
              if not successful then
                RemoveQueen
            end
          end
        until successful or NoMorePositions
      end {try};

```

Chapter 14

14.1 At the beginning of step 4 add

$w := \text{nil};$

Just before using w in the statement

$w \uparrow . \text{status} := \text{CostFixed};$

a test can be inserted to see if w equals *nil*. If so, an error message can be produced and the program terminated.

14.2 A dummy header will be needed as well as a sentinel (see Wirth, 1975*b*, chapter 4). The change does not significantly alter the performance of the program.

14.3

```

procedure PrintRoute(dest : PtrToNode);
type
  RoutePtr = trouteitem;
  RouteItem =
    record
      ptr : PtrToNode;
      next : RoutePtr
    end;
var
  p : PtrToNode;
  RouteList, r : RoutePtr;
begin
  writeln('The best route is');
  p := dest; RouteList := nil;
  while p <> nil do
    begin
      new(r);
      r.ptr := p;
      r.next := RouteList;
      RouteList := r;
      p := p.RouteToHere
    end;
  while RouteList <> nil do
    begin
      writeln(RouteList.ptr.name);
      r := RouteList;
      RouteList := RouteList.next;
      dispose(r)
    end
  end { of PrintRoute };

```

14.4 A suitable data structure is a linked list of records, which are used to form a sequence of the members of the set. The members of the set should be kept in (ascending) order in the sequence to facilitate the programming of the set operations. For further details, see the references in chapter 9.

Bibliography

The Programming Language Pascal

- Addyman, A. M., *et al.* (1979), A draft description of Pascal, *Pascal News*, 14, and *Software Practice and Experience*, 9,5.
- Addyman, A. M. (1980), A Draft Proposal for Pascal, *SIGPLAN Notices*, 15, 4 and *Pascal News*, 18.
- Moffat, D. V. (1981), Pascal Bibliography (June 1981), *SIGPLAN Notices*, 16, 11.

Programming

- Aho, A. V., Hopcroft, J. E., and Ullman, J. D. (1974), *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, Mass.).
- Atkinson, L. V., Should **if . . . then . . . else** Follow the Dodo?, *Software Practice and Experience*, 9,9.
- Atkinson, L. V., Should **if . . . then . . . else** Follow the Dodo?, *Software Practice and Experience*, 9.
- Computing Surveys* (1974), 6, part 4 (complete issue).
- Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R. (1972), *Structured Programming* (Academic Press, New York).
- Knuth, D. (1973), *The Art of Computer Programming*, 3 vols (Addison-Wesley, Reading, Mass.).
- Wirth, N. (1971), Program development by step-wise refinement, *Communs ACM*, 14, 221–27.
- Wirth, N. (1975), *Algorithms + Data Structures = Programs* (Prentice-Hall, Englewood Cliffs, N. J.).

The Pascal Standard

Authorised reproduction of BS6192: 1982 Specification for Computer programming language Pascal.

Preface

This standard has been prepared under the direction of the Office and Information Standards Committee. The text following this preface has been prepared as an international standard to be published by the International Organization for Standardization (ISO). Following a precedent set by other programming languages (e.g. ISO 1539 'Programming language – FORTRAN') the intention is that the draft international standard, ISO/DIS 7185, and the resulting standard, ISO 7185, will refer to this British Standard for the whole of the technical content.

Terminology and conventions. For ease of reproduction, the actual text produced for the draft international standard, with minor editorial changes, has been used for this British Standard. As a result of using the international text, some terminology and certain conventions are not identical with those used in British Standards; attention is especially drawn to the following.

Wherever 'ISO 7185' appears, it should be read as 'BS 6192'.

Cross-reference

International standard	Corresponding British Standard
ISO 646- 1973	BS 4730: 1974 Specification for the United Kingdom 7-bit data code (Technically equivalent)

	Contents	Page		
	Foreword	164	6.6	Procedure and function declarations
	Specification			187
0.	Introduction	165	6.7	Expressions
1.	Scope	165	6.8	Statements
2.	Reference	165	6.9	Input and output
3.	Definitions	165	6.10	Programs
4.	Definitional conventions	166		Appendices
5.	Compliance	167	A.	Collected syntax
5.1	Processors	167	B.	Index
5.2	Programs	169	C.	Required identifiers
6.	Requirements	169	D.	Errors
6.1	Lexical tokens	169		Tables
6.2	Blocks, scope and activations	172	1.	Metalanguage symbols
6.3	Constant-definitions	174	2.	Dyadic arithmetic operations
6.4	Type-definitions	175	3.	Monadic arithmetic operations
6.5	Declarations and denotations of variables	184	4.	Set operations
			5.	Relational operations
				204

Foreword

The computer programming 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 that 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.

In drafting this standard the continued stability of Pascal has been a prime objective. However, apart from changes to clarify the specification, two major changes have been introduced.

- (1) The syntax used to specify procedural and functional parameters has been changed to require the use of a procedure or function heading, as appropriate (see 6.6.3.1); this change was introduced to overcome a language insecurity.
- (2) A fifth kind of parameter, the conformant array parameter, has been introduced (see 6.6.3.7). With this kind of parameter, the required bounds of the index-type of an actual parameter are not fixed, but are restricted to a specified range of values.

Editorial note. It is normal convention to use italic type for algebraic quantities. Since the status of such quantities contained in this standard may or may not directly represent true variable quantities, this convention has not been adopted in this standard.

British Standard Specification for

Computer programming language Pascal

0. Introduction

The appendices are included for the convenience of the reader of this standard. They do not form a part of the requirements of this standard.

1. Scope

1.1 This standard specifies the semantics and syntax of the computer programming language Pascal by specifying requirements for a processor and for a conforming program. Two levels of compliance are defined for both processors and programs.

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, nor the actions to be taken when the corresponding limits are exceeded;
- (b) the minimal requirements of a data processing system that is capable of supporting an implementation of a processor for Pascal;
- (c) the method of activating the program-block or 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;
- (e) the method for reporting errors or warnings;
- (f) the typographical representation of a program published for human reading.

2. Reference

ISO 646 : The 7-bit coded character set for information processing interchange

3. Definitions

For the purposes of this standard, the following definitions apply.

NOTE. To draw attention to language concepts, some terms are printed in *italics* on their first mention in this standard.

3.1 error. A violation by a program of the requirements of this standard that a processor is permitted to leave undetected.

NOTES

1. If it is possible to construct a program in which the violation or non-violation of this standard requires knowledge of the data read by the program or the implementation definition of implementation-defined features, then violation of that requirement is classified as an *error*. Processors may report on such violations of the requirement without such knowledge, but there always remain some cases that require execution or simulated execution, or proof procedures with the required knowledge. Requirements that can be verified without such knowledge are not classified as errors.

2. Processors should attempt the detection of as many errors as possible, and to as complete a degree as possible. Permission to omit detection is provided for implementations in which the detection would be an excessive burden, or which are not of the highest quality.

3.2 extension. A modification to clause 6 of this standard that neither invalidates any program complying with the requirements of this standard, as defined by 5.2, except by prohibiting the use of one or more particular spellings of identifiers, nor alters the status of any implementation-dependent feature or error.

3.3 implementation-defined. Possibly differing between processors, but defined for any particular processor.

3.4 implementation-dependent. Possibly differing between processors and not necessarily defined for any particular processor.

3.5 processor. A system or mechanism that accepts a program as input, prepares it for execution, and executes the process so defined with data to produce results.

NOTE. A processor may consist of an interpreter, a compiler and run-time system, or other mechanism, together with an associated host computing machine and operating system, or other mechanism for achieving the same effect. A compiler in itself, for example, does not constitute a processor.

4. Definitional conventions

The metalanguage used in this standard to specify the syntax of the constructs is based on Backus-Naur Form. The notation has been modified from the original to permit greater convenience of description and to allow for iterative productions to replace recursive ones. Table 1 lists the meanings of the various metasymbols. Further specification of the constructs is given by prose and, in some cases, by equivalent program fragments. Any identifier that is defined in clause 6 as a required identifier shall denote the corresponding required entity by its occurrence in such a program fragment. In all other respects, any such program fragment is bound by any pertinent requirement of this standard.

Table 1. Metalanguage symbols

Metasymbol	Meaning
=	shall be defined to be
>	shall have as an alternative definition
!	alternatively
.	end of definition
[x]	0 or 1 instance of x
{ x }	0 or more instances of x
(x ! y)	grouping: either of x or y
"xyz"	the terminal symbol xyz
meta-identifier	a non-terminal symbol

A meta-identifier shall be a sequence of letters and hyphens beginning with a letter.

A sequence of terminal and non-terminal symbols in a production implies the concatenation of the text that they ultimately represent. Within 6.1 this concatenation is direct; no characters shall intervene. In all other parts of this standard the concatenation is in accordance with the rules set out in 6.1.

The characters required to form Pascal programs shall be those implicitly required to form the tokens and separators defined in 6.1.

Use of the words *of*, *in*, *containing* and *closest-containing* when expressing a relationship between terminal or non-terminal symbols shall have the following meanings.

the x *of* a y: refers to the x occurring directly in a production defining y.

the x *in* a y: is synonymous with 'the x of a y'.

a y *containing* an x: refers to any y from which an x is directly or indirectly derived.

the y *closest-containing* an x: that y which contains an x but does not contain another y containing that x.

These syntactic conventions are used in clause 6 to specify certain syntactic requirements and also the contexts within which certain semantic specifications apply.

5. Compliance

NOTE. There are two levels of compliance, level 0 and level 1. Level 0 does not include conformant array parameters. Level 1 does include conformant array parameters.

5.1 Processors. A processor complying with the requirements of this standard shall:

- (a) If it complies at level 0, accept all the features of the language specified in

clause 6, except for 6.6.3.6(e), 6.6.3.7 and 6.6.3.8, with the meanings defined in clause 6;

(b) if it complies at level 1, accept all the features of the language specified in clause 6 with the meanings defined in clause 6;

(c) not require the inclusion of substitute or additional language elements in a program in order to accomplish a feature of the language that is specified in clause 6;

(d) be accompanied by a document that provides a definition of all implementation-defined features;

(e) be able to determine whether or not a program violates any requirement of this standard, where such a violation is not designated an error, and report the result of this determination to the user of the processor; in the case where the processor does not examine the whole program, the user shall be notified that the determination is incomplete whenever no violations have been detected in the program text examined;

(f) treat each violation that is designated 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 report during preparation of the program for execution 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, and terminate execution of the program;

and if any violations that are designated as errors are treated in the manner described in 5.1(f)(1), then a note referencing each such treatment shall appear in a separate section of the accompanying document;

(g) be accompanied by a document that separately describes any features accepted by the processor that are prohibited or not specified in clause 6: such extensions shall be described as being 'extensions to Pascal as specified by ISO 7185';

(h) be able to process in a manner similar to that specified for errors any use of any such extension;

(i) be able to process in a manner similar to that specified for errors any use of an implementation-dependent feature.

NOTES

1. The phrase 'be able to' is used in 5.1 to permit the implementation of a switch with which the user may control the reporting.

2. In cases where the compilation is aborted due to some limitation of tables, etc., an incomplete determination of the kind 'No violations were detected, but the examination is incomplete.' will satisfy the requirements of clause 5.1(e). In a similar manner, an interpretive or direct execution processor may report an incomplete determination for a program of which all aspects have not been examined.

A processor that purports to comply, wholly or partially, with the requirements of this

standard shall do so only in the following terms. A *compliance statement* shall be produced by the processor as a consequence of using the processor, or shall be included in accompanying documentation. If the processor complies in all respects with the requirements of this standard the compliance statement shall be:

<This processor> complies with the requirements of level <number> of ISO 7185.

If the processor complies with some but not all of the requirements of this standard then it shall not use the above statement, but shall instead use the following compliance statement:

<This processor> complies with the requirements of level <number> of ISO 7185,

with the following exceptions:

<followed by a reference to, or a complete list of, the requirements of the standard with which the processor does not comply>.

In both cases the text *<This processor>* shall be replaced by an unambiguous name identifying the processor, and the text *<number>* shall be replaced by the appropriate level number.

NOTE 3. Processors that do not comply fully with the requirements of the standard are not required to give full details of their failures to comply in the compliance statement; a brief reference to accompanying documentation that contains a complete list in sufficient detail to identify the defects is sufficient.

5.2 Programs. A program complying with the requirements of this standard shall:

(a) if it complies at level 0, use only those features of the language specified in clause 6, except for 6.6.3.6(e), 6.6.3.7 and 6.6.3.8;

(b) if it complies at level 1, use only those features of the language specified in clause 6;

(c) not rely on any particular interpretation of implementation-dependent features.

NOTES

1. A program that complies with the requirements of this standard may rely on particular implementation-defined values or features.

2. The requirements for compliant programs and compliant processors do not require that the results produced by a compliant program are always the same when processed by a compliant processor. They may be, or they may differ, depending on the program. A simple program to illustrate this is:
program x(output); begin writein(maxint) end.

6. Requirements

6.1 Lexical tokens

NOTE. The syntax given in this subclause (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 representation of any letter (upper-case or lower-case, differences of font, etc.) 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 may be of any length. All characters of an identifier shall be significant in distinguishing between identifiers. No identifier shall have the same spelling as any word-symbol. Identifiers that are specified to be *required* shall have special significance (see 6.2.2.10 and 6.10).

identifier = letter { letter | digit } .

Examples:

```
X
time
readinteger
WG4
AlterHeatSetting
InquireWorkstationTransformation
InquireWorkstationIdentification
```

6.1.4 Directives. A directive shall occur only in a procedure-declaration or function-declaration. The directive *forward* shall be the only required directive (see 6.6.1 and 6.6.2). No directive shall have the same spelling as any word-symbol.

directive = letter { letter | digit } .

NOTE. Many processors provide, as an extension, the directive *external*, which is used to specify that the procedure-block or function-block corresponding to that procedure-heading or function-heading is external to the program-block. Usually it is in a library in a form to be input to, or that has been produced by, the processor.

6.1.5 Numbers. An unsigned-integer shall denote in decimal notation a value of integer-type (see 6.4.2.2). An unsigned-real shall denote in decimal notation a value of real-type (see 6.4.2.2). The letter 'e' preceding a scale factor shall mean *times ten to the power of*. The value denoted by an unsigned-integer shall be in the closed interval 0 to *maxint* (see 6.4.2.2 and 6.7.2.2).

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

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 and shall be in the closed interval 0 to 9999.

label = digit-sequence .

6.1.7 Character-strings. A character-string containing a single string-element shall denote a value of the required char-type (see 6.4.2.2). A character-string containing more than one string-element shall denote a value of a string-type (see 6.4.3.2) with the same number of components as the character-string contains string-elements. Each string-character shall denote an implementation-defined value of the required char-type, subject to the restriction that no such value shall be denoted by more than one string-element.

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

NOTE. Conventionally, the apostrophe-image is regarded as a substitute for the apostrophe character, which cannot be a string-character.

Examples:

```
'A'
','
''
''''
'Pascal'
'THIS IS A STRING'
```

6.1.8 Token separators. The construct

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

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

Comments, spaces (except in character-strings), and the separation of consecutive 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, labels or unsigned-numbers. No separators shall occur within tokens.

6.1.9 Lexical alternatives. The representation for lexical tokens and separators given in 6.1.1 to 6.1.8 shall constitute a *reference representation* for these tokens and separators. The reference representation shall be used for program interchange.

To facilitate the use of Pascal on processors that do not support the reference representation, the following alternatives have been defined. All processors that have the required characters in their character set shall provide both the reference

representations and the alternative representations, and the corresponding tokens or separators shall not be distinguished.

The alternative representations for the tokens shall be:

Reference token	Alternative token
↑	@
{	(
})

NOTE 1. The character ↑ that appears in some national variants of ISO 646 is regarded as identical to the character ^ . In this standard, the character ↑ has been used because of its greater visibility.

The alternative forms of comment shall be all forms of comment where one or both of the following substitutions are made:

Delimiting character	Alternative delimiting pair of characters
{	(*
}	*)

NOTES

2. A comment may thus commence with { and end with *), or commence with (* and end with }.

3. If the sequence (**) occurs in a comment, it is equivalent to {} and marks the end of the comment, because the substitution is only for a delimiting character.

4. See also 1.2(f).

6.2 Blocks, scope and activations

6.2.1 Block. A block closest-containing a label-declaration-part in which a label occurs shall closest-contain exactly one statement in which that label occurs. The occurrence of a label in the label-declaration-part of a block shall be its defining-point as a label for the region that is the block.

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

6.2.2 Scope

6.2.2.1 Each identifier or label contained by the program-block 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, 6.5.3.3, 6.6.1, 6.6.2, 6.6.3.1, 6.8.3.10, 6.10).

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 has a defining-point for region A and another identifier or label having the same spelling has a 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 region that is the field-specifier of a field-designator shall be excluded from the enclosing scopes.

6.2.2.7 When an identifier or label has a defining-point for a region, another identifier or label with the same spelling shall not have a defining-point for that region.

6.2.2.8 Within the scope of a defining-point of an identifier or label, each occurrence of an identifier or label having the same spelling as the identifier or label of the defining-point shall be designated an applied occurrence of the identifier or label of the defining-point, except for an occurrence that constituted the defining-point of that identifier or label; such an occurrence shall be designated a defining occurrence. No occurrence outside that scope shall be an applied occurrence.

NOTE. Within the scope of a defining-point of an identifier or label, there are no applied occurrences of an identifier or label that cannot be distinguished from it and have a defining-point for a region enclosing that scope.

6.2.2.9 The defining-point of an identifier or label shall precede all applied occurrences of that identifier or label contained by the program-block with one exception, namely that an identifier may have an applied occurrence in the type-identifier of the domain-type of any new-pointer-types contained by the type-definition-part that contains the defining-point of the type-identifier.

6.2.2.10 Identifiers that denote required constants, types, procedures and functions shall be used as if their defining-points have a region enclosing the program (see 6.1.3, 6.4.1 and 6.6.4.1).

NOTE. The required identifiers *input* and *output* are not included, since these denote variables.

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

NOTE. Within syntax definitions, an applied occurrence of an identifier is qualified, e.g. type-identifier, whereas a use that constitutes a defining-point is not qualified.

6.2.3 Activations

6.2.3.1 A procedure-identifier or function-identifier having a defining-point for a region that is a block within the procedure-and-function-declaration-part of that block shall be designated *local* to that block.

6.2.3.2 The activation of a block shall contain:

- (a) for the statement-part of the block, an algorithm, the completion of which shall terminate the activation (see also 6.8.2.4);
- (b) for each label in a statement having a defining-point in the label-declaration-part of the block, a program-point in the algorithm of the activation of that statement;
- (c) for each variable-identifier having a defining-point for the region that is the block, a variable possessing the type associated with the variable-identifier;
- (d) for each procedure-identifier local to the block, a procedure with the the procedure-block corresponding to the procedure-identifier, and the formal parameters of that procedure-block;
- (e) for each function-identifier local to the block, a function with the function-block corresponding to, and the result type associated with, the function-identifier, and the formal parameters of that function-block;
- (f) if the block is a function-block, a result possessing the associated result type.

NOTE. Each activation contains its own algorithm, set of program-points, set of variables, set of procedures, and set of functions, distinct from every other activation.

6.2.3.3 The activation of a procedure or function shall be the activation of the block of its procedure-block or function-block, respectively, and shall be designated as *within*:

- (a) the activation containing the procedure or function; *and*
- (b) all activations that that containing activation is *within*.

NOTE. An activation of a block B can only be within activations of blocks containing B. Thus an activation is not within another activation of the same block.

Within an activation, an applied occurrence of a label or variable-identifier, or of a procedure-identifier or function-identifier local to the block of the activation, shall denote the corresponding program-point, variable, procedure, or function, respectively, of that activation; except that the function-identifier of an assignment-statement shall, within an activation of the function denoted by that function-identifier, denote the result of that activation.

6.2.3.4 A procedure-statement or function-designator contained in the algorithm of an activation and that specifies the activation of a block shall be designated the activation-point of that activation of the block.**6.2.3.5** All variables contained by an activation, except for those listed as program-parameters, and any result of an activation, shall be totally-undefined at the commencement of that activation. The algorithm, program-points, variables, procedures and functions, if any, shall exist until the termination of the activation.**6.3 Constant-definitions.** A constant-definition shall introduce an identifier to denote a value.

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

The occurrence of an identifier in a constant-definition of a constant-definition-part of a block shall constitute its defining-point for the region that is the block. The constant in a constant-definition shall not contain an applied occurrence of the identifier in the constant-definition. Each applied occurrence of that identifier shall be a constant-identifier and shall denote the value denoted by the constant of the constant-definition. A constant-identifier in a constant containing an occurrence of a sign shall have been defined to denote a value of real-type or of integer-type. The required constant-identifiers shall be as specified in 6.4.2.2 and 6.7.2.2

6.4 Type-definitions

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

```
type-definition = identifier "=" type-denoter .
type-denoter = type-identifier ! new-type .
new-type = new-ordinal-type ! new-structured-type ! new-pointer-type .
```

The occurrence of an identifier in a type-definition of a type-definition-part of a block shall constitute its defining-point for the region that is the block. Each applied occurrence of that identifier shall be a type-identifier and shall denote the same type as that which is denoted by the type-denoter of the type-definition. Except for applied occurrences in the domain-type of a new-pointer-type, the type-denoter shall not contain an applied occurrence of the identifier in the type-definition.

Types shall be classified as simple, structured or pointer types. The required type-identifiers and corresponding required types shall be as specified in 6.4.2.2 and 6.4.3.5.

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

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.

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. An ordinal-type-identifier shall denote an ordinal-type.

```
simple-type = ordinal-type ! real-type-identifier .
ordinal-type = new-ordinal-type ! ordinal-type-identifier .
new-ordinal-type = enumerated-type ! subrange-type .
ordinal-type-identifier = type-identifier .
real-type-identifier = type-identifier .
```

6.4.2.2 Required simple-types. The following types shall exist:

- (a) *integer-type*. The required ordinal-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 signed-integer (see also 6.7.2.2). The ordinal number of a value of integer-type shall be the value itself.
- (b) *real-type*. The required 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 signed-real.

(c) *Boolean-type*. The required ordinal-type-identifier *Boolean* shall denote the Boolean-type. The values shall be the enumeration of truth values denoted by the required 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.

(d) *char-type*. The required ordinal-type-identifier *char* shall denote the char-type. The values 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.

- (1) The subset of character values representing the digits 0 to 9 shall be numerically ordered and contiguous.
- (2) The subset of character values representing the upper-case letters A to Z, if available, shall be alphabetically ordered but not necessarily contiguous.
- (3) The subset of character values representing the lower-case letters a to z, if available, shall be alphabetically ordered but not necessarily contiguous.
- (4) The ordering relationship between any two character values shall be the same as between their ordinal numbers.

NOTE. Operators applicable to the required simple-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 on to consecutive non-negative values of integer-type starting from zero. The mapping shall be order preserving.

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

The occurrence of an identifier in the identifier-list of an enumerated-type shall constitute its defining-point as a constant-identifier for the region that is the block closest-containing the enumerated-type.

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. A subrange-type shall include identification of the smallest and the largest value in the subrange. The first constant of a subrange-type shall specify the smallest value, and this shall be less than or equal to the largest value which shall be specified by the other constant of the subrange-type. 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. A new-structured-type shall be classified as an array-type, record-type, set-type or file-type according to the unpacked-structured-type closest-contained by the new-structured-type. A component of a value of a structured-type shall be a value.

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

The occurrence of the token *packed* in a new-structured-type shall designate the type denoted thereby as *packed*. The designation of a structured-type as *packed* shall indicate to the processor that data-storage of values should be economized, even if this causes operations on, or accesses to components of, variables possessing the type to be less efficient in terms of space or time.

The designation of a structured-type as *packed* shall affect the representation in data-storage of that structured-type only; i.e., 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. The ways in which the treatment of entities of a type is affected by whether or not the type is designated *packed* are specified in 6.4.3.2, 6.4.5, 6.6.3.3, 6.6.3.8, 6.6.5.4 and 6.7.1.

6.4.3.2 Array-types. An array-type shall be structured as a mapping from each value specified by its index-type on to a distinct component. Each component shall have the type denoted by the type-denoter of the component-type of the array-type.

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

Examples 1:

```
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 abbreviated notation for an array-type specified to have as its index-type 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*. The abbreviated form and the full form shall be equivalent.

NOTE 1. Each of the following two examples thus contains different ways of expressing its array-type.

Example 2:

```
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 3:

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 specified by the index-type be denoted by m and n ; and let $k = (\text{ord}(n) - \text{ord}(m) + 1)$ denote the number of values specified by the index-type; then the values of the array-type shall be the distinct k -tuples of the form

$(v[m], \dots, v[n])$.

NOTE 2 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 designated packed and denoted by an array-type having as its index-type a denotation of a subrange-type specifying a smallest value of 1 and a largest value of greater than 1, and having as its component-type a denotation of the char-type, shall be designated a *string-type*.

The correspondence of character-strings to values of string-types is obtained by relating the individual string-elements of the character-string, taken in textual order, to the components of the values of the string-type in order of increasing index.

NOTE 3. The values of a string-type possess additional properties which allow writing them to textfiles (see 6.9.3.6) and define their use with relational-operators (see 6.7.25).

6.4.3.3 Record-types. The structure and values of a record-type shall be the structure and values of the field-list of the record-type.

```
record-type = "record" field-list "end" .
field-list = [ ( fixed-part [ ":" variant-part ] | variant-part ) [ ":" ] ] .
fixed-part = record-section { ":" record-section } .
record-section = identifier-list ":" type-denoter .
variant-part = "case" variant-selector "of" variant { ":" variant } .
variant-selector = [ tag-field ":" ] tag-type .
tag-field = identifier .
variant = case-constant-list ":" "(" field-list ")" .
tag-type = ordinal-type-identifier .
case-constant-list = case-constant { "," case-constant } .
case-constant = constant .
```

A field-list that contains neither a fixed-part nor a variant-part shall have no components, shall define a single null value, and shall be designated *empty*.

The occurrence of an identifier in the identifier-list of a record-section of a fixed-part of a field-list shall constitute its defining-point as a field-identifier for the region that is the record-type closest-containing the field-list, and shall associate the field-identifier with a distinct component, which shall be designated a *field*, of the record-type and of the field-list. That component shall have the type denoted by the type-denoter of the record-section.

The field-list closest-containing a variant-part shall have a distinct component that shall have the values and structure defined by the variant-part.

Let V_i denote the value of the i -th component of a non-empty field-list having m components; then the values of the field-list shall be distinct m -tuples of the form

(V_1, V_2, \dots, V_m) .

NOTE 1. If the type of the i -th component has F_i values, then the cardinality of the set of values of the field-list shall be $(F_1 * F_2 * \dots * F_m)$.

A tag-type shall denote the type denoted by the ordinal-type-identifier of the tag-type.
A case-constant shall denote the value denoted by the constant of the case-constant.

The type of each case-constant in the case-constant-list of a variant of a variant-part shall be compatible with the tag-type of the variant-selector of the variant-part. The values denoted by all case-constants of a type that is required to be compatible with a given tag-type shall be distinct and the set thereof shall be equal to the set of values specified by the tag-type. The values denoted by the case-constants of the case-constant-list of a variant shall be designated as corresponding to the variant.

With each variant-part shall be associated a type designated the selector-type possessed by the variant-part. If the variant-selector of the variant-part contains a tag-field, or if the case-constant-list of each variant of the variant-part contains only one case-constant, then the selector-type shall be denoted by the tag-type, and each variant of the variant-part shall be associated with those values specified by the selector-type denoted by the case-constants of the case-constant-list of the variant. Otherwise, the selector-type possessed by the variant-part shall be a new ordinal-type constructed such that there is exactly one value of the type for each variant of the variant-part, and no others, and each variant shall be associated with a distinct value of that type.

Each variant-part shall have a component that shall be designated the *selector* of the variant-part, and which shall possess the selector-type of the variant-part. If the variant-selector of the variant-part contains a tag-field, then the occurrence of an identifier in the tag-field shall constitute the defining-point of the identifier as a field-identifier for the region that is the record-type closest-containing the variant-part, and shall associate the field-identifier with the selector of the variant-part. The selector shall be designated a field of the record-type if and only if it is associated with a field-identifier.

Each variant of a variant-part shall denote a distinct component of the variant-part; the component shall have the values and structure of the field-list of the variant, and shall be associated with those values specified by the selector-type possessed by the variant-part associated with the variant. The value of the selector of the variant-part shall cause the associated variant and component of the variant-part to be in a state that shall be designated *active*.

The values of a variant-part shall be the distinct pairs

(k, X_k)

where k represents a value of the selector of the variant-part, and X_k is a value of the field-list of the active variant of the variant-part.

NOTES

2. If there are n values specified by the selector-type, and if the field-list of the variant associated with the i -th value has T_i values, then the cardinality of the set of values of the variant-part is $(T_1 + T_2 + \dots + T_n)$. There is no component of a value of a variant-part corresponding to any non-active variant of the variant-part.

3. Restrictions placed on the use of fields of a record-variable pertaining to variant-parts are specified in 6.5.3.3, 6.6.3.3 and 6.6.5.3.

Examples:

```
record
  year : 0..2000;
  month : 1..12;
  day : 1..31
end
```

```
record
  name, firstname : string;
  age : 0..99;
  case married : Boolean of
    true : (Spousesname : string);
    false : 0
  end
end
```

```
record
  x, y : real;
  area : real;
  case shape of
    triangle :
      (side : real; inclination, angle1, angle2 : angle);
    rectangle :
      (side1, side2 : real; skew : angle);
    circle :
      (diameter : real);
  end
end
```

6.4.3.4 Set-types. A set-type shall determine the set of values that is structured as the powerset of the base-type of the set-type. Thus each value of a set-type shall be a set whose members shall be unique values of the base-type.

```
set-type = "set" "of" base-type .
base-type = ordinal-type .
```

NOTE 1. Operators applicable to values of set-types are specified in 6.7.24.

Examples:

```
set of char
set of (club, diamond, heart, spade)
```

NOTE 2. If the base-type of a set-type has b values then the cardinality of the set of values is 2 raised to the power b .

For every ordinal-type S , there exists an unpacked set type designated the *unpacked canonical set-of- T type* and there exists a packed set type designated the *packed canonical set-of- T type*. If S is a subrange-type then T is the host type of S ; otherwise T is S . Each value of the type set of S is also a value of the unpacked canonical set-of- T type, and each value of the type packed set of S is also a value of the packed canonical set-of- T type.

6.4.3.5 File-types

NOTE 1. A file-type describes sequences of values of the specified component-type, together with a current position in each sequence and a mode that indicates whether the sequence is being inspected or generated.

file-type = "file" "of" component-type .

A type-denoter shall not be permissible as the component-type of a file-type if it denotes either a file-type or a structured-type having any component whose type-denoter is not permissible as the component-type of a file-type.

Examples:

file of real
file of vector

A file-type shall define implicitly a type designated a *sequence-type* having exactly those values, which shall be designated *sequences*, defined by the following five rules in items (a) to (e).

NOTE 2 The notation x^y represents the concatenation of sequences x and y . The explicit representation of sequences (e.g. $S(c)$), of concatenation of sequences, of the first, last and rest selectors, and of sequence equality is not part of the Pascal language. These notations are used to define file values, below, and the required file operations in 6.6.5.2 and 6.6.5.5.

- (a) $S0$ shall be a value of the sequence-type S , and shall be designated the *empty sequence*. The empty sequence shall have no components.
- (b) Let c be a value of the specified component-type, and let x be a value of the sequence-type S ; then $S(c)$ shall be a sequence of type S , consisting of the single component value c , and both $S(c)^x$ and $x^S(c)$ shall be sequences, distinct from $S0$, of type S .
- (c) Let c , S , and x be as in (b); let y denote the sequence $S(c)^x$; and let z denote the sequence $x^S(c)$; then the notation $y.first$ shall denote c (i.e., the first component value of y), $y.rest$ shall denote x (i.e., the sequence obtained from y by deleting the first component), and $z.last$ shall denote c (i.e., the last component value of z).
- (d) Let x and y each be a non-empty sequence of type S ; then $x = y$ shall be true if and only if both $(x.first = y.first)$ and $(x.rest = y.rest)$ are true. If x or y is the empty sequence, then $x = y$ shall be true if and only if both x and y are the empty sequence.
- (e) Let x , y , and z be sequences of type S ; then $x^y(z) = (x^y)^z$, $S0^x = x$, and $x^S0 = x$ shall be true.

A file-type also shall define implicitly a type designated a *mode-type* having exactly two values which are designated *Inspection* and *Generation*.

NOTE 3. The explicit denotation of the values *Inspection* and *Generation* is not part of the Pascal language.

A file-type shall be structured as three components. Two of these components, designated $f.L$ and $f.R$, shall be of the implicit sequence-type. The third component, designated $f.M$, shall be of the implicit mode-type.

Let $f.L$ and $f.R$ each be a single value of the sequence-type; let $f.M$ be a single value of the mode-type; then each value of the file-type shall be a distinct triple of the form

$(f.L, f.R, f.M)$

where $f.R$ shall be the empty sequence if $f.M$ is the value *Generation*. The value, f , of the file-type shall be designated empty if and only if $f.L^f.R$ is the empty sequence.

NOTE 4. The two components, *f.L* and *f.R*, of a value of the file-type may be considered to represent the single sequence *f.L~f.R* together with a current position in that sequence. If *f.R* is non-empty, then *f.R.first* may be considered the current component as determined by the current position; otherwise, the current position is designated the end-of-file position.

There shall be a file-type that is denoted by the required structured-type-identifier *text*. The structure of the type denoted by *text* shall define an additional sequence-type whose values shall be designated *lines*. A line shall be a sequence *cs~S(e)*, where *cs* is a sequence of components having the char-type, and *e* represents a special component value, which shall be designated an *end-of-line*, and which shall be indistinguishable from the char value space except by the required function *eofln* (see 6.6.6.5) and by the required procedures *reset* (see 6.6.5.2), *writeln* (see 6.9.4), and *page* (see 6.9.5). If *l* is a line then no component of *l* other than *l.last* shall be an end-of-line. This definition shall not be construed to determine the underlying representation, if any, of an end-of-line component used by a processor.

A line-sequence, *ls*, shall be either the empty sequence or the sequence *l~ls'* where *l* is a line and *ls'* is a line-sequence.

Every value *t* of the type denoted by *text* shall satisfy one of the following two rules.

- (a) If *t.M* = Inspection, then *t.L~t.R* shall be a line-sequence.
- (b) If *t.M* = Generation, then *t.L~t.R* shall be *ls~cs*, where *ls* is a line-sequence and *cs* is a sequence of components possessing the char-type.

NOTE 5. In rule (b), *cs* may be considered, especially if it is non-empty, to be a partial line that is being generated. Such a partial line cannot occur during inspection of a file. Also, *cs* does not correspond to *t.R* since *t.R* is the empty sequence if *t.M* = Generation.

A variable that possesses the type denoted by the required structured-type-identifier *text* shall be designated a *textfile*.

NOTE 6. All required procedures and functions applicable to a variable of type *file of char* are applicable to textfiles. Additional required procedures and functions, applicable only to textfiles, are defined in 6.6.6.5 and 6.9.

6.4.4 Pointer-types. The values of a pointer-type shall consist of a single nil-value, and a set of identifying-values each identifying a distinct variable possessing the domain-type of the pointer-type. The set of identifying-values shall be dynamic, in that the variables and the values identifying them may be created and destroyed during the execution of the program. Identifying-values and the variables identified by them shall be created only by the required procedure *new* (see 6.6.5.3).

NOTE 1. Since the nil-value is not an identifying-value it does not identify a variable.

The token *nil* shall denote the nil-value in all pointer-types.

pointer-type = new-pointer-type | pointer-type-identifier .
 new-pointer-type = "†" domain-type .
 domain-type = type-identifier .

NOTE 2. The token *nil* does not have a single type, but assumes a suitable pointer-type to satisfy the assignment-compatibility rules, or the compatibility rules for operators, if possible.

6.4.5 Compatible types. Types *T1* and *T2* shall be designated *compatible* if any of the following four statements is true.

- (a) *T1* and *T2* are the same type.

(b) T1 is a subrange of T2, or T2 is a subrange of T1, or both T1 and T2 are subranges of the same host type.

(c) T1 and T2 are set-types of compatible base-types, and either both T1 and T2 are designated packed or neither T1 nor T2 is designated packed.

(d) T1 and T2 are string-types with the same number of components.

6.4.6 Assignment-compatibility. A value of type T2 shall be designated *assignment-compatible* with a type T1 if any of the following five statements is true.

(a) T1 and T2 are the same type and that type is permissible as the component-type of a file-type (see 6.4.3.5).

(b) T1 is the real-type and T2 is the integer-type.

(c) T1 and T2 are compatible ordinal-types and the value of type T2 is in the closed interval specified by the type T1.

(d) T1 and T2 are compatible set-types and all the members of the value of type T2 are in the closed interval specified by the base-type of T1.

(e) T1 and T2 are compatible string-types.

At any place where the rule of assignment-compatibility is used:

(1) it shall be an error if T1 and T2 are compatible ordinal-types and the value of type T2 is not in the closed interval specified by the type T1;

(2) it shall be an error if T1 and T2 are compatible set-types and any member of the value of type T2 is not in the closed interval specified by the base-type of the type T1.

6.4.7 Example of a type-definition-part

type

```
natural = 0..maxint;
count = integer;
range = integer;
colour = (red, yellow, green, blue);
sex = (male, female);
year = 1900..1999;
shape = (triangle, rectangle, circle);
punchedcard = array [1..80] of char;
charsequence = file of char;
polar = record
    r : real;
    theta : angle;
end;
indextype = 1..limit;
vector = array [indextype] of real;
person = †persondetails;
```

```

persondetails = record
    name, firstname : charsequence;
    age : integer;
    married : Boolean;
    father, child, sibling : person;
    case s : sex of
        male :
            (enlisted, bearded : Boolean);
        female :
            (mother, programmer : Boolean);
    end;
FileOfInteger = file of integer;

```

NOTE. In the above example *count*, *range* and *integer* denote the same type. The types denoted by *year* and *natural* are compatible with, but not the same as, the type denoted by *range*, *count* and *integer*.

6.5 Declarations and denotations of variables

6.5.1 Variable-declarations. A variable shall be an entity to which a value may be attributed (see 6.8.2.2). Each identifier in the identifier-list of a variable-declaration shall denote a distinct variable possessing the type denoted by the type-denoter of the variable-declaration.

variable-declaration = identifier-list ":" type-denoter .

The occurrence of an identifier in the identifier-list of a variable-declaration of the variable-declaration-part of a block shall constitute its defining-point as a variable-identifier for the region that is the block. The structure of a variable possessing a structured-type shall be the structure of the structured-type. A use of a variable-access shall be an access, at the time of the use, to the variable thereby denoted. A variable-access, according to whether it is an entire-variable, a component-variable, an identified-variable, or a buffer-variable, shall denote either a declared variable, or a component of a variable, a variable which is identified by a pointer value (see 6.4.4), or a buffer-variable, respectively.

variable-access = entire-variable ! component-variable ! identified-variable !
buffer-variable .

Example of a variable-declaration-part:

```

var
    x, y, z, max : real;
    i, j : integer;
    k : 0..9;
    p, q, r : Boolean;
    operator : (plus, minus, times);
    a : array [0..63] of real;
    c : colour;
    f : file of char;
    hue1, hue2 : set of colour;
    p1, p2 : person;
    m, m1, m2 : array [1..10, 1..10] of real;
    coord : polar;
    pooltape : array [1..4] of FileOfInteger;

date : record
    month : 1..12;
    year : integer
end;

```

NOTE. Variables occurring in examples in the remainder of this standard should be assumed to have been declared as specified in 6.5.1.

6.5.2 Entire-variables

entire-variable = variable-identifier .
variable-identifier = identifier .

6.5.3 Component-variables

6.5.3.1 General. A component of a variable shall be a variable. A component-variable shall denote a component of a variable. A reference, or access to a component of a variable shall constitute a reference, or access, respectively, to the variable. The value, if any, of the component of a variable shall be the same component of the value, if any, of the variable.

component-variable = indexed-variable ! field-designator .

6.5.3.2 Indexed-variables. A component of a variable possessing an array-type shall be denoted by an indexed-variable.

indexed-variable = array-variable "[" index-expression { "," index-expression } "]" .
array-variable = variable-access .
index-expression = expression .

An array-variable shall be a variable-access that denotes a variable possessing an array-type. For an indexed-variable closest-containing a single index-expression, the value of the index-expression shall be assignment-compatible with the index-type of the array-type. The component denoted by the indexed-variable shall be the component that corresponds to the value of the index-expression by the mapping of the type possessed by the array-variable (see 6.4.3.2).

Examples 1:

```
a[12]
a[i + j]
m[k]
```

If the array-variable is itself an indexed-variable an abbreviation may be used. In the abbreviated form, a single comma shall replace the sequence `] [` that occurs in the full form. The abbreviated form and the full form shall be equivalent.

The order of evaluation of the index-expressions of an indexed-variable shall be implementation-dependent.

Examples 2:

```
m[k][1]
m[k, 1]
```

NOTE. These two examples denote the same component variable.

6.5.3.3 Field-designators. A field-designator either shall denote that component of the record-variable of the field-designator associated with the field-identifier of the field-specifier of the field-designator, by the record-type possessed by the record-variable; or shall denote the variable denoted by the field-designator-identifier (see 6.8.3.10) of the field-designator. A record-variable shall be a variable-access that denotes a variable possessing a record-type.

The occurrence of a record-variable in a field-designator shall constitute the

defining-point of the field-identifiers associated with components of the record-type possessed by the record-variable, for the region that is the field-specifier of the field-designator.

```
field-designator = record-variable "." field-specifier ; field-designator-identifier .
record-variable = variable-access .
field-specifier = field-identifier .
field-identifier = identifier .
```

Examples:

```
p2f.mother
coord.theta
```

An access to a component of a variant of a variant-part, where the selector of the variant-part is not a field, shall attribute to the selector that value specified by its type associated with the variant. It shall be an error unless a variant is active for the entirety of each reference and access to each component of the variant.

When a variant becomes not active, all of its components shall become totally-undefined.

NOTE. If the selector of a variant-part is undefined, then no variant of the variant-part is active.

6.5.4 Identified-variables. An identified-variable shall denote the variable (if any) identified by the value of the pointer-variable of the identified-variable (see 6.4.4 and 6.6.5.3).

```
identified-variable = pointer-variable "↑" .
pointer-variable = variable-access .
```

A variable created by the required procedure *new* (see 6.6.5.3) shall be accessible until the termination of the activation of the program-block or until the variable is made inaccessible (see the required procedure *dispose*, 6.6.5.3).

NOTE. The accessibility of the variable also depends on the existence of a pointer-variable that has attributed to it the corresponding identifying value.

A pointer-variable shall be a variable-access that denotes a variable possessing a pointer-type. It shall be an error if the pointer-variable of an identified-variable either denotes a nil-value or is undefined. It shall be an error to remove from its pointer-type the identifying-value of an identified variable (see 6.6.5.3) when a reference to the identified variable exists.

Examples:

```
p1↑
p1↑.father↑
p1↑.sibling↑.father↑
```

6.5.5 Buffer-variables. A file-variable shall be a variable-access that denotes a variable possessing a file-type. A buffer-variable shall denote a variable associated with the variable denoted by the file-variable of the buffer-variable. A buffer-variable associated with a textfile shall possess the char-type; otherwise, a buffer-variable shall possess the component-type of the file-type possessed by the file-variable of the buffer-variable.

```
buffer-variable = file-variable "↑" .
file-variable = variable-access .
```


Examples:

```
inputt
pooltape[2]†
```

It shall be an error to alter the value of a file-variable *f* when a reference to the buffer-variable *ft* exists. A reference or access to a buffer-variable shall constitute a reference or access, respectively, to the associated file-variable.

6.6 Procedure and function declarations

6.6.1 Procedure-declarations

```
procedure-declaration = procedure-heading ":" directive ;
                        procedure-identification ":" procedure-block ;
                        procedure-heading ":" procedure-block .
procedure-heading = "procedure" identifier [ formal-parameter-list ] .
procedure-identification = "procedure" procedure-identifier .
procedure-identifier = identifier .
procedure-block = block .
```

The occurrence of a formal-parameter-list in a procedure-heading of a procedure-declaration shall define the formal parameters of the procedure-block, if any, associated with the identifier of the procedure-heading to be those of the formal-parameter-list.

The occurrence of an identifier in the procedure-heading of a procedure-declaration shall constitute its defining-point as a procedure-identifier for the region that is the block closest-containing the procedure-declaration.

Each identifier having a defining-point as a procedure-identifier in a procedure-heading of a procedure-declaration closest-containing the directive *forward* shall have exactly one of its applied occurrences in a procedure-identification of a procedure-declaration, and that shall be closest-contained by the procedure-and-function-declaration-part closest-containing the procedure-heading.

The occurrence of a procedure-block in a procedure-declaration shall associate the procedure-block with the identifier in the procedure-heading, or with the procedure-identifier in the procedure-identification, of the procedure-declaration.

Examples of a procedure-and-function-declaration-part:

Example 1:

NOTE. This example is not for level 0.

```
procedure AddVectors (var A, B, C : array [low..high : natural] of real);
var
  i : natural;
begin
  for i := low to high do A[i] := B[i] + C[i]
end { of AddVectors };
```

Example 2:

```

procedure readinteger (var f : text; var x : integer);
var
  i : natural;
begin
  while ft = ' ' do get(f);
  {The file buffer contains the first non-space char}
  i := 0;
  while ft in ['0'..'9'] do begin
    i := (10 * i) + (ord(ft) - ord('0'));
    get(f);
  end;
  {The file buffer contains a non-digit}
  x := i
  {Of course if there are no digits, x is zero}
end;

procedure bisection (function f(x : real) : real; a, b : real; var result : real);
{This procedure attempts to find a zero of f(x) in (a,b) by the method of bisection. It
is assumed that the procedure is called with suitable values of a and b such that
f(a) < 0 and f(b) > 0}
The estimate is returned in the last parameter.}
const
  Eps = 1e-10;
var
  midpoint : real;
begin
  {The invariant P is true by calling assumption}
  midpoint := a;
  while abs(a - b) > Eps * abs(a) do begin
    midpoint := (a + b) / 2;
    if f(midpoint) < 0 then a := midpoint
    else b := midpoint
    {Which re-establishes the invariant:
     P = (f(a) < 0) and (f(b) > 0)
     and reduces the interval (a,b) provided that the value of midpoint is
     distinct from both a and b.}
  end;
  {P together with the loop exit condition assures that a zero is contained in
  a small sub-interval. Return the midpoint as the zero.}
  result := midpoint
end;

```

```

procedure PrepareForAppending (var f : FileOfInteger);
{This procedure takes a file in an arbitrary state and sets it up in a
condition for appending data to its end. Simpler conditioning is only
possible if assumptions are made about the initial state of the file.}
var
  LocalCopy : FileOfInteger;

  procedure CopyFiles (var from, into : FileOfInteger);
  begin
    reset(from); rewrite(into);
    while not eof(from) do begin
      intof := fromf;
      put(into); get(from)
    end
  end { of CopyFiles };

begin { of body of PrepareForAppending }
  CopyFiles(f, LocalCopy);
  CopyFiles(LocalCopy, f)
end { of PrepareForAppending };

```

6.6.2 Function-declarations

```

function-declaration = function-heading ":" directive ;
                        function-identification ":" function-block ;
                        function-heading ":" function-block .
function-heading = "function" identifier [ formal-parameter-list ] ":" result-type .
function-identification = "function" function-identifier .
function-identifier = identifier .
result-type = simple-type-identifier ! pointer-type-identifier .
function-block = block .

```

The occurrence of a formal-parameter-list in a function-heading of a function-declaration shall define the formal parameters of the function-block, if any, associated with the identifier of the function-heading to be those of the formal-parameter-list. The function-block shall contain at least one assignment-statement such that the function-identifier of the assignment-statement is associated with the block (see 6.8.2.2).

The occurrence of an identifier in the function-heading of a function-declaration shall constitute its defining-point as a function-identifier associated with the result type denoted by the result-type for the region that is the block closest-containing the function-declaration.

Each identifier having a defining-point as a function-identifier in the function-heading of a function-declaration closest-containing the directive *forward* shall have exactly one of its applied occurrences in a function-identification of a function-declaration, and that shall be closest-contained by the procedure-and-function-declaration-part closest-containing the function-heading.

The occurrence of a function-block in a function-declaration shall associate the function-block with the identifier in the function-heading, or with the function-identifier in the function-identification, of the function-declaration; the block of the function-block shall be associated with the result type that is associated with the identifier or function-identifier, respectively.

Example of a procedure-and-function-declaration-part:

```

function Sqrt (x : real) : real;
{This function computes the square root of x (x > 0) using Newton's method.}
var
  old, estimate : real;
begin
  estimate := x;
  repeat
    old := estimate;
    estimate := (old + x / old) * 0.5;
  until abs(estimate - old) < Eps * estimate;
  {Eps being a global constant}
  Sqrt := estimate
end { of Sqrt };

function max (a : vector) : real;
{This function finds the largest component of the value of a.}
var
  largestsofar : real;
  fence : indextype;
begin
  largestsofar := a[1];
  {Establishes largestsofar = max(a[1])}
  for fence := 2 to limit do begin
    if largestsofar < a[fence] then largestsofar := a[fence]
    {Re-establishing largestsofar = max(a[1], ... ,a[fence])}
  end;
  {So now largestsofar = max(a[1], ... ,a[limit])}
  max := largestsofar
end { of max };

function GCD (m, n : natural) : natural;
begin
  if n=0 then GCD := m else GCD := GCD(n, m mod n);
end;

```

{The following two functions analyse a parenthesized expression and convert it to an internal form. They are declared *forward* since they are mutually recursive, i.e. they call each other.}

```
function ReadExpression : formula;
  forward;
function ReadOperand : formula;
  forward;
```

```
function ReadExpression; {See forward declaration of heading.}
  var
    this : formula;
    op : operation;
  begin
    this := ReadOperand;
    while IsOperator(nextsym) do
      begin
        op := ReadOperator;
        this := MakeFormula(this, op, ReadOperand);
      end;
    ReadExpression := this
  end;
```

```
function ReadOperand; {See forward declaration of heading.}
  begin
    if IsOpenParenthesis(nextsym) then
      begin
        SkipSymbol;
        ReadOperand := ReadExpression;
        {nextsym should be a close-parenthesis}
        SkipSymbol
      end
    else ReadOperand := ReadElement
  end;
```

6.6.3 Parameters

6.6.3.1 General. The identifier-list in a value-parameter-specification shall be a list of value parameters. The identifier-list in a variable-parameter-specification shall be a list of variable parameters.

```
formal-parameter-list = "(" formal-parameter-section
                        { ";" formal-parameter-section } ")" .

formal-parameter-section > value-parameter-specification |
                        variable-parameter-specification |
                        procedural-parameter-specification |
                        functional-parameter-specification .
```

NOTE 1. There is also a syntax rule for formal-parameter-section in 6.6.3.7

```
value-parameter-specification = identifier-list ":" type-identifier .
variable-parameter-specification = "var" identifier-list ":" type-identifier .
procedural-parameter-specification = procedure-heading .
functional-parameter-specification = function-heading .
```

An identifier defined to be a parameter-identifier for the region that is the formal-parameter-list of a procedure-heading shall be designated a formal parameter of the block of the procedure-block, if any, associated with the identifier of the

procedure-heading. An identifier defined to be a parameter-identifier for the region that is the formal-parameter-list of a function-heading shall be designated a formal parameter of the block of the function-block, if any, associated with the identifier of the function-heading.

The occurrence of an identifier in the identifier-list of a value-parameter-specification or a variable-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it and its defining-point as the associated variable-identifier for the region that is the block, if any, of which it is a formal parameter.

The occurrence of the identifier of a procedure-heading in a procedural-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it and its defining-point as the associated procedure-identifier for the region that is the block, if any, of which it is a formal parameter.

The occurrence of the identifier of a function-heading in a functional-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it and its defining-point as the associated function-identifier for the region that is the block, if any, of which it is a formal parameter.

NOTE 2. If the formal-parameter-list is contained in a procedural-parameter-specification or a functional-parameter-specification, there is no corresponding procedure-block or function-block.

6.6.3.2 Value parameters. The formal parameter and its associated variable-identifier shall denote the same variable. The formal parameter shall possess the type denoted by the type-identifier of the value-parameter-specification. The type possessed by a formal parameter shall be one that is permitted as the component-type of a file-type. The actual-parameter (see 6.7.3 and 6.8.2.3) shall be an expression whose value is assignment-compatible with the type possessed by the formal parameter. The current value of the expression shall be attributed upon activation of the block to the variable that is denoted by the formal parameter.

6.6.3.3 Variable parameters. The actual-parameter shall be a variable-access. The type possessed by the actual-parameters shall be the same as that denoted by the type-identifier of the variable-parameter-specification, and the formal parameters shall also possess that type. The actual-parameter shall be accessed before the activation of the block, and this access shall establish a reference to the variable thereby accessed during the entire activation of the block; the corresponding formal parameter and its associated variable-identifier shall denote the referenced variable during the activation.

An actual variable parameter shall not denote a field that is the selector of a variant-part. An actual variable parameter shall not denote a component of a variable where that variable possesses a type that is designated packed.

6.6.3.4 Procedural parameters. The actual-parameter (see 6.7.3 and 6.8.2.3) shall be a procedure-identifier that has a defining-point contained by the program-block. The procedure denoted by the actual-parameter and the procedure denoted by the formal parameter shall have congruous formal-parameter-lists (see 6.6.3.6) if either has a formal-parameter-list. The formal parameter and its associated procedure-identifier shall denote the actual parameter during the entire activation of the block.

6.6.3.5 Functional parameters. The actual-parameter (see 6.7.3 and 6.8.2.3) shall be a function-identifier that has a defining-point contained by the program-block. The function denoted by the actual-parameter and the function denoted by the formal parameter shall have the same result-type and shall have congruous formal-parameter-lists (see 6.6.3.6) if either has a formal-parameter-list. The formal

parameter and its associated function-identifier shall denote the actual parameter during the entire activation of the block.

6.6.3.6 Parameter list congruity. Two formal-parameter-lists shall be congruous if they contain the same number of formal-parameter-sections and if the formal-parameter-sections in corresponding positions match. Two formal-parameter-sections shall match if any of the following statements is true.

(a) They are both value-parameter-specifications containing the same number of parameters and the type-identifier in each value-parameter-specification denotes the same type.

(b) They are both variable-parameter-specifications containing the same number of parameters and the type-identifier in each variable-parameter-specification denotes the same type.

(c) They are both procedural-parameter-specifications and the formal-parameter-lists of the procedure-headings thereof are congruous.

(d) They are both functional-parameter-specifications, the formal-parameter-lists of the function-headings thereof are congruous, and the type-identifiers of the result-types of the function-headings thereof denote the same type.

(e) They are either both value-conformant-array-specifications or both variable-conformant-array-specifications; and in both cases the conformant-array-parameter-specifications contain the same number of parameters and equivalent conformant-array-schemas. Two conformant-array-schemas shall be equivalent if all of the following four statements are true.

(1) There is a single index-type-specification in each conformant-array-schema.

(2) The ordinal-type-identifier in each index-type-specification denotes the same type.

(3) Either the (component) conformant-array-schemas of the conformant-array-schemas are equivalent or the type-identifiers of the conformant-array-schemas denote the same type.

(4) Either both conformant-array-schemas are packed-conformant-array-schemas or both are unpacked-conformant-array-schemas.

NOTES

1. The abbreviated conformant-array-schema and its corresponding full form are equivalent (see 6.6.3.7).

2. For the status of item (e) above see 5.1(a), 5.1(b), 5.1(c), 5.2(a) and 5.2(b).

6.6.3.7 Conformant array parameters

NOTE. For the status of this clause see 5.1(a), 5.1(b), 5.1(c), 5.2(a) and 5.2(b).

6.6.3.7.1 General. The occurrence of an identifier in the identifier-list contained by a conformant-array-parameter-specification shall constitute its defining-point as a parameter-identifier for the region that is the formal-parameter-list closest-containing it and its defining-point as the associated variable-identifier for the region that is the block, if any, of which it is a formal parameter. A variable-identifier so defined shall be designated a *conformant-array-parameter*.

The occurrence of an identifier in an index-type-specification shall constitute its defining-point as a bound-identifier for the region that is the formal-parameter-list

closest-containing it and for the region that is the block, if any, whose formal parameters are specified by that formal-parameter-list.

```

formal-parameter-section > conformant-array-parameter-specification .
conformant-array-parameter-specification = variable-conformant-array-specification ;
                                         value-conformant-array-specification .
value-conformant-array-specification = identifier-list ":" conformant-array-schema .
variable-conformant-array-specification = "var" identifier-list ":"
                                         conformant-array-schema .
conformant-array-schema = packed-conformant-array-schema ;
                          unpacked-conformant-array-schema .
packed-conformant-array-schema = "packed" "array" "[" index-type-specification "]"
                                "of" type-identifier .
unpacked-conformant-array-schema = "array" "[" index-type-specification
                                   { ":" index-type-specification } "]" "of"
                                   ( type-identifier ;
                                     conformant-array-schema ) .
index-type-specification = identifier ".." identifier ":" ordinal-type-identifier .
factor > bound-identifier .
bound-identifier = identifier .

```

NOTE 1. There is also a syntax rule for formal-parameter-section in 6.6.3.1. There is also a syntax rule for factor in 6.7.1.

If a conformant-array-schema closest-contains a conformant-array-schema, then an abbreviated form of definition may be used. In the abbreviated form, a single semicolon shall replace the sequence *I* of array *I* that occurs in the full form. The abbreviated form and the full form shall be equivalent.

Examples:

```

array [u..v : T1] of array [j..k : T2] of T3
array [u..v : T1; j..k : T2] of T3

```

During the entire activation of the block, applied occurrences of the first identifier of an index-type-specification shall denote the smallest value specified by the corresponding index-type (see 6.6.3.8) possessed by each actual-parameter, and applied occurrences of the second identifier of the index-type-specification shall denote the largest value specified by that index-type.

NOTE 2 The object denoted by a bound-identifier is neither a constant nor a variable.

The actual-parameters (see 6.7.3 and 6.8.2.3) corresponding to formal parameters that occur in a single conformant-array-parameter-specification shall all possess the same type. The type possessed by the actual-parameters shall be conformable (see 6.6.3.8) with the conformant-array-schema, and the formal parameters shall possess an array-type that shall be distinct from any other type, and which shall have a component-type that shall be the fixed-component-type of the conformant-array-parameters defined in the conformant-array-parameter-specification and which shall have the index-types of the type possessed by the actual-parameters that correspond (see 6.6.3.8) to the index-type-specifications contained by the conformant-array-schema contained by the conformant-array-parameter-specification. The type denoted by the type-identifier contained by the conformant-array-schema contained by a conformant-array-parameter-specification shall be designated the *fixed-component-type* of the conformant-array-parameters defined by that conformant-array-parameter-specification.

NOTE 3. The type possessed by the formal parameter cannot be a string-type (see 6.4.3.2) because it is not denoted by an array-type.

6.6.3.7.2 Value conformant arrays. The identifier-list in a value-conformant-array-specification shall be a list of value conformant arrays. The actual-parameter shall be an expression. The value of the expression shall be attributed before activation of the block to an auxiliary variable that the program does not otherwise contain. The type possessed by this variable shall be the same as that possessed by the expression. This variable shall be accessed before the activation of the block, and this access shall establish a reference to the variable thereby accessed during the entire activation of the block; the corresponding formal parameter and its associated variable-identifier shall represent the referenced variable during the activation. The fixed-component-type of a value conformant array shall be one that is permitted as the component-type of a file-type.

If the actual-parameter contains an occurrence of a conformant-array-parameter, then for each occurrence of the conformant-array-parameter contained by the actual-parameter, *either*:

- (a) the occurrence of the conformant-array-parameter shall be contained by a function-designator contained by the actual-parameter; or
- (b) the occurrence of the conformant-array-parameter shall be contained by an indexed-variable contained by the actual-parameter, such that the type possessed by that indexed-variable is the fixed-component-type of the conformant-array-parameter.

NOTE. This ensures that the type possessed by the expression and the auxiliary variable will always be known and that, as a consequence, the activation record of a procedure can be of a fixed size.

6.6.3.7.3 Variable conformant arrays. The identifier-list in a variable-conformant-array-specification shall be a list of variable conformant arrays. The actual-parameter shall be a variable-access. The actual-parameter shall be accessed before the activation of the block, and this access shall establish a reference to the variable thereby accessed during the entire activation of the block; the corresponding formal parameter and its associated variable-identifier shall represent the referenced variable during the activation.

An actual-parameter shall not denote a component of a variable where that variable possesses a type that is designated packed.

6.6.3.8 Conformability

NOTE 1. For the status of this clause see 5.1(a), 5.1(b), 5.1(c), 5.2(a) and 5.2(b).

Given a type denoted by an array-type closest-containing a single index-type, and a conformant-array-schema closest-containing a single index-type-specification, then the index-type and the index-type-specification shall be designated as corresponding. Given two conformant-array-schemas closest-containing a single index-type-specification, then the two index-type-specifications shall be designated as corresponding. Let T1 be an array-type with a single index-type and let T2 be the type denoted by the ordinal-type-identifier of the index-type-specification of a conformant-array-schema closest-containing a single index-type-specification, then T1 shall be conformable with the conformant-array-schema if all the following four statements are true.

- (a) The index-type of T1 is compatible with T2.
- (b) The smallest and largest values specified by the index-type of T1 lie within the closed interval specified by T2.
- (c) The component-type of T1 denotes the same type as that denoted by the type-identifier of the conformant-array-schema, or is conformable to the

conformant-array-schema in the conformant-array-schema.

(d) Either T1 is not designated packed and the conformant-array-schema is an unpacked-conformant-array-schema, or T1 is designated packed and the conformant-array-schema is a packed-conformant-array-schema.

NOTE 2. The abbreviated and full forms of a conformant-array-schema are equivalent (see 6.6.3.7). The abbreviated and full forms of an array-type are equivalent (see 6.4.3.2).

It shall be an error if the smallest or largest value specified by the index-type of T1 lies outside the closed interval specified by T2.

6.6.4 Required procedures and functions

6.6.4.1 General. The required procedure-identifiers and function-identifiers and the corresponding required procedures and functions shall be as specified in 6.6.5 and 6.6.6 respectively.

NOTE. Required procedures and functions do not necessarily follow the rules given elsewhere for procedures and functions.

6.6.5 Required procedures

6.6.5.1 General. The required procedures shall be file handling procedures, dynamic allocation procedures and transfer procedures.

6.6.5.2 File handling procedures. Except for the application of *rewrite* or *reset* to the program parameters denoted by *input* or *output*, the effects of applying each of the file handling procedures *rewrite*, *put*, *reset* and *get* to a file-variable *f* shall be defined by pre-assertions and post-assertions about *f*, its components *f.L*, *f.R*, and *f.M*, and about the associated buffer-variable *ft*. The use of the variable *f0* within an assertion shall be considered to represent the state or value, as appropriate, of *f* prior to the operation, while *f* (within an assertion) shall denote the variable after the operation, and similarly for *f0t* and *ft*.

It shall be an error if the stated pre-assertion does not hold immediately prior to any use of the defined operation. It shall be an error if any variable explicitly denoted in an assertion of equality is undefined. The post-assertion shall hold prior to the next subsequent access to the file, its components, or its associated buffer-variable. The post-assertions imply corresponding activities on the external entities, if any, to which the file-variables are bound. These activities, and the point at which they are actually performed, shall be implementation-defined.

rewrite(f) pre-assertion: true.

post-assertion: (*f.L* = *f.R* = S0) and (*f.M* = Generation) and
(*ft* is totally-undefined).

put(f) pre-assertion: (*f0.M* = Generation) and (*f0.L* is not undefined) and
(*f0.R* = S0) and (*f0t* is not undefined).

post-assertion: (*f.M* = Generation) and (*f.L* = (*f0.L*~S(*f0t*))) and
(*f.R* = S0) and (*ft* is totally-undefined).

reset(f) pre-assertion: The components *f0.L* and *f0.R* are not undefined.

post-assertion: (*f.L* = S0) and (*f.R* = (*f0.L*~*f0.R*~X)) and
(*f.M* = Inspection) and (if *f.R* = S0 then
(*ft* is totally-undefined) else (*ft* = *f.R.first*)).

where, if f possesses the type denoted by the required structured-type-identifier $text$ and if $f0.L^*f0.R$ is not empty and if $(f0.L^*f0.R).last$ is not designated an end-of-line, then X shall be a sequence having an end-of-line component as its only component; otherwise $X = S0$.

```
get(f)    pre-assertion: (f0.M = Inspection) and
           (neither f0.L nor f0.R are undefined) and (f0.R  $\diamond$  S0).

           post-assertion: (f.M = Inspection) and (f.L = {f0.L ~ S(f0.R.first)}) and
           (f.R = f0.R.rest) and (if f.R = S0 then
           (f is totally-undefined) else (f = f.R.first)).
```

When the file-variable *f* possesses a type other than that denoted by *text*, the required procedures *read* and *write* shall be defined as follows.

read Read(f, v_1, \dots, v_n) where $v_1 \dots v_n$ denote variable-accesses shall be equivalent to

```
begin read(f,v1); ... ; read(f,vn) end
```

Read(f,v) where v denotes a variable-access shall be equivalent to

```
begin v := ft; get(f) end
```

NOTE 1. The variable-access is not a variable parameter. Consequently it may be a component of a packed structure and the value of the buffer-variable need only be assignment-compatible with it.

write Write(f, e_1, \dots, e_n), where $e_1 \dots e_n$ denote expressions shall be equivalent to

```
begin write(f,e1); ... ; write(f,en) end
```

Write(f,e), where e denotes an expression shall be equivalent to

```
begin ft := e; put(f) end
```

NOTES.

2. The required procedures *read*, *write*, *readln*, *writeln* and *page*, as applied to textfiles, are described in 6.9.

3. Since the definitions of *read* and *write* include the use of *get* and *put*, the implementation-defined aspects of their post-assertions also apply.

6.6.5.3 Dynamic allocation procedures

new(p)

shall create a new variable that is totally-undefined, shall create a new identifying-value of the pointer-type associated with *p*, that identifies the new variable, and shall attribute this identifying-value to the variable denoted by the variable-access *p*. The created variable shall possess the type that is the domain-type of the pointer-type possessed by *p*.

$$\text{new}(p, c1, \dots, cn)$$

shall create a new variable that is totally-undefined, shall create a new identifying-value of the pointer-type associated with p, that identifies the new variable, and shall attribute this identifying-value to the variable denoted by the variable-access p. The created variable shall possess the record-type that is the domain-type of the pointer-type possessed by p and shall have nested variants that correspond to the case-constants c1.....cn. The case-constants shall be listed in order of increasing nesting of the variant-parts. Any variant not specified shall be at a deeper level of nesting than that specified by cn. It shall be an error if a

variant of a variant-part within the new variable becomes active and a different variant of the variant-part is one of the specified variants.

dispose(q)

shall remove the identifying-value denoted by the expression *q* from the pointer-type of *q*. It shall be an error if the identifying-value had been created using the form *new(p,c1,...,cn)*.

dispose(q,k1,...,km)

shall remove the identifying-value denoted by the expression *q* from the pointer-type of *q*. The case-constants *k1,...,km* shall be listed in order of increasing nesting of the variant-parts. It shall be an error if the variable had been created using the form *new(p,c1,...,cn)* and *m* is not equal to *n*. It shall be an error if the variants in the variable identified by the pointer-value of *q* are different from those specified by the case-constants *k1,...,km*.

NOTE. The removal of an identifying-value from the pointer-type to which it belongs renders the identified variable inaccessible (see 6.5.4) and makes undefined all variables and functions that have that value attributed (see 6.6.3.2 and 6.8.2.2).

It shall be an error if *q* has a nil-value or is undefined.

It shall be an error if a variable created using the second form of *new* is accessed by the identified-variable of the variable-access of a factor, of an assignment-statement, or of an actual-parameter.

6.6.5.4 Transfer procedures. Let *a* be a variable possessing a type that can be denoted by

array [*s1*] of *T*.

let *z* be a variable possessing a type that can be denoted by

packed array [*s2*] of *T*.

and *u* and *v* be the smallest and largest values of the type *s2*, then the statement *pack(a,i,z)* shall be equivalent to

```
begin
  k := i;
  for j := u to v do
    begin
      z[j] := a[k];
      if j < v then k := succ(k)
    end
  end
```

and the statement *unpack(z,a,i)* shall be equivalent to

```
begin
  k := i;
  for j := u to v do
    begin
      a[k] := z[j];
      if j < v then k := succ(k)
    end
  end
```

where *j* and *k* denote auxiliary variables that the program does not otherwise contain. The type possessed by *j* shall be *s2*, the type possessed by *k* shall be *s1*, and *i* shall be an expression whose value shall be assignment-compatible with *s1*.

6.6.6 Required functions

6.6.6.1 General. The required functions shall be arithmetic functions, transfer functions, ordinal functions and Boolean functions.

6.6.6.2 Arithmetic functions. For the following arithmetic functions, the expression *x* shall be either of real-type or integer-type. For the functions *abs* and *sqr*, the type of the result shall be the same as the type of the parameter, *x*. For the remaining arithmetic functions, the result shall always be of real-type.

Function	Result
<i>abs</i> (<i>x</i>)	shall compute the absolute value of <i>x</i> .
<i>sqr</i> (<i>x</i>)	shall compute the square of <i>x</i> . It shall be an error if such a value does not exist.
<i>sin</i> (<i>x</i>)	shall compute the sine of <i>x</i> , where <i>x</i> is in radians.
<i>cos</i> (<i>x</i>)	shall compute the cosine of <i>x</i> , where <i>x</i> is in radians.
<i>exp</i> (<i>x</i>)	shall compute the value of the base of natural logarithms raised to the power <i>x</i> .
<i>ln</i> (<i>x</i>)	shall compute the natural logarithm of <i>x</i> , if <i>x</i> is greater than zero. It shall be an error if <i>x</i> is not greater than zero.
<i>sqrt</i> (<i>x</i>)	shall compute the non-negative square root of <i>x</i> , if <i>x</i> is not negative. It shall be an error if <i>x</i> is negative.
<i>arctan</i> (<i>x</i>)	shall compute the principal value, in radians, of the arctangent of <i>x</i> .

6.6.6.3 Transfer functions

trunc(*x*)

From the expression *x* that shall be of real-type, this function shall return a result of integer-type. The value of *trunc*(*x*) shall be such that if *x* is positive or zero then $0 \leq x - \text{trunc}(x) < 1$; otherwise $-1 < x - \text{trunc}(x) \leq 0$. It shall be an error if such a value does not exist.

Examples:

trunc(3.5) yields 3
trunc(-3.5) yields -3

round(*x*)

From the expression *x* that shall be of real-type, this function shall return a result of integer-type. If *x* is positive or zero, *round*(*x*) shall be equivalent to *trunc*(*x*+0.5), otherwise *round*(*x*) shall be equivalent to *trunc*(*x*-0.5). It shall be an error if such a value does not exist.

Examples:

round(3.5) yields 4
round(-3.5) yields -4

6.6.6.4 Ordinal functions

ord(*x*)

From the expression *x* that shall be of an ordinal-type, this function shall return a result of integer-type that shall be the ordinal number (see 6.4.2.2 and 6.4.2.3) of the value of the expression *x*.

chr(*x*)

From the expression *x* that shall be of integer-type, this function shall return a result of char-type that shall be the value whose ordinal number is equal to the value of the expression *x* if such a character value exists. It shall be an error if

such a character value does not exist. For any value, *ch*, of *char*-type, it shall be true that:

$\text{chr}(\text{ord}(\text{ch})) = \text{ch}$

succ(x)

From the expression *x* that shall be of an ordinal-type, this function shall return a result that shall be of the same type as that of the expression (see 6.7.1). The function shall yield a value whose ordinal number is one greater than that of the expression *x*, if such a value exists. It shall be an error if such a value does not exist.

pred(x)

From the expression *x* that shall be of an ordinal-type, this function shall return a result that shall be of the same type as that of the expression (see 6.7.1). The function shall yield a value whose ordinal number is one less than that of the expression *x*, if such a value exists. It shall be an error if such a value does not exist.

6.6.6.5 Boolean functions

odd(x)

From the expression *x* that shall be of integer-type, this function shall be equivalent to the expression
 $(\text{abs}(x) \bmod 2 = 1)$.

eof(f)

The parameter *f* shall be a file-variable; if the actual-parameter-list is omitted, the function shall be applied to the required textfile *input* (see 6.10). When *eof(f)* is activated, it shall be an error if *f* is undefined; otherwise the function shall yield the value true if *f.R* is the empty sequence (see 6.4.3.5), otherwise false.

eoln(f)

The parameter *f* shall be a textfile; if the actual-parameter-list is omitted, the function shall be applied to the required textfile *input* (see 6.10). When *eoln(f)* is activated, it shall be an error if *f* is undefined or if *eof(f)* is true; otherwise the function shall yield the value true if *f.R.first* is an end-of-line component (see 6.4.3.5), otherwise false.

6.7 Expressions

6.7.1 General. An expression shall denote a value unless a variable denoted by a variable-access contained by the expression is undefined at the time of its use, in which case that use shall be an error. The use of a variable-access as a factor shall denote the value, if any, attributed to the variable accessed thereby. Operator precedences shall be according to four classes of operators as follows. The operator *not* shall have the highest precedence, followed by the multiplying-operators, then the adding-operators and signs, and finally, with the lowest precedence, the relational-operators. Sequences of two or more operators of the same precedence shall be left associative.

expression = simple-expression [relational-operator simple-expression] .
 simple-expression = [sign] term { adding-operator term } .
 term = factor { multiplying-operator factor } .
 factor > variable-access ! unsigned-constant ! function-designator !
 set-constructor ! "(" expression ")" ! "not" factor .

NOTE 1. There is also a syntax rule for factor in 6.6.3.7.

unsigned-constant = unsigned-number ! character-string ! constant-identifier !
 "nil" .

set-constructor = "[" [member-designator { "." member-designator }] "]" .
 member-designator = expression [".." expression] .

Any factor whose type is S, where S is a subrange of T, shall be treated as of type T. Similarly, any factor whose type is set of S shall be treated as of the unpacked canonical set-of-T type, and any factor whose type is packed set of S shall be treated as of the canonical packed set-of-T type.

NOTE 2. Consequently, an expression that consists of a single factor of type S is itself of type T, and an expression that consists of a single factor of type set of S is itself of type set of T, and an expression that consists of a single factor of type packed set of S is itself of type packed set of T.

A set-constructor shall denote a value of a set-type. The set-constructor [] shall denote that value in every set-type that contains no members. A set-constructor containing one or more member-designators shall denote either a value of the unpacked canonical set-of-T type or, if the context so requires, the packed canonical set-of-T type, where T is the type of every expression of each member-designator of the set-constructor. The type T shall be an ordinal-type. The value denoted by the set-constructor shall contain zero or more members each of which shall be denoted by at least one member-designator of the set-constructor.

The member-designator x, where x is an expression, shall denote the member that shall have the value x. The member-designator x.y, where x and y are expressions, shall denote zero or more members that shall have the values of the base-type in the closed interval from the value of x to the value of y. The order of evaluation of the expressions of a member-designator shall be implementation-dependent. The order of evaluation of the member-designators of a set-constructor shall be implementation-dependent.

NOTE 3. The member-designator x.y denotes no members if the value of x is greater than the value of y.

Examples:

- (a) *Factors:* x
 15
 (x + y + z)
 sin(x + y)
 [red, c, green]
 [1, 5, 10..19, 23]
 not p
- (b) *Terms:* x * y
 i / (1 - i)
 (x <= y) and (y < z)
- (c) *Simple expressions:* p or q
 x + y
 -x
 hue1 + hue2
 i * j + 1
- (d) *Expressions:* x = 1.5
 p <= q
 p = q and r
 (i < j) = (j < k)
 c in hue1

6.7.2 Operators

6.7.2.1 General

multiplying-operator = "*" | "/" | "div" | "mod" | "and" .

adding-operator = "+" | "-" | "or" .

relational-operator = "=" | "<" | ">" | "<=" | ">=" | "in" .

A factor, or a term, or a simple-expression shall be designated an operand. The order of evaluation of the operands of a dyadic operator shall be implementation-dependent.

NOTE. This means, for example, that the operands may be evaluated in textual order, or in reverse order, or in parallel or they may not both be evaluated.

6.7.2.2 *Arithmetic operators.* The types of operands and results for dyadic and monadic operations shall be as shown in tables 2 and 3 respectively.

Table 2. Dyadic arithmetic operations

Operator	Operation	Type of operands	Type of result
+	addition	integer-type or real-type	integer-type if both operands are of integer-type; otherwise real-type
-	subtraction	integer-type or real-type	
*	multiplication	integer-type or real-type	
/	division	integer-type or real-type	
div	division with truncation	integer-type	integer-type
mod	modulo	integer-type	integer-type

Table 3. Monadic arithmetic operations

Operator	Operation	Type of operand	Type of result
+	identity	integer-type real-type	integer-type real-type
-	sign-inversion	integer-type real-type	integer-type real-type

NOTE 1. The symbols +, - and * are also used as set operators (see 6.7.2.4).

A term of the form x/y shall be an error if y is zero, otherwise the value of x/y shall be the result of dividing x by y .

A term of the form $i \text{ div } j$ shall be an error if j is zero, otherwise the value of $i \text{ div } j$ shall be such that

$\text{abs}(i) - \text{abs}(j) < \text{abs}(i \text{ div } j) * j < \text{abs}(i)$
where the value shall be zero if $\text{abs}(i) < \text{abs}(j)$, otherwise the sign of the value shall be positive if i and j have the same sign and negative if i and j have different signs.

A term of the form $i \text{ mod } j$ shall be an error if j is zero or negative, otherwise the value of $i \text{ mod } j$ shall be that value of $(i - (k*j))$ for integral k such that

$$0 \leq i \text{ mod } j < j.$$

NOTE 2 Only for $i \geq 0$ and $j > 0$ does the relation $(i \text{ div } j) * j + i \text{ mod } j = i$ hold.

The required constant-identifier *maxint* shall denote an implementation-defined value of integer-type. This value shall satisfy the following conditions.

- (a) All integral values in the closed interval from $-maxint$ to $+maxint$ shall be values of the integer-type.
- (b) Any monadic operation performed on an integer value in this interval shall be correctly performed according to the mathematical rules for integer arithmetic.
- (c) Any dyadic integer operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic, provided that the result is also in this interval.
- (d) Any relational operation on two integer values in this same interval shall be correctly performed according to the mathematical rules for integer arithmetic.

The results of the real arithmetic operators and functions shall be approximations to the corresponding mathematical results. The accuracy of this approximation shall be implementation-defined.

It shall be an error if an integer operation or function is not performed according to the mathematical rules for integer arithmetic.

6.7.2.3 Boolean operators. Operands and results for Boolean operations shall be of Boolean-type. Boolean operators *or*, *and* and *not* shall denote respectively the logical operations of disjunction, conjunction and negation.

Boolean-expression = expression .

A Boolean-expression shall be an expression that denotes a value of Boolean-type.

6.7.2.4 Set operators. The types of operands and results for set operations shall be as shown in table 4.

Table 4. Set operations

Operator	Operation	Type of operands	Type of result
+	set union	A canonical set-of-T type (see 6.7.1)	same as the operands
-	set difference		
*	set intersection		

6.7.2.5 Relational-operators. The types of operands and results for relational operations shall be as shown in table 5.

Table 5. Relational operations

Operator	Type of operands	Type of result
= \diamond	any simple, pointer or string-type or canonical set-of-T type	Boolean-type
< >	any simple or string-type	Boolean-type
<= >=	any simple or string-type or canonical set-of-T type	Boolean-type
in	left operand: any ordinal type T right operand: a canonical set-of-T type	Boolean-type

The operands of =, \diamond , <, >, >=, and <= shall be either of compatible types, the same canonical set-of-T type, or one operand shall be of real-type and the other shall be of integer-type.

The operators =, \diamond , <, > shall stand for *equal to*, *not equal to*, *less than* and *greater than* respectively.

Except when applied to sets, the operators <= and >= shall stand for *less than or equal to* and *greater than or equal to* respectively.

Where u and v denote operands of a set-type, u <= v shall denote the inclusion of u in v and u >= v shall denote the inclusion of v in u.

NOTE. Since the Boolean-type is an ordinal-type with false less than true, then if p and q are operands of Boolean-type, p = q denotes their equivalence and p <= q means p implies q.

When the relational operators =, \diamond , <, >, <=, >= are used to compare operands of compatible string-types (see 6.4.3.2), they denote lexicographic relations defined below. Lexicographic ordering imposes a total ordering on values of a string-type. If s1 and s2 are two values of compatible string-types, then

$s1 = s2$ iff for all i in [1..n]: $s1[i] = s2[i]$

$s1 < s2$ iff there exists a p in [1..n]:
(for all i in [1..p-1]: $s1[i] = s2[i]$) and $s1[p] < s2[p]$

The operator *in* shall yield the value true if the value of the operand of ordinal-type is a member of the value of the set-type, otherwise it shall yield the value false.

6.7.3 Function-designators. A function-designator shall specify the activation of the block of the function-block associated with the function-identifier of the function-designator, and shall yield the value of the result of the activation upon completion of the algorithm of the activation; it shall be an error if the result is undefined upon completion of the algorithm. If the function has any formal parameters the function-designator shall contain a list of actual-parameters that shall be bound to their corresponding formal parameters defined in the function-declaration. The correspondence shall be established by the positions of the parameters in the lists of actual and formal parameters respectively. The number of actual-parameters shall be

equal to the number of formal parameters. The types of the actual-parameters shall correspond to the types of the formal parameters as specified by 6.6.3. The order of evaluation, accessing and binding of the actual-parameters shall be implementation-dependent.

```
function-designator = function-identifier [ actual-parameter-list ] .
actual-parameter-list = "(" actual-parameter { "." actual-parameter } ")" .
actual-parameter = expression ! variable-access ! procedure-identifier !
                  function-identifier
```

Examples:

```
Sum(a, 63)
GCD(147, k)
sin(x + y)
eof(f)
ord(ff)
```

6.8 Statements

6.8.1 General. Statements shall denote algorithmic actions, and shall be executable.

NOTE 1. Statements may be prefixed by a label.

A label, if any, of a statement S shall be designated as prefixing S, and shall be allowed to occur in a goto-statement G (see 6.8.2.4) if and only if any of the following three conditions is satisfied.

- (a) S contains G.
- (b) S is a statement of a statement-sequence containing G.
- (c) S is a statement of the statement-sequence of the compound-statement of the statement-part of a block containing G.

```
statement = [ label ":" ] ( simple-statement ! structured-statement ) .
```

NOTE 2. A goto-statement within a block may refer to a label in an enclosing block, provided that the label prefixes a simple-statement or structured-statement at the outermost level of nesting of the block.

6.8.2 Simple-statements

6.8.2.1 General. A simple-statement shall be a statement not containing a statement. An empty-statement shall contain no symbol and shall denote no action.

```
simple-statement = empty-statement ! assignment-statement !
                 procedure-statement ! goto-statement .
empty-statement = .
```

6.8.2.2 Assignment-statements. An assignment-statement shall attribute the value of the expression of the assignment-statement either to the variable denoted by the variable-access of the assignment-statement, or to the activation result that is denoted by the function-identifier of the assignment-statement; the value shall be assignment-compatible with the type possessed, respectively, by the variable or by the activation result. The function-block associated (see 6.6.2) with the function-identifier of an assignment-statement shall contain the assignment-statement.

```
assignment-statement = ( variable-access ! function-identifier ) "!=" expression .
```

The decision as to the order of accessing the variable and evaluating the expression

shall be implementation-dependent; the access shall establish a reference to the variable during the remaining execution of the assignment-statement.

The state of a variable or activation result when the variable or activation result does not have attributed to it a value specified by its type shall be designated *undefined*. If a variable possesses a structured-type, the state of the variable when every component of the variable is totally-undefined shall be designated *totally-undefined*. Totally-undefined shall be synonymous with undefined for an activation result or a variable that does not possess a structured-type.

Examples:

```
x := y + z
p := (1 <= i) and (i < 100)
i := sqrt(k) - (i * j)
hue1 := [blue, succ(c)]
plf.mother := true
```

6.8.2.3 Procedure-statements. A procedure-statement shall specify the activation of the block of the procedure-block associated with the procedure-identifier of the procedure-statement. If the procedure has any formal parameters the procedure-statement shall contain an actual-parameter-list, which is the list of actual-parameters that shall be bound to their corresponding formal parameters defined in the procedure-declaration. The correspondence shall be established by the positions of the parameters in the lists of actual and formal parameters respectively. The number of actual-parameters shall be equal to the number of formal parameters. The types of the actual-parameters shall correspond to the types of the formal parameters as specified by 6.6.3. The order of evaluation, accessing and binding of the actual-parameters shall be implementation-dependent.

The procedure-identifier in a procedure-statement containing a read-parameter-list shall denote the required procedure *read*; the procedure-identifier in a procedure-statement containing a readin-parameter-list shall denote the required procedure *readIn*; the procedure-identifier in a procedure-statement containing a write-parameter-list shall denote the required procedure *write*; the procedure-identifier in a procedure-statement containing a writeln-parameter-list shall denote the required procedure *writeln*.

```
procedure-statement = procedure-identifier ( [ actual-parameter-list ] ;
                                         read-parameter-list ; readin-parameter-list ;
                                         write-parameter-list ; writeln-parameter-list ) .
```

Examples:

```
printheadng
transpose(a, n, m)
bisect(fct, -1.0, +1.0, x)
AddVectors(m[1], m[2], m[k])
```

NOTE. The fourth example is not for level 0.

6.8.2.4 Goto-statements. A goto-statement shall indicate that further processing is to continue at the program-point denoted by the label in the goto-statement and shall cause the termination of all activations except:

- (a) the activation containing the program-point; *and*
- (b) any activation containing the activation-point of an activation required by exceptions (a) or (b) not to be terminated.

```
goto-statement = "goto" label .
```

6.8.3 Structured-statements

6.8.3.1 General

```

structured-statement = compound-statement ; conditional-statement ;
                      ; repetitive-statement ; with-statement .
statement-sequence = statement { ";" statement } .

```

The execution of a statement-sequence shall specify the execution of the statements of the statement-sequence in textual order, except as modified by execution of a goto-statement.

6.8.3.2 Compound-statements. A compound-statement shall specify execution of the statement-sequence of the compound-statement.

compound-statement = "begin" statement-sequence "end" .

Example: `begin z := x ; x := y; y := z end`

6.8.3.3 Conditional-statements

conditional-statement = if-statement ; case-statement .

6.8.3.4 If-statements

```
if-statement = "if" Boolean-expression "then" statement [ else-part ] .
else-part = "else" statement .
```

If the Boolean-expression of the if-statement yields the value true, the statement of the if-statement shall be executed. If the Boolean-expression yields the value false, the statement of the if-statement shall not be executed and the statement of the else-part (if any) shall be executed.

An if-statement without an else-part shall not be immediately followed by the token *e/se*.

NOTE. An else-part is thus paired with the nearest preceding otherwise unpaired *then*.

Examples:

```
if x < 1.5 then z := x + y else z := 1.5
```

```
if p1 <> nil then p1 := p1↑.father
```

```

if j = 0 then
    if i = 0 then writeln('indefinite')
    else writeln('infinite')
else writeln( i / j )

```

6.8.3.5 Case-statements: The values denoted by the case-constants of the case-constant-lists of the case-list-elements of a case-statement shall be distinct and of the same ordinal-type as the expression of the case-index of the case-statement. On execution of the case-statement the case-index shall be evaluated. That value shall then specify execution of the statement of the case-list-element closest-containing the case-constant denoting that value. One of the case-constants shall be equal to the value of the case-index upon entry to the case-statement, otherwise it shall be an error.

NOTE. Case-constants are not the same as statement labels.

```

case-statement = "case" case-index "of" case-list-element
                { ":" case-list-element } [ ":" ] "end" .
case-list-element = case-constant-list ":" statement .
case-index = expression .

```

Example:

```

case operator of
  plus:   x := x + y;
  minus:  x := x - y;
  times:  x := x * y
end

```

6.8.3.6 Repetitive-statements. Repetitive-statements shall specify that certain statements are to be executed repeatedly.

```

repetitive-statement = repeat-statement | while-statement | for-statement .

```

6.8.3.7 Repeat-statements

```

repeat-statement = "repeat" statement-sequence "until" Boolean-expression .

```

The statement-sequence of the repeat-statement shall be repeatedly executed (except as modified by the execution of a goto-statement) until the Boolean-expression of the repeat-statement yields the value true on completion of the statement-sequence. The statement-sequence shall be executed at least once, because the Boolean-expression is evaluated after execution of the statement-sequence.

Example:

```

repeat k := i mod j;
  i := j;
  j := k
until j = 0

```

6.8.3.8 While-statements

```

while-statement = "while" Boolean-expression "do" statement .

```

The while-statement

```

while b do body

```

shall be equivalent to

```

begin
  if b then
    repeat
      body
    until not (b)
  end
end

```

Examples:

```

while i > 0 do
  begin if odd(i) then z := z * x;
        i := i div 2;
        x := sqrt(x)
      end
end

```

```

while not eof(f) do
  begin process(ff); get(f)
end

```

6.8.3.9 For-statements. The for-statement shall specify that the statement of the for-statement is to be repeatedly executed while a progression of values is attributed to a variable that is designated the control-variable of the for-statement.

```

for-statement = "for" control-variable ":=" initial-value ( "to" | "downto" )
               final-value "do" statement .
control-variable = entire-variable .
initial-value = expression .
final-value = expression .

```

The control-variable shall be an entire-variable whose identifier is declared in the variable-declaration-part of the block closest-containing the for-statement. The control-variable shall possess an ordinal-type, and the initial-value and final-value shall be of a type compatible with this type. The initial-value and the final-value shall be assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed. After a for-statement is executed (other than being left by a goto-statement leading out of it) the control-variable shall be undefined. Neither a for-statement nor any procedure-and-function-declaration-part of the block that closest-contains a for-statement shall contain a statement threatening the variable denoted by the control-variable of the for-statement.

A statement S shall be designated as *threatening* a variable V if one or more of the following statements is true.

- (a) S is an assignment-statement and V is denoted by the variable-access of S.
- (b) S contains an actual variable parameter that denotes V.
- (c) S is a procedure-statement that specifies the activation of the required procedure *read* or the required procedure *readln*, and V is denoted by an actual parameter contained by S.
- (d) S is a for-statement and the control-variable of S denotes V.

Apart from the restrictions imposed by these requirements, the for-statement

```
for v := e1 to e2 do body
```

shall be equivalent to

```

begin
temp1 := e1;
temp2 := e2;
if temp1 <= temp2 then
begin
v := temp1;
body;
while v <> temp2 do
begin
v := succ(v);
body
end
end
end
end

```

and the for-statement

```
for v := e1 downto e2 do body
```

shall be equivalent to

```

begin
temp1 := e1;
temp2 := e2;
if temp1 >= temp2 then
begin
v := temp1;
body;
while v <> temp2 do
begin
v := pred(v);
body
end
end
end
end

```

where temp1 and temp2 denote auxiliary variables that the program does not otherwise contain, and that possess the type possessed by the variable v if that type is not a subrange-type; otherwise the host type of the type possessed by the variable v.

Examples:

```

for i := 2 to 63 do
if a[i] > max then max := a[i]

for i := 1 to 10 do
for j := 1 to 10 do
begin
x := 0;
for k := 1 to 10 do
x := x + m1[i,k] * m2[k,j];
m[i,j] := x
end

for i:= 1 to 10 do
for j := 1 to i - 1 do
m[i][j] := 0.0

```



```
for c := blue downto red do q(c)
```

6.8.3.10 With-statements

```
with-statement = "with" record-variable-list "do" statement .
record-variable-list = record-variable { "," record-variable } .
field-designator-identifier = identifier .
```

A with-statement shall specify the execution of the statement of the with-statement. The occurrence of a record-variable as the only record-variable in the record-variable-list of a with-statement shall constitute a defining-point of each of the field-identifiers associated with components of the record-type possessed by the record-variable as a field-designator-identifier for the region that is the statement of the with-statement; each applied occurrence of a field-designator-identifier shall denote that component of the record-variable that is associated with the field-identifier by the record-type. The record-variable shall be accessed before the statement of the with-statement is executed, and that access shall establish a reference to the variable during the entire execution of the statement of the with-statement.

The statement

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

shall be equivalent to

```
with v1 do
  with v2 do
    ...
    with vn do s
```

Example:

```
with date do
  if month = 12 then
    begin month := 1; year := year + 1
  end
else month := month+1
```

has the same effect on the variable *date* as

```
if date.month = 12 then
  begin date.month := 1; date.year := date.year+1
end
else date.month := date.month+1
```

6.9 Input and output

6.9.1 The procedure read. The syntax of the parameter list of *read* when applied to a textfile shall be:

```
read-parameter-list = "(" [ file-variable "," ] variable-access
                      { "," variable-access } ")" .
```

If the file-variable is omitted, the procedure shall be applied to the required textfile *input*.

The following requirements shall apply for the procedure *read* (where *f* denotes a textfile and *v1...vn* denote variable-accesses possessing the char-type (or a subrange of char-type), the integer-type (or a subrange of integer-type), or the real-type).

(a) `read(f,v1,...,vn)` shall be equivalent to

```
begin read(f,v1); ... ; read(f,vn) end
```

(b) If `v` is a variable-access possessing the char-type (or subrange thereof), `read(f,v)` shall be equivalent to

```
begin v := ft; get(f) end
```

NOTE. The variable-access is not a variable parameter. Consequently it may be a component of a packed structure and the value of the buffer-variable need only be assignment-compatible with it.

(c) If `v` is a variable-access possessing the integer-type (or subrange thereof), `read(f,v)` shall cause the reading from `f` of a sequence of characters. Preceding spaces and end-of-lines shall be skipped. It shall be an error if the rest of the sequence does not form a signed-integer according to the syntax of 6.1.5. Reading shall cease as soon as the buffer-variable `ft` does not have attributed to it a character contained by the signed-integer. The value of the signed-integer thus read shall be assignment-compatible with the type possessed by `v`, and shall be attributed to `v`.

(d) If `v` is a variable-access possessing the real-type, `read(f,v)` shall cause the reading from `f` of a sequence of characters. Preceding spaces and end-of-lines shall be skipped. It shall be an error if the rest of the sequence does not form a signed-number according to the syntax of 6.1.5. Reading shall cease as soon as the buffer-variable `ft` does not have attributed to it a character contained by the signed-number. The value denoted by the number thus read shall be attributed to the variable `v`.

(e) When `read` is applied to `f`, it shall be an error if the buffer-variable `ft` is undefined or the pre-assertions for `get` do not hold (see 6.4.3.5).

6.9.2 The procedure `readln`. The syntax of the parameter list of `readln` shall be:

```
readln-parameter-list = [ "(" [ file-variable ; variable-access ]
                          { "," variable-access } ")" ] .
```

Readln shall only be applied to textfiles. If the file-variable or the entire *readln-parameter-list* is omitted, the procedure shall be applied to the required textfile *input*.

`Readln(f,v1,...,vn)` shall be equivalent to

```
begin read(f,v1,...,vn); readln(f) end
```

`Readln(f)` shall be equivalent to

```
begin while not eoln(f) do get(f); get(f) end
```

NOTE. The effect of *readln* is to place the current file position just past the end of the current line in the textfile. Unless this is the end-of-file position, the current file position is therefore at the start of the next line.

6.9.3 The procedure `write`. The syntax of the parameter list of `write` when applied to a textfile shall be:

```
write-parameter-list = "(" [ file-variable "," ] write-parameter
                      { "," write-parameter } ")" .
write-parameter = expression [ ":" expression [ ":" expression ] ] .
```

If the file-variable is omitted, the procedure shall be applied to the required textfile

output. When *write* is applied to a textfile *f*, it shall be an error if *f* is undefined or *f.M* = inspection (see 6.4.3.5). An application of *write* to a textfile *f* shall cause the buffer-variable *ft* to become undefined.

Write(f,p1,...,pn) shall be equivalent to

```
begin write(f,p1); ... ; write(f,pn) end
```

where *f* denotes a textfile, and *p1,...,pn* denote write-parameters.

6.9.3.1 Write-parameters. The write-parameters *p* shall have the following forms:

```
e : TotalWidth : FracDigits
e : TotalWidth
e
```

where *e* is an expression whose value is to be written on the file *f* and may be of integer-type, real-type, char-type, Boolean-type or a string-type, and where *TotalWidth* and *FracDigits* are expressions of integer-type whose values are the field-width parameters. The values of *TotalWidth* and *FracDigits* shall be greater than or equal to one; it shall be an error if either value is less than one.

Write(f,e) shall be equivalent to the form *write(f,e : TotalWidth)*, using a default value for *TotalWidth* that depends on the type of *e*; for integer-type, real-type and Boolean-type the default values shall be implementation-defined.

Write(f,e : TotalWidth : FracDigits) shall be applicable only if *e* is of real-type (see 6.9.3.4.2).

6.9.3.2 Char-type. If *e* is of char-type, the default value of *TotalWidth* shall be one. The representation written on the file *f* shall be:

(*TotalWidth* - 1) spaces, the character value of *e*.

6.9.3.3 Integer-type. If *e* is of integer-type, the decimal representation of *e* shall be written on the file *f*. Assume a function

```
function IntegerSize ( x : Integer ) : Integer ;
{ returns the number of digits, z, such that
  10 to the power (z-1) ≤ abs(x) < 10 to the power z }
```

and let *IntDigits* be the positive integer defined by:

```
if e = 0
then IntDigits := 1
else IntDigits := IntegerSize(e);
```

then the representation shall consist of:

- (a) if *TotalWidth* ≥ *IntDigits* + 1:
(*TotalWidth* - *IntDigits* - 1) spaces.
the sign character: '-' if *e* < 0, otherwise a space.
IntDigits digit-characters of the decimal representation of *abs(e)*.
- (b) if *TotalWidth* < *IntDigits* + 1:
if *e* < 0 the sign character '-'.
IntDigits digit-characters of the decimal representation of *abs(e)*.

6.9.3.4 Real-Type. If *e* is of real-type, a decimal representation of the number *e*, rounded to the specified number of significant figures or decimal places, shall be

written on the file *f*.

6.9.3.4.1 The floating-point representation. Write(*f*,*e* : TotalWidth) shall cause a floating-point representation of *e* to be written. Assume functions

```
function TenPower ( Int : integer ) : real ;
{ Returns 10.0 raised to the power Int }

function RealSize ( y : real ) : integer ;
{ Returns the value, z, such that TenPower(z-1) ≤ abs(y) < TenPower(z) }

function Truncate ( y : real ; DecPlaces : integer ) : real ;
{ Returns the value of y after truncation to DecPlaces decimal places }
```

let *ExpDigits* be an implementation-defined value representing the number of digit-characters written in an exponent;

let *ActWidth* be the positive integer defined by:

```
if TotalWidth ≥ ExpDigits + 6
then ActWidth := TotalWidth
else ActWidth := ExpDigits + 6;
```

and let the non-negative number *eWritten*, the positive integer *DecPlaces* and the integer *ExpValue* be defined by:

```
DecPlaces := ActWidth - ExpDigits - 5;
if e = 0.0
then begin eWritten := 0.0; ExpValue := 0 end
else
begin
eWritten := abs(e);
ExpValue := RealSize ( eWritten ) - 1;
eWritten := eWritten / TenPower ( ExpValue );
eWritten := eWritten + 0.5 * TenPower ( -DecPlaces );
if eWritten ≥ 10.0
then
begin
eWritten := eWritten / 10.0;
ExpValue := ExpValue + 1
end;
eWritten := Truncate ( eWritten, DecPlaces )
end;
```

then the floating-point representation of the value of *e* shall consist of:

- the sign character ('-' if (*e* < 0) and (*eWritten* > 0), otherwise a space),
- the leading digit-character of the decimal representation of *eWritten*,
- the character '.' ,
- the next *DecPlaces* digit-characters of the decimal representation of *eWritten*,
- an implementation-defined exponent character (either 'e' or 'E'),
- the sign of *ExpValue* ('-' if *ExpValue* < 0, otherwise '+'),
- the *ExpDigits* digit-characters of the decimal representation of *ExpValue* (with leading zeros if the value requires them).

6.9.3.4.2 The fixed-point representation. Write(*f*,*e* : TotalWidth : FracDigits) shall cause a fixed-point representation of *e* to be written. Assume the functions *TenPower* and *Truncate* described in 6.9.3.4.1:

let *eWritten* be the non-negative number defined by:

```

if e = 0.0
then eWritten := 0.0
else
begin
eWritten := abs(e);
eWritten := eWritten + 0.5 * TenPower ( - FracDigits );
eWritten := Truncate ( eWritten, FracDigits )
end;

```

let *IntDigits* be the positive integer defined by:

```

if RealSize ( eWritten ) < 1
then IntDigits := 1
else IntDigits := RealSize ( eWritten );

```

and let *MinNumChars* be the positive integer defined by:

```

MinNumChars := IntDigits + FracDigits + 1;
if (e < 0.0) and (eWritten > 0)
then MinNumChars := MinNumChars + 1; { '-' required }

```

then the fixed-point representation of the value of *e* shall consist of:

```

if TotalWidth ≥ MinNumChars, (TotalWidth - MinNumChars) spaces,
the character '-' if (e < 0) and (eWritten > 0),
the first IntDigits digit-characters of the decimal representation of the
value of eWritten,
the character '.',
the next FracDigits digit-characters of the decimal representation
of the value of eWritten.

```

NOTE. At least *MinNumChars* characters are written. If *TotalWidth* is less than this value, no initial spaces are written.

6.9.3.5 Boolean-types. If *e* is of Boolean-type, a representation of the word true or the word false (as appropriate to the value of *e*) shall be written on the file *f*. This shall be equivalent to writing the appropriate character-strings 'True' or 'False' (see 6.9.3.6), where the case of each letter is implementation-defined, with a field-width parameter of *TotalWidth*.

6.9.3.6 String-types. If the type of *e* is a string-type with *n* components, the default value of *TotalWidth* shall be *n*. The representation shall consist of:

```

if TotalWidth > n,
(TotalWidth - n) spaces,
the first through nth characters of the value of e in that order.

if 1 ≤ TotalWidth ≤ n,
the first through TotalWidthth characters in that order.

```

6.9.4 The procedure *writeln*. The syntax of the parameter list of *writeln* shall be:

```

writeln-parameter-list = [ "(" ( file-variable ! write-parameter )
{ "," write-parameter } ")" ] .

```

Writeln shall only be applied to textfiles. If the file-variable or the *writeln-parameter-list* is omitted, the procedure shall be applied to the required textfile output.

Writeln(f,p1.....pn) shall be equivalent to

```
begin write(f.p1.....pn); writeln(f) end
```

Writeln shall be defined by a pre-assertion and a post-assertion using the notation of 6.6.5.2.

pre-assertion: (f0 is not undefined) and (f0.M = Generation) and
(f0.R = S0).

post-assertion: (f.L = (f0.L~S(e))) and (ft is totally-undefined) and
(f.R = S0) and (f.M = Generation),
where S(e) is the sequence consisting solely of the
end-of-line component defined in 6.4.3.5.

NOTE. *Writeln*(f) terminates the partial line, if any, which is being generated. By the conventions of 6.6.5.2 it is an error if the pre-assertion is not true prior to *writeln*(f).

6.9.5 The procedure page. It shall be an error if the pre-assertion required for *writeln*(f) (see 6.9.4) does not hold prior to the activation of *page*(f). If the actual-parameter-list is omitted the procedure shall be applied to the required textfile *output*. *Page*(f) shall cause an implementation-defined effect on the textfile f, such that subsequent text written to f will be on a new page if the textfile is printed on a suitable device, shall perform an implicit *writeln*(f) if f.L is not empty and if f.L.last is not the end-of-line component (see 6.4.3.5), and shall cause the buffer-variable ft to become totally-undefined. The effect of inspecting a textfile to which the *page* procedure was applied during generation shall be implementation-dependent.

6.10 Programs

```
program = program-heading ";" program-block "." .
program-heading = "program" identifier [ "(" program-parameters ")" ] .
program-parameters = identifier-list .
program-block = block .
```

The identifier of the program-heading shall be the program name that shall have no significance within the program. The identifiers contained by the program-parameters shall be distinct and shall be designated program parameters. Each program parameter shall have a defining-point as a variable-identifier for the region that is the program-block. The binding of the variables denoted by the program parameters to entities external to the program shall be implementation-dependent, except if the variable possesses a file-type in which case the binding shall be implementation-defined.

NOTE. The external representation of such external entities is not defined by this standard, nor is any property of a Pascal program dependent on such representation.

The occurrence of the required identifier *input* or the required identifier *output* as a program parameter shall constitute its defining-point for the region that is the program-block as a variable-identifier of the required type denoted by the required type-identifier text. Such occurrence of the identifier *input* shall cause the post-assertions of *reset* to hold, and of *output*, the post-assertions of *rewrite* to hold, prior to the first access to the textfile or its associated buffer-variable. The effect of the application of the required procedure *reset* or the required procedure *rewrite* to either of these textfiles shall be implementation-defined.

Examples:

```
program copy (f, g);  
var f, g : file of real;  
begin reset(f); rewrite(g);  
  while not eof(f) do  
    begin gt := ft; get(f); put(g)  
  end  
end.
```

```
program copytext (input, output);  
{This program copies the characters and line structure of the textfile input to the  
textfile output.}  
var ch : char;  
begin  
  while not eof do  
    begin  
      while not eoln do  
        begin read(ch); write(ch)  
      end;  
      readln; writeln  
    end  
  end  
end.
```

```

program t6p6p3p4 (output);
var globalone, globaltwo : integer;

procedure dummy;
begin
  writeln('fail4')
end { of dummy };

procedure p (procedure f(procedure ff; procedure gg); procedure g);
var localtop : integer;
procedure r;
begin
  if globalone = 1 then
    begin
      if (globaltwo < 2) or (localtop < 1) then
        writeln('fail1')
      end
    else if globalone = 2 then
      begin
        if (globaltwo < 2) or (localtop < 2) then
          writeln('fail2')
        else
          writeln('pass')
        end
      else
        writeln('fail3');
        globalone := globalone + 1
      end { of r };
    begin { of p }
      globaltwo := globaltwo + 1;
      localtop := globaltwo;
      if globaltwo = 1 then
        p(f, r)
      else
        f(g, r)
      end { of p };
    end { of p };

procedure q (procedure f; procedure g);
begin
  f;
  g
end { of q };

begin
  globalone := 1;
  globaltwo := 0;
  p(q, dummy)
end.

```


Appendix A

Collected syntax

The non-terminal symbols *pointer-type*, *program*, *signed-number*, *simple-type*, *special-symbol* and *structured-type* are only referenced by the semantics and are not used in the right-hand-side of any production. The non-terminal symbol *program* is the start symbol of the grammar.

```

actual-parameter = expression | variable-access | procedure-identifier |
                  function-identifier .

actual-parameter-list = "(" actual-parameter { "," actual-parameter } ")" .

adding-operator = "+" | "-" | "or" .

apostrophe-image = "'" .

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

array-variable = variable-access .

assignment-statement = ( variable-access | function-identifier ) "!=" expression .

base-type = ordinal-type .

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

Boolean-expression = expression .

bound-identifier = identifier .

buffer-variable = file-variable "†" .

case-constant = constant .

case-constant-list = case-constant { "," case-constant } .

case-index = expression .

case-list-element = case-constant-list ":" statement .

case-statement = "case" case-index "of" case-list-element
                { "," case-list-element } [ ";" ] "end" .

character-string = "" string-element { string-element } "" .

component-type = type-denoter .

component-variable = indexed-variable | field-designator .

compound-statement = "begin" statement-sequence "end" .

conditional-statement = if-statement | case-statement .

conformant-array-parameter-specification = value-conformant-array-specification |
                                           variable-conformant-array-specification .

```

[illegible]

```

formal-parameter-section = value-parameter-specification ;
                           variable-parameter-specification ;
                           procedural-parameter-specification ;
                           functional-parameter-specification ;
                           conformant-array-parameter-specification .

fractional-part = digit-sequence .

function-block = block .

function-declaration = function-heading ":" directive ;
                     function-identification ":" function-block ;
                     function-heading ":" function-block .

function-designator = function-identifier [ actual-parameter-list ] .

function-heading = "function" identifier [ formal-parameter-list ] ":" result-type .

function-identification = "function" function-identifier .

function-identifier = identifier .

functional-parameter-specification = function-heading .

goto-statement = "goto" label .

identified-variable = pointer-variable "*" .

identifier = letter { letter ! digit } .

identifier-list = identifier { "," identifier } .

if-statement = "if" Boolean-expression "then" statement [ else-part ] .

index-expression = expression .

index-type = ordinal-type .

index-type-specification = identifier ".." identifier ":" ordinal-type-identifier .

indexed-variable = array-variable "[" index-expression { "," index-expression } "]" .

initial-value = expression .

label = digit-sequence .

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

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" .

member-designator = expression [ ".." expression ] .

multiplying-operator = "*" | "/" | "div" | "mod" | "and" .

new-ordinal-type = enumerated-type | subrange-type .

new-pointer-type = "*" domain-type .

```



```

record-variable-list = record-variable { "," record-variable } .
relational-operator = "=" | "<" | "<=" | ">" | ">=" | "in" .
repeat-statement = "repeat" statement-sequence "until" Boolean-expression .
repetitive-statement = repeat-statement | while-statement | for-statement .
result-type = simple-type-identifier | pointer-type-identifier .
scale-factor = signed-integer .
set-constructor = "[" [ member-designator { "," member-designator } ] "]" .
set-type = "set" "of" base-type .
sign = "+" | "-" .
signed-integer = [ sign ] unsigned-integer .
signed-number = signed-integer | signed-real .
signed-real = [ sign ] unsigned-real .
simple-expression = [ sign ] term { adding-operator term } .
simple-statement = empty-statement | assignment-statement | procedure-statement |
    goto-statement .
simple-type = ordinal-type | real-type-identifier .
simple-type-identifier = type-identifier .
special-symbol = "+" | "-" | "*" | "/" | "=" | "<" | ">" | "[" | "]" |
    "." | "," | ":" | ";" | "\t" | "(" | ")" |
    "<=" | ">=" | "!=" | ".." | word-symbol .
statement = [ label ":" ] ( simple-statement | structured-statement ) .
statement-part = compound-statement .
statement-sequence = statement { ";" statement } .
string-character = one-of-a-set-of-implementation-defined-characters .
string-element = apostrophe-image | string-character .
structured-statement = compound-statement | conditional-statement |
    repetitive-statement | with-statement .
structured-type = new-structured-type | structured-type-identifier .
structured-type-identifier = type-identifier .
subrange-type = constant ".." constant .
tag-field = identifier .
tag-type = ordinal-type-identifier .

```

```

term = factor { multiplying-operator factor } .

type-definition = identifier "=" type-denoter .

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

type-denoter = type-identifier | new-type .

type-identifier = identifier .

unpacked-conformant-array-schema = "array" "[" index-type-specification
                                   { ":" index-type-specification } "]" "of"
                                   ( type-identifier |
                                     conformant-array-schema ) .

unpacked-structured-type = array-type | record-type | set-type | file-type .

unsigned-constant = unsigned-number | character-string | constant-identifier |
                  "nil" .

unsigned-integer = digit-sequence .

unsigned-number = unsigned-integer | unsigned-real .

unsigned-real = unsigned-integer "." fractional-part [ "e" scale-factor ] |
              unsigned-integer "e" scale-factor .

value-conformant-array-specification = identifier-list ":" conformant-array-schema .

value-parameter-specification = identifier-list ":" type-identifier .

variable-access = entire-variable | component-variable | identified-variable |
                buffer-variable .

variable-conformant-array-specification = "var" identifier-list ":"
                                         conformant-array-schema .

variable-declaration = identifier-list ":" type-denoter .

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

variable-identifier = identifier .

variable-parameter-specification = "var" identifier-list ":" type-identifier .

variant = case-constant-list ":" "(" field-list ")" .

variant-part = "case" variant-selector "of" variant { ":" variant } .

variant-selector = [ tag-field ":" ] tag-type .

while-statement = "while" Boolean-expression "do" statement .

with-statement = "with" record-variable-list "do" statement .

```

```
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" .
```

```
write-parameter = expression [ ":" expression [ ":" expression ] ] .
```

```
write-parameter-list = "(" [ file-variable "," ] write-parameter  
                      { "," write-parameter } ")" .
```

```
writeln-parameter-list = [ "(" ( file-variable | write-parameter )  
                          { "," write-parameter } ")" ] .
```

Appendix B

Index

access	6.5.1	6.5.3.1	6.5.3.3
	6.5.5	6.6.3.3	6.6.3.7.2
	6.6.3.7.3	6.6.5.2	6.8.2.2
	6.8.3.10	6.10	
actual	6.6.3.3	6.6.3.4	6.6.3.5
	6.7.3	6.8.2.3	6.8.3.9
	6.6.3.2	6.6.3.3	6.6.3.4
actual-parameter	6.6.3.5	6.6.3.7.1	6.6.3.7.2
	6.6.3.7.3	6.6.5.3	6.7.3
	6.6.6.5	6.7.3	6.8.2.3
actual-parameter-list	6.9.5		
	6.4.3.1	6.4.3.2	6.5.3.2
	6.6.3.7.1	6.6.3.8	
assignment-compatible	6.4.6	6.5.3.2	6.6.3.2
	6.6.5.2	6.6.5.4	6.8.2.2
	6.8.3.9	6.9.1	
assignment-statement	6.2.3.3	6.6.2	6.6.5.3
	6.8.2.1	6.8.2.2	6.8.3.9
	6.4.3.4	6.4.5	6.4.6
base-type	6.7.1		
	6.2.1	6.2.3.1	6.2.3.2
	6.2.3.3	6.2.3.4	6.3
block	6.4.1	6.4.2.3	6.5.1
	6.6.1	6.6.2	6.6.3.1
	6.6.3.2	6.6.3.3	6.6.3.4
	6.6.3.5	6.6.3.7.1	6.6.3.7.2
	6.6.3.7.3	6.7.3	6.8.1
	6.8.2.3	6.8.3.9	6.10
body	6.6.1	6.8.3.8	6.8.3.9
	6.7.2.3	6.8.3.4	6.8.3.7
	6.8.3.8		
boolean-expression	6.4.2.2	6.7.2.3	6.7.2.5
	6.9.3.1	6.9.3.5	
	6.5.1	6.5.5	6.6.5.2
boolean-type	6.9.1	6.9.3	6.9.5
	6.10		
	6.4.3.3	6.6.5.3	6.8.3.5
case-constants	6.1.7	6.4.2.2	6.4.3.2
	6.4.3.5	6.5.5	6.6.6.4
	6.9.1	6.9.3.1	6.9.3.2
char-type	6.1.7	6.1.9	6.4.2.2
	6.6.6.4	6.9.1	6.9.3.2
	6.9.3.3	6.9.3.4.1	6.9.3.4.2
character	6.1.1	6.1.7	6.1.8
	6.3	6.4.3.2	6.7.1
	6.1.5	6.1.6	6.4.6
character-string	6.6.3.8	6.7.1	6.7.2.2
	6.4.3.3	6.4.5	6.4.6
	6.4.7	6.6.3.8	6.7.2.5
closed	6.8.3.9		
compatible			

component	6.4.3.1	6.4.3.2	6.4.3.3
	6.4.3.5	6.5.1	6.5.3.1
	6.5.3.2	6.5.3.3	6.6.2
	6.6.3.3	6.6.3.6	6.6.3.7.3
	6.6.5.2	6.6.6.5	6.8.2.2
	6.8.3.10	6.9.1	6.9.4
component-type	6.9.5		
	6.4.3.2	6.4.3.5	6.4.6
	6.5.5	6.6.3.2	6.6.3.7.1
components	6.6.3.8		
	6.1.7	6.4.3.1	6.4.3.2
	6.4.3.3	6.4.3.5	6.4.5
	6.5.3.3	6.6.5.2	6.8.3.10
compound-statement	6.9.3.6		
	6.2.1	6.8.1	6.8.3.1
	6.8.3.2		
conformant-array-schema	6.6.3.6	6.6.3.7.1	6.6.3.8
congruous	6.6.3.4	6.6.3.5	6.6.3.6
constant	6.3	6.4.2.4	6.4.3.3
corresponding	6.6.2	6.6.3.7.1	
	1.2	4	6.1.4
	6.1.9	6.2.3.2	6.2.3.3
	6.4.1	6.4.3.3	6.5.4
	6.6.3.1	6.6.3.3	6.6.3.6
	6.6.3.7.1	6.6.3.7.2	6.6.3.7.3
	6.6.3.8	6.6.4.1	6.6.5.2
	6.7.2.2	6.7.3	6.8.2.3
defining-point	6.2.1	6.2.2.1	6.2.2.2
	6.2.2.3	6.2.2.4	6.2.2.5
	6.2.2.7	6.2.2.8	6.2.2.9
	6.2.2.11	6.2.3.1	6.2.3.2
	6.3	6.4.1	6.4.2.3
	6.4.3.3	6.5.1	6.5.3.3
	6.6.1	6.6.2	6.6.3.1
	6.6.3.4	6.6.3.5	6.6.3.7.1
	6.8.3.10	6.10	
	3.1	4	5.1
	6.4.3.5	6.6.3.7.1	
directive	6.1.4	6.6.1	6.6.2
entire-variable	6.5.1	6.5.2	6.8.3.9
enumerated-type	6.4.2.1	6.4.2.3	
error	3.1	3.2	5.1
	6.4.6	6.5.3.3	6.5.4
	6.5.5	6.6.3.8	6.6.5.2
	6.6.5.3	6.6.6.2	6.6.6.3
	6.6.6.4	6.6.6.5	6.7.1
	6.7.2.2	6.7.3	6.8.3.5
	6.9.1	6.9.3	6.9.3.1
	6.9.4	6.9.5	
	6.5.3.2	6.6.2	6.6.3.2
	6.6.3.7.2	6.6.5.2	6.6.5.3
expression	6.6.5.4	6.6.6.2	6.6.6.3
	6.6.6.4	6.6.6.5	6.7.1
	6.7.2.3	6.7.3	6.8.2.2
	6.8.3.5	6.8.3.9	6.9.3
	6.9.3.1		
factor	6.1.5	6.6.3.7.1	6.6.5.3
	6.7.1	6.7.2.1	
field	6.4.3.3	6.5.3.3	6.6.3.3
field-designator	6.2.2.6	6.5.3.1	6.5.3.3

field-identifier	6.4.3.3	6.5.3.3	6.8.3.10
file-type	6.4.3.1	6.4.3.5	6.4.6
	6.5.5	6.6.3.2	6.10
file-variable	6.5.5	6.6.5.2	6.6.6.5
	6.9.1	6.9.2	6.9.3
	6.9.4		
formal	6.2.3.2	6.6.1	6.6.2
	6.6.3.1	6.6.3.2	6.6.3.3
	6.6.3.4	6.6.3.5	6.6.3.7.1
	6.6.3.7.2	6.6.3.7.3	6.7.3
	6.8.2.3		
formal-parameter-list	6.6.1	6.6.2	6.6.3.1
	6.6.3.4	6.6.3.5	6.6.3.7.1
function	6.1.2	6.2.3.2	6.2.3.3
	6.4.3.5	6.6	6.6.1
	6.6.2	6.6.3.5	6.6.6.3
	6.6.6.4	6.6.6.5	6.7.2.2
	6.7.3	6.9.3.3	6.9.3.4.1
	6.9.3.4.2		
function-block	6.1.4	6.2.3.2	6.2.3.3
	6.6.2	6.6.3.1	6.8.2.2
function-declaration	6.1.4	6.2.1	6.6.2
	6.7.3		
function-designator	6.2.3.4	6.6.3.7.2	6.7.1
	6.7.3		
function-identifier	6.2.3.1	6.2.3.2	6.2.3.3
	6.6.2	6.6.3.1	6.6.3.5
	6.7.3	6.8.2.2	
goto-statement	6.8.1	6.8.2.1	6.8.2.4
	6.8.3.1	6.8.3.7	6.8.3.9
identifier	4	6.1.3	6.2.2.1
	6.2.2.5	6.2.2.7	6.2.2.8
	6.2.2.9	6.2.2.11	6.3
	6.4.1	6.4.2.3	6.4.3.3
	6.5.1	6.5.2	6.5.3.3
	6.6.1	6.6.2	6.6.3.1
	6.6.3.7.1	6.6.3.7.2	6.8.3.9
	6.8.3.10	6.10	
identifier-list	6.4.2.3	6.4.3.3	6.5.1
	6.6.3.1	6.6.3.7.1	6.6.3.7.3
	6.10		
implementation-defined	3.1	3.3	5.1
	5.2	6.1.7	6.4.2.2
	6.6.5.2	6.7.2.2	6.9.3.1
	6.9.3.4.1	6.9.3.5	6.9.5
	6.10		
implementation-dependent	3.2	3.4	5.1
	5.2	6.5.3.2	6.7.1
	6.7.2.1	6.7.3	6.8.2.2
	6.8.2.3	6.9.5	6.10
index-type	6.4.3.2	6.5.3.2	6.6.3.7.1
	6.6.3.8		
indexed-variable	6.5.3.1	6.5.3.2	6.6.3.7.2
integer-type	6.1.5	6.3	6.4.2.2
	6.4.2.3	6.4.6	6.6.6.2
	6.6.6.3	6.6.6.4	6.6.6.5
	6.7.2.2	6.7.2.5	6.9.1
	6.9.3.1	6.9.3.3	

label	6.1.2	6.1.6	6.2.1
	6.2.2.1	6.2.2.5	6.2.2.7
	6.2.2.8	6.2.2.9	6.2.2.11
	6.2.3.2	6.2.3.3	6.8.1
	6.8.2.4		
	6.4.6	6.7.1	6.7.2.5
	3.5	5	5.1
	6.1	6.1.4	6.1.7
	6.1.9	6.2.2.10	6.2.2.11
	6.2.3.3	6.4.2.2	6.4.3.1
member note	6.4.3.2	6.4.3.3	6.4.3.4
	6.4.3.5	6.4.4	6.4.7
	6.5.1	6.5.3.2	6.5.3.3
	6.5.4	6.6.3.1	6.6.3.7
	6.6.3.7.1	6.6.3.7.2	6.6.3.8
	6.6.4.1	6.6.5.2	6.6.5.3
	6.7.1	6.7.2.1	6.7.2.2
	6.7.2.5	6.8.1	6.8.2.4
	6.8.3.4	6.8.3.5	6.9.1
	6.9.2	6.9.3.4.2	6.9.4
number	6.10		
	5.1	6.1.7	6.4.2.2
	6.4.2.3	6.4.3.2	6.4.5
	6.6.3.6	6.6.6.4	6.7.3
	6.8.2.3	6.9.1	6.9.3.3
	6.9.3.4	6.9.3.4.1	6.9.3.4.2
	6.7.2.1	6.7.2.2	6.7.2.5
	6.5.1	6.7.1	6.7.2.1
	6.7.2.2	6.7.2.4	6.7.2.5
	6.8.3.5		
ordinal	6.4.2.1	6.4.2.2	6.4.2.3
	6.6.6.1	6.6.6.4	6.7.2.5
ordinal-type	6.4.2.1	6.4.2.4	6.4.3.2
	6.4.3.3	6.4.3.4	6.6.6.4
	6.7.1	6.7.2.5	6.8.3.5
parameter	6.8.3.9		
	6.6.1	6.6.3.1	6.6.3.2
	6.6.3.3	6.6.3.4	6.6.3.5
	6.6.3.6	6.6.3.7.1	6.6.3.7.2
	6.6.3.7.3	6.6.5.2	6.6.6.2
	6.6.6.5	6.8.3.9	6.9.1
	6.9.2	6.9.3	6.9.3.5
	6.9.4	6.10	
	6.4.1	6.5.1	6.7.2.5
	6.4.4	6.5.4	6.6.5.3
pointer pointer-type procedure	6.1.2	6.2.3.2	6.2.3.3
	6.4.4	6.5.4	6.6
	6.6.1	6.6.3.4	6.6.3.7.2
	6.8.2.3	6.8.3.9	6.9.1
	6.9.2	6.9.3	6.9.4
	6.9.5	6.10	
	6.1.4	6.2.3.2	6.2.3.3
	6.6.1	6.6.3.1	6.8.2.3
	6.1.4	6.2.1	6.6.1
	6.8.2.3		
procedure-block	6.2.3.1	6.2.3.2	6.2.3.3
	6.6.1	6.6.3.1	6.6.3.4
procedure-declaration	6.7.3	6.8.2.3	
	6.2.3.4	6.8.2.1	6.8.2.3
procedure-identifier	6.8.3.9		
procedure-statement			

program-parameters	6.2.3.5	6.10	
real-type	6.1.5	6.3	6.4.2.2
	6.4.6	6.6.6.2	6.6.6.3
	6.7.2.2	6.7.2.5	6.9.1
	6.9.3.1	6.9.3.4	
record-type	6.4.3.1	6.4.3.3	6.5.3.3
	6.6.5.3	6.8.3.10	
record-variable	6.4.3.3	6.5.3.3	6.8.3.10
reference	5.1	6.1.9	6.5.3.1
	6.5.3.3	6.5.4	6.5.5
	6.6.3.3	6.6.3.7.2	6.6.3.7.3
	6.8.2.2	6.8.3.10	
region	6.2.1	6.2.2.2	6.2.2.3
	6.2.2.4	6.2.2.5	6.2.2.6
	6.2.2.7	6.2.2.10	6.2.3.1
	6.2.3.2	6.3	6.4.1
	6.4.2.3	6.4.3.3	6.5.1
	6.5.3.3	6.6.1	6.6.2
	6.6.3.1	6.6.3.7.1	6.8.3.10
	6.10		
result	5.1	6.2.3.2	6.2.3.3
	6.2.3.5	6.6.1	6.6.2
	6.6.6.2	6.6.6.3	6.6.6.4
	6.7.2.2	6.7.2.4	6.7.2.5
	6.7.3	6.8.2.2	
same	3.5	5.2	6.1
	6.1.3	6.1.4	6.1.7
	6.2.3.3	6.4.1	6.4.2.2
	6.4.2.4	6.4.3.2	6.4.5
	6.4.6	6.4.7	6.5.3.1
	6.5.3.2	6.6.3.2	6.6.3.3
	6.6.3.5	6.6.3.6	6.6.3.7.1
	6.6.3.7.2	6.6.3.8	6.6.6.2
	6.6.6.4	6.7.1	6.7.2.2
	6.7.2.4	6.7.2.5	6.8.3.5
	6.8.3.10		
scope	1	6.2	6.2.2
	6.2.2.2	6.2.2.4	6.2.2.5
	6.2.2.8		
set-type	6.4.3.1	6.4.3.4	6.7.1
	6.7.2.5		
statement	5.1	6.2.1	6.2.3.2
	6.6.5.4	6.8.1	6.8.2.1
	6.8.3.1	6.8.3.4	6.8.3.5
	6.8.3.8	6.8.3.9	6.8.3.10
string-type	6.1.7	6.4.3.2	6.4.5
	6.4.6	6.6.3.7.1	6.7.2.5
	6.9.3.1	6.9.3.6	
structured-type	6.4.3.1	6.4.3.5	6.5.1
	6.8.2.2		
subrange	6.4.2.4	6.4.5	6.7.1
	6.9.1		
textfile	6.4.3.5	6.5.5	6.6.6.5
	6.9.1	6.9.2	6.9.3
	6.9.4	6.9.5	6.10
token	4	6.1	6.1.1
	6.1.2	6.1.8	6.1.9
totally-undefined	6.2.3.5	6.5.3.3	6.6.5.2
	6.6.5.3	6.8.2.2	6.9.4
	6.9.5		

type-identifier	6.2.2.9	6.2.2.11	6.4.1
	6.4.2.1	6.4.4	6.6.3.1
	6.6.3.2	6.6.3.3	6.6.3.6
	6.6.3.7.1	6.6.3.8	6.10
undefined	6.5.3.3	6.5.4	6.6.5.2
	6.6.5.3	6.6.6.5	6.7.1
	6.7.3	6.8.2.2	6.8.3.9
	6.9.1	6.9.3	6.9.4
variable	6.2.3.2	6.2.3.3	6.4.1
	6.4.3.5	6.4.4	6.5.1
	6.5.3.1	6.5.3.2	6.5.3.3
	6.5.4	6.5.5	6.6.3.1
	6.6.3.2	6.6.3.3	6.6.3.7.1
	6.6.3.7.2	6.6.3.7.3	6.6.5.2
	6.6.5.3	6.6.5.4	6.7.1
	6.8.2.2	6.8.3.9	6.8.3.10
	6.9.1	6.10	
	6.5.1	6.5.3.2	6.5.3.3
variable-access	6.5.4	6.5.5	6.6.3.3
	6.6.3.7.3	6.6.5.2	6.6.5.3
	6.7.1	6.7.3	6.8.2.2
	6.8.3.9	6.9.1	6.9.2
variant	6.4.3.3	6.5.3.3	6.6.5.3
word-symbol	6.1.2	6.1.3	6.1.4

Appendix C**Required Identifiers**

Identifier	Reference(s)
abs	6.6.6.2
arctan	6.6.6.2
Boolean	6.4.2.2
char	6.4.2.2
chr	6.6.6.4
cos	6.6.6.2
dispose	6.6.5.3
eof	6.6.6.5
eofin	6.6.6.5
exp	6.6.6.2
false	6.4.2.2
get	6.6.5.2
input	6.10
integer	6.4.2.2
in	6.6.6.2
maxint	6.7.2.2
new	6.6.5.3
odd	6.6.6.5
ord	6.6.6.4
output	6.10
pack	6.6.5.4
page	6.9.5
pred	6.6.6.4
put	6.6.5.2
read	6.6.5.2, 6.9.1
readln	6.9.2
real	6.4.2.2
reset	6.6.5.2
rewrite	6.6.5.2
round	6.6.6.3
sin	6.6.6.2
sqr	6.6.6.2
sqrt	6.6.6.2
succ	6.6.6.4
text	6.4.3.5
true	6.4.2.2
trunc	6.6.6.3
unpack	6.6.5.4
write	6.6.5.2, 6.9.3
writeln	6.9.4

Appendix C**Required Identifiers**

Identifier	Reference(s)
abs	6.6.6.2
arctan	6.6.6.2
Boolean	6.4.2.2
char	6.4.2.2
chr	6.6.6.4
cos	6.6.6.2
dispose	6.6.5.3
eof	6.6.6.5
eofln	6.6.6.5
exp	6.6.6.2
false	6.4.2.2
get	6.6.5.2
input	6.10
integer	6.4.2.2
ln	6.6.6.2
maxint	6.7.2.2
new	6.6.5.3
odd	6.6.6.5
ord	6.6.6.4
output	6.10
pack	6.6.5.4
page	6.9.5
pred	6.6.6.4
put	6.6.5.2
read	6.6.5.2, 6.9.1
readln	6.9.2
real	6.4.2.2
reset	6.6.5.2
rewrite	6.6.5.2
round	6.6.6.3
sin	6.6.6.2
sqr	6.6.6.2
sqrt	6.6.6.2
succ	6.6.6.4
text	6.4.3.5
true	6.4.2.2
trunc	6.6.6.3
unpack	6.6.5.4
write	6.6.5.2, 6.9.3
writeln	6.9.4

the *buffer-variable*.

D.19 For *new*(*p*,*c*₁.....*c*_{*n*}), it is an error if a variant of a variant-part within the *new* variable becomes active and a different variant of the variant-part is one of the specified variants.

D.20 For *dispose*(*p*), it is an error if the identifying-value had been created using the form *new*(*p*,*c*₁.....*c*_{*n*}).

D.21 For *dispose*(*p*,*k*₁.....*k*_{*m*}), it is an error if the variable had been created using the form *new*(*p*,*c*₁.....*c*_{*n*}) and *m* is not equal to *n*.

D.22 For *dispose*(*p*,*k*₁.....*k*_{*m*}), it is an error if the variants in the variable identified by the pointer value of *p* are different from those specified by the case-constants *k*₁.....*k*_{*m*}.

D.23 For *dispose*, it is an error if the parameter of a pointer-type has a nil-value.

D.24 For *dispose*, it is an error if the parameter of a pointer-type is undefined.

D.25 It is an error if a variable created using the second form of *new* is accessed by the identified-variable of the variable-access of a factor, of an assignment-statement, or of an actual-parameter.

D.26 For *pack*, the parameter of ordinal-type is assignment-compatible with the index-type of the unpacked array parameter.

D.27 For *pack*, it is an error if any of the components of the unpacked array are both undefined and accessed.

D.28 For *pack*, it is an error if the index-type of the unpacked array is exceeded.

D.29 For *unpack*, the parameter of ordinal-type is assignment-compatible with the index-type of the unpacked array parameter.

D.30 For *unpack*, it is an error if any of the components of the packed array are undefined.

D.31 For *unpack*, it is an error if the index-type of the unpacked array is exceeded.

D.32 *Sqr*(*x*) computes the square of *x*. It is an error if such a value does not exist.

D.33 For *In*(*x*), it is an error if *x* is not greater than zero.

D.34 For *sqrt*(*x*), it is an error if *x* is negative.

D.35 For *trunc*(*x*), the value of *trunc*(*x*) is such that if *x* is positive or zero then $0 \leq x - \text{trunc}(x) < 1$; otherwise $-1 \leq x - \text{trunc}(x) < 0$. It is an error if such a value does not exist.

D.36 For *round*(*x*), if *x* is positive or zero then *round*(*x*) is equivalent to *trunc*(*x*+0.5), otherwise *round*(*x*) is equivalent to *trunc*(*x*-0.5). It is an error if such a value does not exist.

D.37 For *chr*(*x*), the function returns a result of char-type which is the value whose ordinal number is equal to the value of the expression *x* if such a character value exists. It is an error if such a character value does not exist.

D.38 For *succ*(*x*), the function yields a value whose ordinal number is one greater than that of *x*, if such a value exists. It is an error if such a value does not exist.

D.39 For $\text{pred}(x)$, the function yields a value whose ordinal number is one less than that of x , if such a value exists. It is an error if such a value does not exist.

D.40 When $\text{eof}(f)$ is activated, it is an error if f is undefined.

D.41 When $\text{eoln}(f)$ is activated, it is an error if f is undefined.

D.42 When $\text{eoln}(f)$ is activated, it is an error if $\text{eof}(f)$ is true.

D.43 An expression denotes a value unless a variable denoted by a variable-access contained by the expression is undefined at the time of its use, in which case that use is an error.

D.44 A term of the form x/y is an error if y is zero.

D.45 A term of the form $i \text{ div } j$ is an error if j is zero.

D.46 A term of the form $i \text{ mod } j$ is an error if j is zero or negative.

D.47 It is an error if an integer operation or function is not performed according to the mathematical rules for integer arithmetic.

D.48 It is an error if the result of an activation of a function is undefined upon completion of the algorithm of the activation.

D.49 For an assignment-statement, the expression is of an ordinal-type whose value is assignment-compatible with the type possessed by the variable or function-identifier.

D.50 For an assignment-statement, the expression is of a set-type whose value is assignment-compatible with the type possessed by the variable.

D.51 For a case-statement, it is an error if none of the case-constants is equal to the value of the case-index upon entry to the case-statement.

D.52 For a for-statement, the value of the initial-value is assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed.

D.53 For a for-statement, the value of the final-value is assignment-compatible with the type possessed by the control-variable if the statement of the for-statement is executed.

D.54 On reading an integer from a textfile, after skipping preceding spaces and end-of-lines, it is an error if the rest of the sequence does not form a signed-integer.

D.55 On reading an integer from a textfile, the value of the signed-integer read is assignment-compatible with the type possessed by variable-access.

D.56 On reading a number from a textfile, after skipping preceding spaces and end-of-lines, it is an error if the rest of the sequence does not form a signed-number.

D.57 It is an error if the buffer-variable is undefined immediately prior to any use of *read*.

D.58 On writing to a textfile, the values of *TotalWidth* and *FracDigits* are greater than or equal to one; it is an error if either value is less than one.

D.59 For a conformant array, it is an error if the smallest or largest value specified by the index-type of the actual parameter lies outside the closed interval specified by the index-type-specification of the conformant-array-schema.

Copies of the British Standard 6192 may be purchased from

BSI Sales Department
Newton House
101 Pentonville Road
London N1 9ND

Telephone 01-837-8801 Telex 23218

Index

- Abs* 22, 24
- Algorithm 1, 2, 126
- And* 27
- Arctan* 23
- Array* 41, 78
 - of arrays 79
 - of *char* 80
- Assignment compatibility 53, 56, 79
- Assignment statement 5
- Automatic conversion 23

- Base *see* Type
- Begin* 6, 11
- Block 38, 43, 117
- Boolean* 27
- Boolean expression 27, 30
- Bound identifier 88
- Buffer variable 62, 63, 65

- Cardinality 49, 53, 72, 78, 80, 93, 120
- Case* 16, 17, 50, 117
- Char* 25, 55
- Character constant 25
- Choice 9, 13
- Chr* 26
- Code 2
- Comments 7
- Compatibility *see* Type
- Compilation 2
- Compile time mistake 2
- Compiler 2
- Component 55
- Compound statement 11
- Conditions, complex 28
- Conformability 88
- Conformant array parameter 80
- Conformant array schema 86
 - one-dimensional 87
 - multidimensional 89
- Congruity 111
- Const* 7
- Constant 5
- Constructor 55, 72

- Control constructs 9, 107
- Control variable 12, 57
- Conversion 85
- Cos* 23

- Debugging 2
- Declaration 4
- Delimiter 4
- Design *see* Top-down design, Stepwise refinement design
- Dispose* 124
- Div* 24
- Do* 11
- Downto* 12
- Dynamic *see* variable
- Dynamic data structure 119
- Empty statement 117
- Else* 14
- End* 6, 11, 16, 93
- Eof* 63
- Eoln* 33, 64
- Event indicator 58, 84
- Exp* 23
- Expression 6

- Factorial 84
- False* 27
- Field designator 94, 96
- Field width 35
- Field 94
- File* 62
- File, *Input* 64
 - internal/external 64
- Output* 63, 64
- sequential 61
- text 64
- File definition 62
- File generation 62
- File inspection 63
- For* 12, 50, 56
- Forward* 115
- Free-list 124
- Full evaluation 30

Function 42, 107
Functional parameter 110

Get 63
Goto 116
 restrictions 117

Heap 121
Host *see* Type

Identifier 4
If 13
In 73
Inclusion *see* Set
Index *see* Type
Input *see* File
Input devices 2
Instructions 1
Integer 23, 54
Intersection *see* Set
Inversion 85

Job control 2

Label 116
 restrictions 117
Line printer 2
Linked list 127
Ln 23

Mapping function 84, 99
Matrix multiplication 89
Maxint 54
Mod 24

n-tuple 82, 93
New 121
Nil 122
Not 27

Odd 28
On-line 2
Operator 6, 22
Or 27
Ord 26, 50
Ordinal *see* Type
Output *see* File
Output device 2

Packed 56, 80
Packing 82
Page 34, 63
Parameter, actual 40
 formal 40
 procedural/functional 110
 value 39, 41
 variable 40, 41, 56, 110
Pointer 41, 122
Powerset 70
Precedence 21, 29
Pred 50
Program 4
Procedural parameter 110
Procedure 38, 107
Procedure body 38
Procedure declaration 38
Procedure heading 38
Put 62, 63

Read 32, 64
Readln 32, 64, 65
Real 22, 54
Real number, unsigned 22
Record 82, 93
Recursion 112
Region 43, 94, 111
Relative complement 71
Repeat 10
Repetition 9
Reset 63, 64
Rewrite 62, 64
Running 2
Run time error 2
Round 24

Searching 83
Sin 23
Scalar product 82, 87
Scope 44, 108, 111
Selection 9, 16
Selector 55, 93
Sentinel 84, 129
Sequence 61
Set 71
Set, canonical 73
 constructor 72
 empty 73
 in Pascal 71

- inclusion 70
- intersection 71
- operations 73
- relative complement 71
- strict inclusion 70
- theory 70
- union 70
- Sqr* 23, 24
- Sqrt* 23
- Standard functions 21
- Standard types 21, 53
- Static *see* variable
- Statement 4
- Stepwise refinement design 17, 107
- Store allocation 109
- Strict inclusion *see* Set
- String constant 81
- Structured *see* Type
- Succ* 50
- Synonym 49
- Syntax 2
- Syntax diagram 2
- Tag field 104, 110, 122, 124
- Text* 66
- Text *see* File
- Then* 15
- To* 12
- Top-down design 108
- True* 27
- Trunc* 24
- Type 48
 - base 71
 - compatibility 56
 - definition 49
 - enumerated 51, 57
 - host 52
 - index 78
 - ordinal 12, 51, 52, 57, 71, 78
 - pointer 121
 - same 56
 - simple 50
 - standard 21, 53
 - structured 56
 - subrange 52, 57
 - unions 102
- Union *see* Type
 - discriminated 105
 - free 105
- Unpacking 81
- Until* 10
- Value *see* parameter
- Var* 5, 40
- Variable 5, *see also* Parameter
 - dynamic 41, 119
 - static 119
- Variant 102, 122
- While* 11
- With* 95, 130
- Write* 33, 62, 63, 81
- Writeln* 33, 63
- Zahn loop 57