1. **Type identifier**:

<table>
<thead>
<tr>
<th>Type Identifier</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>Any integer number</td>
</tr>
<tr>
<td>Real</td>
<td>Any real number</td>
</tr>
<tr>
<td>Char</td>
<td>Any character</td>
</tr>
<tr>
<td>Boolean</td>
<td>True or False</td>
</tr>
<tr>
<td>String</td>
<td>Any character string</td>
</tr>
</tbody>
</table>

2. **Operator**:

2.1 **Arithmetic operator**.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>( )</td>
<td>Parenthesis</td>
<td>1</td>
</tr>
<tr>
<td>–</td>
<td>Negation</td>
<td>1</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>2</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>2</td>
</tr>
<tr>
<td>div</td>
<td>Modular division</td>
<td>2</td>
</tr>
<tr>
<td>mod</td>
<td>Integer division</td>
<td>2</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>3</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>3</td>
</tr>
</tbody>
</table>

The order of precedence follows the priority of the table to the bottom.

div found the quotient of a division.

- Example: 23 div 6 = 3

mod found the remainder of a division.

- Example: 23 mod 6 = 5

2.2 **Relational Operator**.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Relation Tested</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equality</td>
<td>X = Y</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>X &gt; Y</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>X &lt; Y</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>X &gt;= Y</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>X &lt;= Y</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Inequality</td>
<td>X &lt;&gt; Y</td>
</tr>
</tbody>
</table>

When arithmetic and relational operators are used together, the arithmetic operators are always performed first.

2.3 **Logical operator**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>not</td>
<td>Opposite test</td>
<td>NOT (X)</td>
</tr>
<tr>
<td>and</td>
<td>Existence together</td>
<td>(X&gt;5) AND (Y&lt;3)</td>
</tr>
<tr>
<td>or</td>
<td>Any existence</td>
<td>(X&gt;5) OR (Y&lt;3)</td>
</tr>
</tbody>
</table>

Logical operator gives test of existence of certain conditions. It will return a single bit result, either "true" or "false".

3. **Mathematical Function**

3.1 sqr

- **Purpose**: Return the square.
- **Format**: sqr(X)
- **Remarks**: This function returns the square of X.

- Example:
  
  ```pascal
  Y := sqr(5);
  writeln(Y:1:0);
  SCREEN OUTPUT: 25
  ``

  This example calculates the square of 5.

3.2 sqrt

- **Purpose**: Return the square root.
- **Format**: sqrt(X)
- **Remarks**: X must be greater than or equal to zero. This function returns the square root of X.

- Example:
  
  ```pascal
  Y := sqrt(25);
  writeln(Y:1:0);
  SCREEN OUTPUT: 5
  ``

  This example calculates the square root of 25.

3.3 trunc

- **Purpose**: Return the largest integer that is less than or equal to X.
- **Format**: trunc(X)
- **Remarks**: X is any numeric expression and the result must be stored in integer type.

- Example:
  
  ```pascal
  Y := trunc(45.6);
  writeln(Y);
  SCREEN OUTPUT: 45
  ``

  Since 45.6 = 45 + 0.6, 45 is the largest integer that less than 45.6.

3.4 round

- **Purpose**: Return the largest integer that is nearest to integer.
- **Format**: round(X)
- **Remarks**: X is any numeric expression.

- Example:
  
  ```pascal
  writeln(round(12.3));
  writeln(round(-12.3));
  SCREEN OUTPUT: 12
  -12
  ```

Example: A = 5, B = 0, C = 6, D = 6

- (A>=B) or not (C=D)
- = (5>=0) or not (6=6)
- = (true) or not (true)
- = (true) or (false)
- = true
3.5 **random**

**Purpose**: Return a random number between 0 and 1.

**Format**: random

**Remarks**: The returned value is greater or equal to 0 and smaller than 1

**Example**:

```pascal
Y := random;
writeln (Y:8:6);
SCREEN OUTPUT : 0.595219
```

It returns a values \( 0 \leq Y < 1 \).

```pascal
Y := random*5;
writeln (Y:8:6);
SCREEN OUTPUT : 1.823544
```

It returns a values \( 0 \leq Y < 5 \).

3.6 **abs**

**Purpose**: Return the absolute value of an expression.

**Format**: abs(X)

**Remarks**: X may be any numeric expression. The absolute value of a number is always positive or zero.

**Example**:

```pascal
Y := abs(-5*0.325);
writeln (Y:4:3);
SCREEN OUTPUT : 1.625
```

3.7 **sin**

**Purpose**: Return the sine of X in radians.

**Format**: sin(X)

**Remarks**: X is an angle in radians. To convert degrees to radians, multiply by \( \frac{\pi}{180} \), where \( \pi = 3.141593 \).

**Example**:

```pascal
Y := sin(1);
writeln (Y:4:3);
SCREEN OUTPUT : 0.841
```

3.8 **cos**

**Purpose**: Return the cosine of X in radians.

**Format**: cos(X)

**Remarks**: X is an angle in radians. To convert degrees to radians, multiply by \( \frac{\pi}{180} \), where \( \pi = 3.141593 \).

**Example**:

```pascal
writeln (cos(1):4:3);
SCREEN OUTPUT : 0.540
```

3.9 **arctan**

**Purpose**: Return the arctangent of a numeric expression in radians.

**Format**: arctan(X)

**Remarks**: The expression X may be any numeric type. Result lies in the range \(-\pi/2\) to \(\pi/2\). To convert radians to degrees, multiply by \(180/\pi\), where \( \pi = 3.141593 \).

**Example**:

```pascal
PI := 3.141593;
RAD := arctan (2.5);
writeln (RAD:2:2);
DEG := RAD * 180/PI;
writeln ('TAN(' , DEG:2:2 , ') = ' , 2.5 ');
SCREEN OUTPUT : 1.19
TAN(68.20) = 2.5
```

3.10 **ln**

**Purpose**: Return the natural logarithm of X.

**Format**: ln(X)

**Remarks**: X must be greater than zero. The natural logarithm is the logarithm to the base e.

**Example**:

```pascal
Y := ln (3);
writeln (Y:4:3);
SCREEN OUTPUT : 1.099
```

The natural logarithm of 3 is 1.098661291.

3.11 **exp**

**Purpose**: Calculate the exponential function.

**Format**: exp(X)

**Remarks**: This function returns the mathematical number e to the power of X \( \left( e^X \right) \). e is the base of natural logarithm. If exp overflows, the "Overflow" error message is printed.

**Example**:

```pascal
Y := exp (2);
writeln (Y:4:3);
SCREEN OUTPUT : 7.389
```

4. **String Function**

4.1 **length**

**Purpose**: Return the number of characters in a string S.

**Format**: length(S)

**Remarks**: S is a string expression. Non-printing characters and blanks are counted.

**Example**:

```pascal
var S : string;
begin
S := 'bd42 5%?';
writeln (length(S));
end.
SCREEN OUTPUT : 8
```

There are 8 characters in the string S and spacing is counted as well.

4.2 **ord**

**Purpose**: Return the ASCII code for a character.

**Format**: ord(X)

**Remarks**: The result of the ord function is a numerical value that is the ASCII code of the given character. X must not be a variable (var).

**Example**:

```pascal
writeln (ord ('D'));
writeln (ord ('e'));
writeln (ord ('1'));
writeln (ord (1));
SCREEN OUTPUT : 68
```

This example shows that if a character is replaced by a number, the result is the number itself.
Below is a WRONG example:

```pascal
var X : char;
begin
  X := 'D';
  writeln (ord (X));
end.
SCREEN OUTPUT : ERROR
```

In this wrong example, X is declared as a variable, therefore it is wrong and ASCII code of ‘D’ can’t be outputted.

4.3 chr

**Purpose**: To convert an ASCII code to its character equivalent.

**Format**: `chr(X)`

**Remarks**: X lies within the range of 0 to 255. This function returns one character string of ASCII code X.

**Example**:

```pascal
Y := chr(75);
writeln (Y);
SCREEN OUTPUT : K
```

The character of ASCII code 75 is ‘K’.

4.4 val

**Purpose**: Return the numeric value of a string.

**Format**: `val(S,X,P)`

**Remarks**: S is a string expression. The val function converts the string into numeric value and stored in X, also strips leading blanks from the argument string. If any characters of S is not numeric, that is, non-leading blanks, non-numeric characters, etc., then X will return 0 and P stores the position in the string S of non-numeric character.

**Example**:

```pascal
var S1, S2 : string;
X1, X2 : real;
P1, P2 : integer;
begin
  S1 := '-23.6';
  val (S1, X1, P1);
  writeln (X1:3:2);
  writeln (P1);
  S2 := '56 d';
  val (S2, X2, P2);
  writeln (X2:3:2);
  writeln (P2);
end.
SCREEN OUTPUT :
-23.60
0
0
3
```

4.6 copy

**Purpose**: Return a string which is a substring of the given string.

**Format**: `copy(S, I, X)`

**Remarks**: S is any given string expression. I specifies start position from leftmost of the string. X counts the number of characters.

**Example**:

```pascal
var S1, S2 : string;
I, X : integer;
begin
  S1 := 'computer';
  I := 2;
  X := 4;
  S2 := copy (S1, I, X);
  writeln (S2);
  writeln (length (S2));
end.
SCREEN OUTPUT :
ompu
4
```

The output is ompu, it is a string with length 4.

5. Input-Output Function:

5.1 read and readln

**Purpose**: To input data from the keyboard.

**Format**: `read(X)` , `readln(X)`

**Remarks**: read will read a variable from the keyboard and then wait for another instruction. Readln will start reading a variable from the keyboard after pressing the ENTER and then wait for another instruction.

**Example**:

```pascal
variable (string) output :
S := 'abd';
writeln (S);
SCREEN OUTPUT :
abd
```

```pascal
variable (integer) output :
X := 5;
writeln (X);
SCREEN OUTPUT :
5
```

```pascal
variable (real) output :
X := 5.3;
writeln (X:2:3);
SCREEN OUTPUT :
5.300
```
variable (boolean) output:
X := true;
writeln (X);
SCREEN OUTPUT:
true

ordinary/constant output
writeln (‘sgd’);
SCREEN OUTPUT:
sgd:

The two different output can be used together and separated by a comma.
N := 30;
writeln (’The average mark of ’ , N , ’ students is ’ , X:5:2);
SCREEN OUTPUT:
The average mark of 30 students is 63.78

6. Subprogram Function:

6.1 Procedure
Purpose: Subroutine for carrying out a specified task.
Format: Procedure <name> ( <variable1 : type1> ; <variable2 : type2> ; … );
Remarks: The procedure is called by the procedure name. <variable> stores the value passed from the main program (or other subroutine). If the value of the <variable> is changed in the procedure and will be passed back to the main program (or subroutine), the declaration should become <variable : type>

Example:
Program PP;
var Y1 : integer;
Y2 : real;
Y3 : string;
Procedure Proc (X1:integer ; var X2:real ; S1:string);
begin
 ...
end;
Begin { this is the main program } 
Proc(Y1, Y2, Y3);
End.

* As you see, the variable X2 has “var” to identify its state, that is, after execution of the procedure, X2 will be stored back into Y2. However, there is no change in Y1 and Y3 (although X1 and S1 may be changed in the procedure).

6.2 Function
Purpose: Subroutine for calculating and storing a result.
Format: Function <name> ( <variable1 : type1> ; <variable2 : type2> ; … );
<type>;
Remarks: The function is called by the function name. <variable> stores the value passed from the main program (or other subroutine). The value of the <variable> in the function will NOT be passed back to the main program (or subroutine), the result calculated by the function will be passed back to the main program (or subroutine) through the function name, in other words, the function name is a variable. <type> define the type of the function name. In the function, the function name is used as a variable to calculate the result.

Example:
Program PP;
var Y1 : integer;
Y2 : real;
Y3 : real;
Function Func (X1:integer ; X2:real) : real;
begin
 ...
end;
Begin { this is the main program } 
Y3 := Func(Y1, Y2);
End.

* In this example, there are three variables constructed in the main program, that is Y1, Y2, and Y3. The function named as “Func” is created.

* The function is called from the main program just by including the name of the procedure and the values requested to be passed.

* As shown in the example, when the function is called, the value Y1 is stored into X1, and Y2 is stored into X2. Of course, the type of the values must be matched.

* Besides, the function name, “Func” is also a variable and with type real.

* After execution, the result is stored in “Func”, therefore, only “Func” will be passed back to the main program (or subroutine).

* Here in the example, Y3 is used to stored the value “Func” after the execution of the function,
that is, Y3 stores Func. However, there is no change in Y1 and Y2.

Below shows another example:

Program PP;
var Y1 : integer;
Y2 : integer;

Function Mean (X1:integer ; X2:integer) : real;
begin
    Mean := (X1 + X2)/2;
end;

Begin
    write ('Input first number : ');
    readln (Y1);
    write ('Input second number : ');
    readln(Y2);
    writeln ('Average : ' , Mean(Y1,Y2):1:2 );
End.

SCREEN OUTPUT :
Input first number : 6
Input second number : 11
Average : 8.50

7. Data Structure :

7.1 Array

Purpose : To arrange a series of data in a particular order.
Format : Array [1.. N] of <type>
Remarks : N is the size of array, or the number of elements in the array.
<type> is the type of the elements.
All elements in the array must have same type.
Example :
var mark : array [1..40] of real;
name : array [1..20] of string;
age : array [1..30] of integer;

* Often array is declared in “var”.

* First example shows that there is an array called mark with size 40, that is mark[1], mark[2], ... mark[40]. One stands for one variable (totally 40 variables) and all elements are in type of real.

* Second example shows that there is another array called name with size 20, that is name[1], name[2], ... name[20]. One stands for one variable (totally 20 variables) and all elements are in type of string.

* Third example shows that there is the third array called age with size 30, that is age[1], age[2], ... age[30]. One stands for one variable (totally 30 variables) and all elements are in type of integer.

* They are called as one-dimensional array.

Besides there is two-dimensional array, example :
var weather : array [1..12, 1..31] of string;

* The example shows that there is an array called weather with size 12 x 31, that is weather[1,1], weather[1,2], ... weather[1,31], weather[2,1], weather[2,2], ... weather[2,31], weather[3,1], ... weather[12,1], ... weather[12,31]. One stands for one variable (totally 372 variables) and all elements are in type of string.

* In general, the weather[i,j] may be used to stored the weather in a year. For example :
  weather[1,5] := 'cloudy';
  It stores the weather of 5 Jan which is cloudy.
  weather[8,1] := 'rainy';
  It stores the weather of 1 Aug which is rainy.
  It stores the weather of 25 Nov which is sunny.

* Thus, the first index represents the month, and the second one represents the day.

7.2 Record

Purpose : To define a type in order to store a series of data in a particular order.
Format : type <record name> = record
Remarks : Record aims at creating a type
With record, elements of different type can be stored together.
Example :

type buildtype = record
    name : string;
    number of room : integer;
    number of toilet : integer;
    number of chair : integer;
end;

var school : buildtype;
hotel : buildtype;

* In this example, a record data type is created.

* The data type contains four elements, one is string and three are integers.

* Therefore, if a “var” is created with the type, four variables are included.

* In the example, a “var” school is created, that means, there are four variables:
school.name
school.number of room
school.number of toilet
school.number of chair

* Each can only store a value. Thus, the “school” can only store one school name with its number of rooms, toilets and chairs.

* Also “hotel” creates four variables:
hotel.name
hotel.number of room
hotel.number of toilet
hotel.number of chair
and stores only data about one hotel.

* If an array is used and tried to replaced the record then only the number of rooms, toilets, and chairs can be stored in the array. For example :

var school : array [1..3] of integer;
hotel : array [1..3] of integer;
school[1] stores number of rooms.

* It is not a good way in this example using array as the meaning of school[1], school[2], school[3] are different.

* Up to here, it is observed that although the record data type can allow us to store different type of data together as a group, it stores a few number of data, such as, the above example, can only store information of one school and one hotel.

* In order to store more groups of data with same format, array is used. Refer to the above example, it means that we want to store some schools’ data (says 100 schools). Each school is a group, that is, totally 100 groups. Of course, each groups has four variables (name, number of room, number of toilet number of chair).

* Thus, the above example is rewritten as:

```
type buildtype = record
nename : string;
number of room : integer;
number of toilet : integer;
number of chair : integer;
end;

var school : array [1..100] of buildtype;
```

* The first group is:
  school[1].name
  school[1].number of room
  school[1].number of toilet
  school[1].number of chair

The second group is:
  school[2].name
  school[2].number of room
  school[2].number of toilet
  school[2].number of chair

  :

  :

  :

The last group is:
  school[100].name
  school[100].number of room
  school[100].number of toilet
  school[100].number of chair

* Actually, each group is a record in database and each element, such as, name, is a field in database.