JPas

Reference Guide

Version 1.0 (Draft)

Contents

1. Introduction

JPas is an interpreter of Pascal-like language, called *JPascal*, running in Java Virtual Machine (JVM). It supports a variety of features, including console input and output, as well as graphical interface, mouse and keyboard events.

Why?

To do: invent a decent motivation.

1.1 About JPascal

JPascal language draws its inspiration from early Turbo Pascal versions 3.0 to 5.0, before the object oriented programming was introduced.

> **Disclaimer:** *Back in the days, Turbo Pascal was a registered trademark of Borland International, Inc., USA, and it is still a registered trademark owned by CodeGear LLC, USA. Even though, to the author's knowledge, Turbo Pascal IDE has been released to public for free, it is important to emphasise that JPascal does not copy Turbo Pascal version of Pascal programming language, as explained in this section and with more details in Section 1.2. Turbo Pascal version 5.0 is used in this manual only as a historical reference and a baseline for feature comparison, and is further referred as* classical Pascal.

...

1.2 Differences from Classical Pascal

JPascal adds new features and removes some from the classical Pascal. Many of the changes are done because of JPas environment is running in JVM. JPascal is a higher level programming language than classical Pascal. Most of the removed features are linked to a specific system architecture.

Keywords that existed in classical Pascal, but have not been added to JPascal:

absolute	**inline**	**program**
external	**interrupt**	**set**
goto	**packed**	**unit**

1.2.1 Program structure

Program does not start with the keyword **Program**, and units do not start with **Unit**. In fact, these are not keywords in JPascal.

Minimal “Hello world” program in classical Pascal:

```
Program Hello;
Begin
  WriteLn('Hello world!');
End.
```

In JPascal it becomes a one-liner:

```
WriteLn('Hello world!').
```

1.2.2 Variable, type, and constant declarations

There are no dedicated declaration blocks (apart from the **interface** part in a unit), all declarations can be done in-place, anywhere in the code.

1.2.3 Unit structure

Interface/implementation structure is not limited to units, however it is not very useful anywhere else. The syntax is also different, as shown below.

In classical Pascal:

```
Unit UnitName;
interface
{Public declarations.}
implementation
{Implementation and private declarations.}
Begin
{Unit initialisaton code.}
End.
```

In JPascal:

```
interface
{Public declarations.}
implementation
{Unit initialisaton code with implementation.}
end.
```

1.2.4 Basic data types

Integer numbers are represented only with Integer type, which represents 32-bit signed integer numbers. There are no Byte, Word, or LongInteger.

Real is implemented using double precision floating point numbers (64-bit).

String type does not take size as a parameter. The strings have variable length realised by Java strings.

No *packed* data types; **packed** is not a reserved keyword.

1.2.5 Enumerated types

Enumerated types are defined using square brackets [and] instead of parentheses (and). This is done for consistency with array and record literals, see Section 1.2.8.

1.2.6 Arrays

In JPascal, array sizes are not enforced, and it is possible to write **array of** SomeType without the range. This is called abstracted array types, and they can be used in function and procedure argument types or within pointer types. It is not possible, however, to initialise an actual array object of this type, hence it cannot be used for variables.

1.2.7 Records

Variant records and tag fields are not possible.

Type-forwarding for pointers is even more permissive than in classical Pascal. It is possible to make a pointer to a record type within this record type's declaration:

```
type Person = record
      Name: String;
      Age: Integer;
      Next : ^Person;
   end;
```

1.2.8 Array and record literals

Array and record literals are defined using square brackets [and] (unlike regular parentheses (and) in classical Pascal. Because of this, you can use array and record literals anywhere in the code. Regular parentheses would make no distinction between single-value array and a parenthetic expression.

1.2.9 Sets

Sets are not added in this version of JPas, and probably will not be added any time soon. Even though it is a nice feature, it has a very limited application, so may not worth time investment. In the current implementation of JPas, **set** is not a reserved keyword

1.2.10 Files

Version 1.0 of JPas does not support working with files, however this functionality may be added later in one way or another. At the moment, **file** is not a keyword, and Text is not a reserved identifier.

1.2.11 For loops

Counter may be any L-value, not just a variable. You are allowed to change the counter inside the loop body. Starting and ending conditions are still evaluated only once, before the loop starts.

1.2.12 Pointers and memory addressing

JPas interpreter implements its own memory structure based on Java objects, which is different from the classical DOS memory model. It is not possible to get numerical addresses of memory items. As the result, pointer arithmetic is not possible.

Mem and Port arrays, provided in classical Pascal, do not exist in JPas. Almost all of the memory access functions and procedures are also removed.

No *absolute* access; **absolute** is not a reserved keyword.

Overlays are meaningless to modern systems, so they haven't been added to JPas.

1.2.13 Garbage collection

On the bright side, the memory management became much easier thanks to JVM's Garbage Collector (GC).

In classical Pascal, it was mandatory to call either Dispose or Mark/Release for every pointer allocated by the function New. Failing to do so would lead to memory leaks.

In JPas, disposing of unused memory is done by GC, and there are no such functions as Dispose, Mark, and Release.

```
var Q: ^Integer;
begin
  var P: ^Integer;
  New(P); {Allocate memory.}
  P^ := 8;
  Q := P;
end; {Variable P is destroyed.}

{The memory is still referenced by Q.}
WriteLn(Q^); {Prints '8'.}

Q := nil; {Clear the pointer.}
{Memory is automatically unallocated
because there are no pointers referencing it.}
```

1.2.14 Functions and procedures

No untyped arguments because absolute access is not allowed.

No **external** , **inline**, and other procedural modifiers.

No procedural types or function pointers.

Many standard functions and procedures are changed. Please refer to Section 9 for the reference on the updated versions.

1.2.15 Labels, exit and goto

Labels do not need an in-advance declaration. In JPascal, a label marks a statement.

goto statement has been deprecated in many languages. JPascal does not implement it.

Most interesting behaviour can be realised using **exit** Label statements, which exits the statement marked with the label. This always guarantees safe behaviour. It is still possible to use **exit** without a label to exit a function, procedure, or the program, like in classical Pascal.

2. Language Elements

...

2.1 Whitespace and Comments

Whitespace is any number of tabs, spaces and newline characters in any combination. A comment is any text enclosed in curly braces { and }, or between (* and *). Whitespaces and comments are ignored by the parser.

2.2 Keywords and Identifiers

The list of keywords:

and	**exit***	**mod**	**then**
array	**file**	**nil**	**to**
begin	**for**	**not**	**type**
case	**forward**	**of**	**until**
const	**function**	**or**	**uses**
div	**if**	**procedure**	**var**
do	**implementation**	**record**	**while**
downto	**in**	**repeat**	**with**
else	**interface**	**shl**	**xor**
end	**label**	**shr**	

* **exit** was a function in classical Pascal, not a keyword.

Identifier is a sequence of characters including Latin letters, numbers and underscore. It must contain at least one letter and cannot start with a number. Identifiers can be used as names for variables, constants, types, functions, procedures, etc.

The list of reserved identifiers:

Boolean	Integer	String*
Char	Real	Text
False	Result	True

* String was a keyword in classical Pascal, now it is considered a registered identifier.

Identifiers and keywords are case-insensitive. It is considered a good programming style to write keywords in lower case and start identifiers in capital case. The exceptions are the main **Begin** and **End** keywords of the program, which are usually capitalised, and local variables with single-character names, e.g. x or y, which can be lower case.

2.3 Constant Literals

2.3.1 Numeric literals

Any sequence of digits 0–9 form a decimal integer number. Unary 'minus' operator – creates a negative number. Unary 'plus' operator + can be optionally used to emphasise positive numbers. In JPascal, integer numbers are 32-bit, hence specifying numbers larger than $\left(2^{31}-1\right)$ or smaller than $\left(-2^{31}\right)$ will produce a compilation error.

Hexadecimal integer literals are preceded with $ symbol. It is possible to specify larger numbers in hexadecimal format, but they will be cropped to 32-bit with overflow. For instance, $FFFFFFFF will produce decimal (-1).

```
{Integer numbers:}
123
–8
+8 {Same as 8}
00025 {Same as 25}
$FF {Decimal 255}
$FFFFFFFF {Decimal –1 (overflow).}
```

Real numbers have integer and fractional parts separated by . (period). Scientific notation is also allowed. Examples are shown below.

```
{Real numbers:}
0.01
3.141592
2.5E-6 {Same as 0.000025}
2.5E+6 {Same as 250000}
1E9 {1000000000.0 – one billion as a real number.}
```

> **Note:** *Scientific notation always produces real numbers, so they need to be explicitly converted to integer.*

```
Int(1E9) {Now it is an integer number.}
```

There is an implicit cast from Integer to Real type.

2.3.2 Character and string literals

Strings of characters enclosed in single quotes ' produce string or character literals. Character literals contain exactly one character. JPascal supports Unicode managed by Java. All characters are 16-bit.

```
'Hello, I am a string.'
'A' {...and this is a character.}
```

It is also possible to create single character constants from their ANSI/Unicode values by preceding a decimal or hexadecimal numeric literal with a hash # symbol.

```
{Characters from ANSI/Unicode values:}
#48 {Same as '0' (zero)}
#65 {Same as 'A' (Latin A)}
#$41 {Also 'A' (Latin A)}
#$03A9 {Capital Greek Omega Ω}
```

There is an implicit cast from Char to Sting type.

2.3.3 Built-in constants

Boolean constants True and False.
Null-pointer constant **nil** is a keyword.

2.4 Operator symbols

Operator symbols recognised by JPascal are:

;	*	=	@
,	/	<>	^
:	[	>	.
:=	]	<	..
+	(	>=	
−	)	<=	

3. Program Structure

...

JPascal program is a single statement ending with . (period) symbol. “Hello world” program in JPascal looks like this:

```
WriteLn('Hello world!').
```

WriteLn outputs information to system console, which in this case is the string literal ‘Hello world!’.

Statement is defined as an executable action. Statements can be compound, internally executing more statements. For example, a block statement **begin** ... **end** is used to combine a sequence of multiple statements into one. Statements within a block statement are separated using semicolons.

```
Begin
  WriteLn('Hello world!');
  WriteLn('JPascal is here.');
  WriteLn('Bye!')
End.
```

3.1 Types of statements

This sections gives a brief overview of statements and program structuring. A detailed reference on statements can be found in the Section 7 of this guide.

Statements can be one of the following types:

- Empty statement.
- Declaration statements include variable, constant, type, function, and procedure declarations.
- Inclusion statement **uses**.
- Interface statement **interface** ... **implementation** ... **end**.
- Block statement.
- Control statements include exit, branching and loop statements.
- Assignment.

- Procedure call or expression statements.
- Scope statements include only **with** statement.

...

3.2 Scope

...

3.3 Memory and Stack

...

4. Declarations

...

4.1 Variable declarations

```
var VarName: VarType;
var VarName: VarType = InitialValue;
```

...

4.2 Constant declarations

```
const ConstName = Value;
const ConstName: ConstType = Value;
```

...

4.3 Type declarations

Type declaration statement:

```
type TypeName = TypeDefinition;
```

The statement above create a new type with the name *TypeName*. *TypeDefinition* can be either a name of another type, simple or declared, or one of the structured type definitions:

- Enumerated type.
- Subrange type.
- Array type.

- Record type.
- Pointer type.

4.4 Function and procedure declarations

See Section 8.

4.5 Units

...

4.5.1 Unit inclusion

```
uses StandardUnitName;
uses 'PathToUnitFile'; {User defined module.}
```

...

4.5.2 Interface blocks

...

```
interface
{Public declarations.}
implementation
{Unit initialisaton code with implementation.}
end.
```

...

5. Types

...

5.1 Simple Types

5.1.1 Integer

32-bit signed integer numbers. Default value is 0.

5.1.2 Real

64-bit floating point numbers. Default value is 0.0.

5.1.3 Boolean

False and True. Default value is False.

5.1.4 Char

Unicode characters (16-bit). Default value is a null-character #0.

5.2 String Type

Strings of characters. Default value is an empty string.
To do: String indexing.

5.3 Enumerated Types

Enumerated type definition example:

```
[North, East, South, West]
```

...

5.4 Subrange Types

Enumerated type definition example:

```
0..99
```

...

5.5 Ordinal Types

Ordinal types are the types that have ordering of their values. Ordinal types are Integer, Boolean, Char, enumerated, and subrange types.

5.6 Array Types

...

5.6.1 Array definitions

Array definition example:

```
array[1..5] of Integer
```

Multi-dimensional arrays:

```
array[1..5] of array[0..2] ofInteger
```

This is the same as:

```
array[1..5, 0..2] ofInteger
```

Index ranges can be of any ordinal type:

```
type Month = [January, February, March, April,
      May, June, July, August,
      September, October, November, December];
type SummerTime = array[June..August] of Boolean;
```

Enumerated or subrange types can be used for the entire index range.

```
type Direction = [North, East, South, West];
type Passability = array[Direction] of Boolean;
{The same as [North..West].}
```

5.6.2 Array literals

...

5.6.3 Accessing elements

```
SomeArray[3]
MultiArray[1, 3]
EnumArray[North]
```

Index types must match:

```
var P: Passability;
P[East] := True; {Ok.}
P[Direction(1)] := True; {Also ok.}
P[1] := True; {Error! Index type mismatch.}
```

5.6.4 Copy on assign

Arrays implement copy on assign:

```
var X, Y: array[0..3] of Integer;
Y[2] := 5;
X := Y; {Copies all values from Y to X.}
X[2] := 8; {Does not affect Y.}
WriteLn(X[2], ', ', Y[2]);
```

Outputs:

```
8, 5
```

Assigning array variables requires both arrays to have not only the same sizes, but also same index types and ranges. Copying is done recursively in multi-dimensional arrays, meaning that all dimensions of the array are copied over.

5.6.5 Abstract arrays

Defined without the range:

```
array of Integer
```

Cannot be instantiated, but can be used as argument types for functions and procedures, or within pointer types.

Abstract arrays cannot be used to declare types:

```
type ArrayOfInt = array of Integer;
{Error! Unknown range, cannot create array.}
```

To do: array literals as abstract array initialisers.

...

5.7 Record Types

...

5.7.1 Record definitions

Record definition example:

```
record
  X: Integer;
  Y: Integer;
  Pressed: Boolean;
end
```

Fields of the same type can be grouped together:

```
record
  X, Y: Integer;
  Pressed: Boolean;
end
```

Record fields may have any type, including arrays and other records. A record type cannot contain itself, but it can contain a pointer to the itself, as described in Section 5.8. If a record contains an abstract array, it is also considered abstract and cannot be instantiated or declared as a type.

5.7.2 Record literals

```
[X: 5; Y: 10; Pressed: False]
```

Record literal with all fields set to compile-time constants is itself a compile-time constant.

Examples of an array literal inside a record literal and an array literal of records:

```
[Values: [1, 2, 3]]
[[X: 0; Y: 1], [X: 1; Y: 0]]
```

...

5.7.3 Accessing fields

```
SomeRecord.X
```

A great way to efficiently access multiple fields of the same record variable or value is to use **with** statement:

```
with Rec do
  begin
    X := 10;
    Y := 3;
    Pressed := True;
  end;
```

...

5.7.4 Copy on assign

Records also implement copy on assign, which is done recursively for all embedded records and arrays.

5.8 Pointers

...

5.8.1 Pointer definitions

A pointer is defined by putting ^ in front of another type. The pointer will be able to reference memory objects of that type. For instance, a pointer to integer is defined as follows:

```
^Integer;
```

Pointer can reference any type, including arrays, records, and other pointers.

Sometimes it is necessary to use a pointer to a type even before that type is defined. The syntax of JPascal allows that. Pointer to a record type can be defined inside that type's declaration:

```
type Item = record
    Value: Integer;
    Next: ^Item;
  end;
```

The following example is also allowed:

```
type PRecA = ^RecA;
type PRecB = ^RecB;

type RecA = record
    P: PRecB;
  end;
type RecB = record
    P: PRecA;
  end;
```

This is called *forward type referencing*, and it is only allowed when defining pointers. Referenced types must be declared within the same scope as the pointers.

To do: pointers to abstract arrays.

5.8.2 Referencing and allocation

Operator `@` returns a reference to an existing memory object, such as variables, items within arrays, or record fields:

```
var N: Integer;
var A: array[1..5] of Integer;
var Rec: record
    X, Y: Integer;
  end;

var P: ^Integer;

P := @N; {Reference a variable.}
P := @A[3]; {Reference an array item.}
P := @Rec.X; {Reference a record field.}
```

It is possible to allocate memory objects directly using `New` procedure. Such an object will persist in memory as long as there is at least one pointer referencing it. Once there are no references, the object will be automatically deleted by the Garbage Collector.

```
New(P); {Create new integer object.}
```

To do: array allocation using New and NewArray.

5.8.3 Dereferencing

Dereferencing a pointer is done by placing ^ after the pointer.

```
var N: Integer = 3;
var P: ^Integer = @N;
P^ := 5;
WriteLn(P^, ', ', N);
```

Because *P* refers to the variable *N*, writing to P^ actually writes to *N*. Hence the code above will output:

5, 5

To do: object lifetime.

5.9 File types

...

6. Expressions

...

6.1 Operator Precedence

Highest precedence first:

- Literals: constant literals, array and record literals, variables, function calls, parentheses ( ).
- Postfix operators: indexed access [ ], record field access, pointer dereferencing ^.
- Prefix operators: +, −, **not**, @.
- *, /, **div**, **mod**, **and**, **shl**, **shr**.
- +, −, **or**, **xor**.
- Relational operations, **in** operator.

Operations with equal precedence are performed left to right.

...

6.2 Arithmetic operators

...

Table 6.1: Binary arithmetic operators

Operator	**Operation**	**Operand Types**	**Result Types**
+	addition	Integer or Real	Integer or Real
−	subtraction	Integer or Real	Integer or Real
*	multiplication	Integer or Real	Integer or Real
/	division	Integer or Real	Real
div	integer division	Integer	Integer
mod	modulo	Integer	Integer

Table 6.2: Unary arithmetic operators

Operator	Operation	Operand Types	Result Types
+	identity	Integer or Real	Integer or Real
–	negation	Integer or Real	Integer or Real

...

6.3 Logical Operators

...

Table 6.3: Boolean logical operators

Operator	Operation	Operand Types	Result Types
not	inversion	Boolean	Boolean
and	logical and	Boolean	Boolean
or	logical or	Boolean	Boolean
xor	logical xor	Boolean	Boolean

Table 6.4: Bitwise logical operators

Operator	Operation	Operand Types	Result Types
not	bitwise inversion	Integer	Integer
and	bitwise and	Integer	Integer
or	bitwise or	Integer	Integer
xor	bitwise xor	Integer	Integer
shl	shift left	Integer	Integer
shr	shift right	Integer	Integer

...

6.4 String Operators

...

Table 6.5: String operators

Operator	Operation	Operand Types	Result Types
+	concatenation	String	String

...

6.5 Relational Operators

...

Table 6.6: Relational operators

Operator	Operation	Operand Types	Result Types
=	equal	any type	Boolean
<>	not equal	any type	Boolean
<	less	comparable	Boolean
>	greater	comparable	Boolean
<=	less or equal	comparable	Boolean
>=	greater or equal	comparable	Boolean

...

7. Statements

...

7.1 Assignments

...

7.1.1 L-values versus R-values

L-values are expressions that can receive values from assignment operation. The following are L-values:

- Variables.
- Function and procedure arguments, including by-reference and by-value arguments, see Section 8.
- Dereferenced pointers.
- Items of L-value arrays.
- Fields of L-value records.

...

7.2 Block Statements

7.3 Declaration Statements

7.4 Exit Statement

Exit statement interrupts the execution of the current procedure of function and exits it immediately. In the top level block, the statement exits the program with exit code 0 (normal termination).

```
exit;
```

...

Labels can be used to mark certain statements. While inside that statement, the **exit** can be use to interrupt the statement by its label name.

```
exit LabelId;
```

The operation will interrupt all statements until *LabelId* is reached.

...

7.5 Branching Statements

...

7.5.1 If statement

```
if Condition then
  Statement;
```

...

7.5.2 If-else statement

```
if Condition then
  Statement
else
  ElseStatement;
```

...

7.5.3 Case statement

```
case Expression of
  Value1: Statement1;
  Value2: Statement2;
  ...
  else
    ElseStatement;
end;
```

...

7.6 Loop Statements

7.6.1 While loop

```
while Condition do
  Statement;
```

...

7.6.2 Repeat-until loop

```
repeat
  Statement1;
  Statement2;
  ...
until Condition;
```

...

7.6.3 For loop

```
for Index:=Start to Finish do
  Statement;
```

Or alternatively, counting down loop:

```
for Index:=Start downto Finish do
  Statement;
```

...

8. Procedures and Functions

...

8.1 Declarations

...

8.1.1 Procedure declarations

Example of a procedure:

```
procedure WriteSum(X, Y: Integer);
begin
  var Sum: Integer = X+Y;
  Write(X, '+', Y, '=', Sum);
end;
```

> **Note:** *Unlike in classical Pascal, variables and other declarations must be placed inside the* **begin** ... **end** *block.*

If the body is a single statement, it does not need to be enclosed in a **begin** ... **end** block:

```
procedure WriteSum(X, Y: Integer);
  Write(X, '+', Y, '=', X+Y);
```

8.1.2 Function declarations

Example of a function:

```
function Sum(X, Y: Integer): Integer;
begin
  Result := X+Y;
end;
```

Result is a reserved identifier representing a special variable that stores the function's return value. It is a bad programming style not to write any value to the Result, but it is not an error. A default value for the function type is returned in this case.

8.1.3 Forward declarations

...

8.2 Parameters

...

8.2.1 Value parameters

...

8.2.2 Reference parameters

... also called *variable parameters*.

In classical Pascal it was possible to use untyped variable parameters and access them via absolute addressing. JPascal does not allow absolute addressing, hence untyped parameters are not allowed too.

8.3 Procedural Types

...

```
type RealFunc = function(X, Y: Real): Real;

function Sum(X, Y: Real): Real;
  Result := X+Y;

var Func: RealFunc;
Func := Sum;

WriteLn(Func(3, 5)); {Calls Sum(3, 5) and outputs 8.}
```

...

9. Standard Procedures and Functions

...

Unlike user-defined procedures, the standard procedures and functions are processed by the pre-compiler in a special way, which opens the following advantages:

- The same function or procedure name can be used by different functions. The choice of the actual function depends on the types of the arguments (this is called *overloading*). The most notable example of overloading is type of the result being dependent on the type of the argument, like in Min, Max, or Abs functions.
- Variable number of arguments, denoted as (X, ...), means 0 or more arguments like *X*.
- Functions and procedures may take arguments of unspecified types or multiple specific types. Because of this, the section uses “invented” types that are not a part of the language but clarify what types can be accepted by the given procedure or function. These identifiers are listed below.

SomeType	Any type is accepted.
SimpleType	Only simple types (Integer, Real, Boolean, Char) or String are accepted.
Ordinal	Only ordinal types are accepted. Ordinal types are Integer, Boolean, Char, enumerated, and subrange types.
AnyFile	Untyped files (**file**), typed files (**file of** SomeType), and Text files are accepted.

9.1 Execution Control

9.1.1 Halt

```
procedure Halt;
```

Stop the program execution end exit JPas with exit code 0 (no error). This is the same as calling **exit** statement from the main block of the program; however, Halt can be called from anywhere in the code.

9.1.2 RunError

```
procedure RunError(Msg: String);
```

Stop the program execution with the runtime error and the specified message *Msg*. JPas exits with the code 1 (error).

9.2 Pointers

9.2.1 New

```
procedure New(var Ptr: ^SomeType);
```

Allocates a new value of type SomeType and puts the pointer to the variable.

9.2.2 NewArray

```
procedure NewArray(var Ptr: ^array of SomeType; Size, ... : Integer);
```

Allocates a new array of type SomeType with the given *Size* and puts the pointer to the variable. You can allocate multi-dimensional arrays by specifying sizes for each dimension. If the type of array has a fixed range and *Size* does not match, a range check error will be issued during runtime.

The benefit of using NewArray instead of New is that it does not require knowing the size of the array in advance. Consider the following example:

```
var P: ^array[0..3] of Integer;
var Q: ^array of Integer;

New(P); {Works fine.}
New(Q); {Error! Unknown range, cannot create array.}
```

The array cannot be allocated unless its size is known. However, sometimes it is only known in runtime. This problem can be solved using NewArray, which creates the range with integer indexing from 0 to ($Size - 1$):

```
var N: Integer = 8; {N is variable.}
NewArray(Q, N); {Creates an array[0..7] of Integer.}
```

9.3 Time

9.3.1 Delay

```
procedure Delay(Milliseconds: Integer);
```

Puts the execution into sleep for the specified number of *Milliseconds*. The exact timing is not guaranteed and the accuracy is determined by the operating system.

9.3.2 SysTime

```
function SysTime: Integer;
```

Returns system time: the number of seconds passed since 1 January 1970.

9.3.3 Elapsed

```
function Elapsed: Real;
```

Returns the number of seconds passed since the start of the program execution. The value is returned to millisecond precision, however the accuracy depends on the operating system.

9.4 Mathematical Functions

9.4.1 Int

```
function Int(X: Real): Integer;
```

Returns the integer part of the real number X. This function can be used to convert Real type values to Integer.

9.4.2 Frac

```
function Frac(X: Real): Real;
```

Returns the fractional part of the real number X. The result is always a positive number.

9.4.3 Round

```
function Round(X: Real): Integer;
```

Rounds the real number X to a closest integer. This function can be used to convert Real type values to Integer.

9.4.4 Sqrt

```
function Sqrt(X: Real): Real;
```

Returns square root of X.

9.4.5 Pi

```
function Pi: Real;
```

Returns constant π.

> **Note:** *During pre-compilation, this function is always optimised to a constant value.*

9.4.6 ArcTan

```
function ArcTan(X: Real): Real;
```

Returns arctangent of X in radians.

9.4.7 Sin and Cos

```
function Sin(X: Real): Real;
function Cos(X: Real): Real;
```

Returns the sine (or cosine) value of the angle X, where X is in radians.

9.4.8 Exp

```
function Exp(X: Real): Real;
```

9.4.9 Ln

```
function Ln(X: Real): Real;
```

9.4.10 Abs

```
function Abs(X: Real): Real;
function Abs(X: Integer): Integer;
```

Returns the absolute value of X. X can be integer or real, the result of the function will be of the same type as the argument.

9.4.11 Sqr

```
function Sqr(X: Real): Real;
function Sqr(X: Integer): Integer;
```

Returns the square of X, i.e. `(X*X)`. X can be integer or real, the result of the function will be of the same type as the argument.

9.4.12 Min and Max

```
function Min(X, Y: Real): Real;
function Min(X, Y: Integer): Integer;
function Max(X, Y: Real): Real;
function Max(X, Y: Integer): Integer;
```

Returns the minimum (or maximum) of X and Y. X and Y can be integer or real, the result of the function will be of the same type as the arguments, or Real if their types differ.

9.4.13 Odd

```
function Odd(X: Integer): Boolean;
```

Returns True if the argument is an odd number, i.e. (X **mod** 2)<>0. Only works for integer numbers.

9.5 Ordinal Functions

9.5.1 Succ

```
function Succ(N: Ordinal): Odrinal;
```

Returns the ordinal successor value for N. For integer N, it is $(N + 1)$. This function never goes out of range, but may overflow, as shown in the example below:

```
type Sides = [North, East, South, West];
var N: Sides = Succ(West);
WriteLn(N=North); {Prints 'true'}
```

9.5.2 Pred

```
function Pred(N: Ordinal): Odrinal;
```

Returns the ordinal predecessor value for N. For integer N, it is $(N - 1)$. This function never goes out of range, but may overflow (see Succ function).

9.5.3 Inc

```
procedure Inc(var N: Ordinal);
```

Change the value of the variable N to its ordinal successor. This is the same as:

```
N := Succ(N);
```

9.5.4 Dec

```
procedure Inc(var N: Ordinal);
```

Change the value of the variable N to its ordinal predecessor. This is the same as:

```
N := Pred(N);
```

9.5.5 Ord

```
function Ord(Ordinal): Integer;
```

Returns the integer index for the ordinal N in relation to its range, starting with 0. If N is a character, the function returns its Unicode value. Integer N returns itself.

9.6 Strings and Characters

9.6.1 Chr

```
function Chr(Code: Integer): Char;
```

Returns a character for the given Unicode value.

9.6.2 UpCase and LowCase

function UpCase(C: Char): Char;
function UpCase(S: String): String;
function LowCase(C: Char): Char;
function LowCase(S: String): String;

Returns an uppercase (or lowercase) value for the given character or string of characters. The type of the function is determined by the type of its argument.

9.6.3 Format

function Format(Fmt: String; Value, ... : SimpleType): String;

To do: Java's String.format.

9.6.4 Str

function Str(Value: SimpleType): String;

Converts a value of simple type (Boolean, Integer, or Real) to a string using some default formatting. For more control, use Format function.

9.6.5 Val

procedure Val(S: String; **var** Target: Integer; **var** Code: Integer);
procedure Val(S: String; **var** Target: Real; **var** Code: Integer);

Attempts to parse a numeric value from the given string *S* assuming the radix in *Code*. Real numbers only accept radix 10 (decimal). On success, the result is put in the variable *Target*. On failure, the *Code* is reset to 0.

9.6.6 Copy

function Copy(S: String; StartPos, EndPos: Integer): String;

Returns a substring of *S* from *StartPos* to *EndPos* (inclusive). Positions are numbered starting from 1.

9.6.7 Length

```
function Length(S: String): Integer;
```

Returns the number of characters in a string. For array length function, see Section 9.7.2.

9.6.8 Pos

```
function Pos(S, Target: String): Integer;
```

Returns the position of the first occurrence of substring *S* in the string *Target*. Positions are starting from 1. If *S* is not found, the function returns 0.

9.6.9 Concat

```
procedure Concat(var Target: String; S, ... : String);
```

The procedure concatenates a number of strings to the string variable *Target*. This is the same as:

```
Target := Target + S + ... ;
```

9.6.10 Delete

```
procedure Delete(var Target: String; StartPos, EndPos: Integer);
```

Deletes a portion from a string. Positions are starting from 1.

9.6.11 Insert

```
procedure Insert(var Target; S: String; StartPos: Integer);
```

Inserts a string into another at a given position. Positions are starting from 1.

9.7 Arrays and Memory

9.7.1 Fill

```
procedure Fill(var Target: Array of SomeType; Value: SomeType);
```

Fills the array *Target* with value *Value*. The procedure also works with multi-dimensional arrays, like shown in the code below:

```
var X: array[1..4, 1..3] of Integer;
Fill(X, 1);
```

The result is:
[[1, 1, 1], [1, 1, 1], [1, 1, 1], [1, 1, 1]]

Value can be an array too, if the ranges match:

```
var X: array[1..4, 1..3] of Integer;
var Y: array[1..3] of Integer = [1, 2, 3];
Fill(X, Y);
```

And the result is:
[[1, 2, 3], [1, 2, 3], [1, 2, 3], [1, 2, 3]]

9.7.2 Length

```
function Length(var Target: Array of SomeType): Integer;
```

Returns the number of items in an array. For non-abstract arrays, this number is statically defined by the range as $(max - min + 1)$. For abstract arrays allocated using NewArray, this number is defined by the *Size* parameter during array allocation.

For string length function, see Section 9.6.7.

9.7.3 Swap

```
procedure Swap(var X, Y: SomeType);
```

Swaps the values of two variables.

9.8 Random Number Generator (RNG)

9.8.1 Randomize

```
procedure Randomize;
procedure Randomize(Seed: Integer);
```

Set the seed for RNG to a specific (if given) or random (default) value.

9.8.2 Random

```
function Random: Real;
function Random(N: Integer): Integer;
```

Generate a random value. Without a parameter, the function generates a real value in the range $[0, 1)$. For the given integer argument N, the function generates an integer value in the range $[0, N)$. N must be positive.

10. Input and Output

...

10.1 Console

...

10.1.1 Write and WriteLn

```
procedure Write(Value, ... : SimpleType);
procedure WriteLn(Value, ... : SimpleType);
```

...

```
procedure WriteLn;
```

WriteLn without parameters outputs a newline character.

10.1.2 Read

```
procedure Read(var Target, ... : SimpleType);
```

...

10.1.3 ReadLn

```
procedure ReadLn;
procedure ReadLn(var Line: String);
```

...

```
procedure ReadLn(var Target, ... : SimpleType);
```

...

10.2 Working with Files

10.2.1 Assign

```
procedure Assign(F: AnyFile; FileName: String);
```

Assign a file name *FileName* to a file object *F*. The file must not be opened, call Close before re-assigning the file.

10.2.2 Reset

```
procedure Reset(F: AnyFile);
```

Open file for reading. If the file is already opened, it will be closed first.

10.2.3 Rewrite

```
procedure Rewrite(F: AnyFile);
```

Create a new file and open it for writing. If the file already exists, it will be overwritten. If the file is already opened, it will be closed first.

10.2.4 Append

```
procedure Append(F: AnyFile);
```

Open a file for writing. If the file already exists, new output will be appended to the end of the file. If the file is already opened, it will be closed first.

10.2.5 Eof

```
function Eof(F: AnyFile): Boolean;
```

Test if there is any data in a file to be read. The file must be opened for reading.

10.2.6 Flush

```
procedure Flush(F: AnyFile);
```

Flush any pending buffers to the disk. The file must be opened for writing.

10.2.7 Close

```
procedure Close(F: AnyFile);
```

Close the file if it has been opened.

10.2.8 Erase

```
procedure Erase(F: AnyFile);
```

Erase the file from the disk. The file must not be opened. The function will fail if *F* is assigned to the console.

10.2.9 Rename

```
procedure Rename(F: AnyFile, NewName: String);
```

Rename the file to *NewName*. The file must not be opened. The function will fail if *F* is assigned to the console.

10.2.10 FileSize

```
function FileSize(F: AnyFile): Integer;
```

Returns the file size in bytes. The file must not be opened. The function will fail if *F* is assigned to the console.

10.3 Working with Directories

10.3.1 GetDir

```
function GetDir: String;
```

Get full path to the current working directory.

10.3.2 ChDir

```
procedure ChDir(Path: String);
```

Change working directory.

10.3.3 MkDir

```
procedure MkDir(Path: String);
```

Create new directory at the specified location.

10.3.4 RmDir

```
procedure RmDir(Path: String);
```

Delete directory. Only empty directories can be deleted using this procedure.

10.3.5 FindFiles and FindDirs

```
function FindFiles(var Ptr: ^array of String): Integer;
function FindDirs(var Ptr: ^array of String): Integer;
```

List names of all files (or subdirectories) in the current working directory.

The function allocates the array of required size and links it to the pointer *Ptr*. The number of elements in the array is returned as the function result.

10.4 Text Files

...

10.4.1 Write and WriteLn

```
procedure Write(var T: Text; Value, ... : SimpleType);
procedure WriteLn(var T: Text; Value, ... : SimpleType);
```

...

```
procedure WriteLn(T: Text);
```

WriteLn without parameters outputs a newline character.

10.4.2 Read

```
procedure Read(var T: Text; var Target, ... : SimpleType);
```

...

10.4.3 ReadLn

```
procedure ReadLn(var T: Text);
procedure ReadLn(var T: Text; var Line: String);
```

...

```
procedure ReadLn(var T: Text; var Target, ... : SimpleType);
```

...

10.5 Untyped Files

...

10.5.1 Write

```
procedure Write(var F: file; Value, ... : SimpleType);
```

...

10.5.2 BlockWrite

```
procedure BlockWrite(var F: file; var Values: array of SimpleType;
    Count: Integer);
```

...

10.5.3 ByteWrite

```
procedure ByteWrite(var F: file; Value, ... : Integer);
procedure ByteWrite(var F: file; var Values: array of Integer;
    Count: Integer);
```

...

10.5.4 Read

```
procedure Write(var F: file; var Target, ... : SimpleType);
```

...

10.5.5 BlockRead

```
procedure BlockWrite(var F: file; var Target: array of SimpleType;
      MaxCount: Integer);
```

...

10.5.6 ByteRead

```
procedure ByteWrite(var F: file; var Target, ... : Integer);
procedure ByteWrite(var F: file; var Target: array of Integer;
      MaxCount: Integer);
```

...

10.6 Typed Files

...

10.6.1 Write

```
procedure Write(var F: file of SomeType; Value, ... : SomeType);
```

...

10.6.2 BlockWrite

```
procedure BlockWrite(var F: file of SomeType;
      var Values: array of SomeType; Count: Integer);
```

...

10.6.3 Read

```
procedure Write(var F: file of SomeType; var Target, ... : SomeType);
```

...

10.6.4 BlockRead

```
procedure BlockWrite(var F: file of SomeType;
    var Target: array of SomeType; MaxCount: Integer);
```

...

11. Graph2D Unit

...

11.1 Graph2D Elements

11.1.1 Window

Graph2D window supports the following built-in control keys:

Alt - +	Zoom in by increasing pixel scale.
Alt - −	Zoom out by decreasing pixel scale.
Alt - **Enter**	Toggle fullscreen mode.

Closing the Graph2D window will immediately terminate the application as if calling Halt procedure.

11.1.2 Canvas

...

11.1.3 Bitmaps

...

11.1.4 Colours

Graph2D uses 32-bit colours: 8-bit per red, green, and blue channels, and also 8-bit transparency.

...

11.2 Window Management

11.2.1 InitWindow

```
procedure InitWindow(Title: String; Width, Height, PixelScale: Integer);
```

Creates and shows Graph2D window with specified *Title*, screen buffer *Width* and *Height*, and *PixelScale*. Pixel scale of 0 triggers fullscreen mode.

11.2.2 PresentWindow

```
procedure PresentWindow;
```

Requests Graph2D window to display screen buffer.

11.3 Mouse and Keyboard Input

11.3.1 MouseX and MouseY

```
function MouseX: Integer;
function MouseY: Integer;
```

Returns current mouse coordinates (X or Y).

11.3.2 LeftMouse and RightMouse

```
function LeftMouse: Boolean;
function RightMouse: Boolean;
```

Returns the current state of mouse buttons (left or right). True means the button is pressed.

11.3.3 KeyDown

```
function KeyDown(ScanCode: Integer): Boolean;
```

Test the current state of a specific key. True means the key is pressed.

11.3.4 KeyPressed

```
function KeyPressed: Boolean;
```

Test if the keyboard event buffer is not empty. See ReadKey for more details.

11.3.5 ReadKey

```
function ReadKey: Char;
```

Retrieve and remove next event from the keyboard event buffer. The buffer logs typed keys and pressed control keys. If the returned value is zero character #0 then the event represents a control key; call ReadKey again to receive the scan code of this key.

Keyboard event buffer can store up to 32 events and does not keep events older than 1 second. If the buffer is empty, ReadKey will keep returning #0.

11.4 Bitmaps

11.4.1 CreateBitmap

```
function CreateBitmap(Width, Height: Integer; Alpha: Boolean): Integer;
```

...

11.4.2 LoadBitmap

```
function LoadBitmap(FileName: String): Integer;
```

...

11.4.3 LoadAtlas

```
procedure LoadAtlas(FileName: String; Width, Height: Integer;
    var Bitmaps: array of Integer; Count: Integer);
```

...

11.4.4 GetBitmap

```
function GetBitmap(X, Y, Width, Height: Integer): Integer;
```

...

11.4.5 PutBitmap

```
procedure PutBitmap(X, Y, Bitmap: Integer);
```

...

11.4.6 StretchBitmap

```
procedure StretchBitmap(X, Y, Width, Height, Bitmap: Integer);
```

...

11.4.7 DiscardBitmap

```
procedure DiscardBitmap(Bitmap: Integer);
```

...

11.5 Canvas Management

11.5.1 UseWindowCanvas

```
procedure UseWindowCanvas;
```

...

11.5.2 UseBitmapCanvas

```
procedure UseBitmapCanvas(Bitmap: Integer);
```

...

11.5.3 ClearCanvas

```
procedure ClearCanvas;
```

Clears the active canvas with the background color, previously set by SetBackground. Default color is opaque black.

11.5.4 CanvasWidth and CanvasHeight

```
function CanvasWidth: Integer;
function CanvasHeight: Integer;
```

Get canvaswidth or height. For the window canvas, these values are set by InitWindow procedure. For a bitmap canvas, these are the width and the height of the bitmap.

11.5.5 GetPixel

```
function GetPixel(X, Y: Integer): Integer;
```

...

11.5.6 PutPixel

```
procedure PutPixel(X, Y, Color: Integer);
```

...

11.6 Canvas Settings

11.6.1 HighQuality and LowQuality

```
procedure HighQuality;
procedure LowQuality;
```

Sets graphics quality for the subsequent drawing operations.
In high quality:

- Antialiasing is on: lines and shapes have smooth non-pixelated edges.
- Stroke control is set to “pure”: coordinates are not normalised to integer values during transforms.

- Bitmaps use bilinear filtering.
- Text antialiasing and fractional metrics are on.

In low quality:

- Antialiasing is off, no sub-pixel rendering.
- Stroke control is set to "normalize".
- Bitmaps use nearest neighbour filtering.
- Text antialiasing and fractional metrics are off.

11.6.2 TransparencyOn and TransparencyOff

```
procedure TransparencyOn;
procedure TransparencyOff;
```

Change the transparency mode. If transparency is on, subsequent colour setting operations SetBackground, SetPen, SetPaint, GradientPaint, and RadialPaint will use higher 8 bits of the colour as alpha channel (transparency). If transparency is off, these bits will be ignored. Transparency mode does not affect previous colour settings.

11.6.3 SetBackground

```
procedure SetBackground(Color: Integer);
```

Set active background colour. Background colour is used only in ClearScreen procedure.

11.6.4 SetPen

```
procedure SetPen(Color :Integer; PenWidth: Real);
```

Set pen colour and width. Pen is used in line and contour drawing procedures, and for rendering text.

11.6.5 SetPaint

```
procedure SetPaint(Color: Integer);
```

Set flat colour paint. The paint is used for filled shape drawing.

11.6.6 GradientPaint

```
procedure GradientPaint(X1, Y1, X2, Y2, Color1, Color2: Integer);
```

Set linear gradient paint stretching from point (*X1*, *Y1*) having *Color1* to point (*X2*, *Y2*) having *Color2*. The paint is used for filled shape drawing.

11.6.7 RadialPaint

```
procedure RadialPaint(X, Y, R, Color1, Color2: Integer);
```

Set radial gradient paint stretching from centre (*X*, *Y*) having *Color1* to the radius *R* having *Color2*. The paint is used for filled shape drawing.

11.6.8 SetClip

```
procedure SetClip(X, Y, Width, Height: Integer);
```

Set the clipping area to the specified rectangle. (*X*, *Y*) is the top-left corner of the rectangle. All subsequent drawing procedures will be limited to this area. This restriction can be removed by ResetClip procedure.

11.6.9 ResetClip

```
procedure ResetClip;
```

Reset the clipping area specified by SetClip procedure, so the drawing can be done on the entire screen.

11.7 Drawing

11.7.1 DrawLine

```
procedure DrawLine(X1, Y1, X2, Y2: Integer);
```

Draws a line from point (*X1*, *Y1*) to point (*X2*, *Y2*) using the current pen.

11.7.2 DrawRect and FillRect

```
procedure DrawRect(X, Y, Width, Height: Integer);
procedure FillRect(X, Y, Width, Height: Integer);
```

Draws a rectangle with the specified *Width* and *Height*. (X, Y) is the top-left corner of the rectangle. DrawRect draws an outline with the current pen. FillRect draws an area filled with the current paint.

DrawRoundRect and FillRoundRect

```
procedure DrawRoundRect(X, Y, Width, Height, R: Integer);
procedure FillRoundRect(X, Y, Width, Height, R: Integer);
```

Draws a rounded rectangle with the specified *Width* and *Height,* and corner radius *R*. (X, Y) is the top-left corner of the rectangle. DrawRoundRect draws an outline with the current pen. FillRoundRect draws an area filled with the current paint.

11.7.3 DrawOval and FillOval

```
procedure DrawOval(X, Y, Width, Height: Integer);
procedure FillOval(X, Y, Width, Height: Integer);
```

Draws an oval with the specified *Width* and *Height*. X is the leftmost coordinate of the oval, Y is the topmost coordinate. DrawOval draws an outline with the current pen. FillOval draws an area filled with the current paint.

11.7.4 DrawArc and FillArc

```
procedure DrawArc(X, Y, Width, Height,
    StartAngle, ArcAngle: Integer);
procedure FillArc(X, Y, Width, Height,
    StartAngle, ArcAngle: Integer);
```

Draws an arc of the oval with the specified *Width* and *Height*. X is the leftmost coordinate of the oval, Y is the topmost coordinate. The arc starts at *StartAngle* degrees and spans *ArcAngle* degrees. DrawArc draws an outline with the current pen. FillArc draws filled sector of an oval.

11.7.5 DrawPolyline, DrawPolygon, and FillPolygon

```
procedure DrawPolyline(NumPoints: Integer;
    PointsX, PointsY: array of Integer);
procedure DrawPolygon(NumPoints: Integer;
    PointsX, PointsY: array of Integer);
procedure FillPolygon(NumPoints: Integer;
    PointsX, PointsY: array of Integer);
```

Draws a polyline or a polygon using the specified points: array *PointsX* stores the *X* coordinates, array *PointsY* stores *Y* coordinates. *NumPoints* is the number of points to draw; the arrays must have at least this number of items each.

DrawPolyline draws the line with the current pen. DrawPolyline draws the line with the current pen and connects the starting and ending points. FillPolygon draws an area filled with the current paint.

11.8 Text and Fonts

11.8.1 SetTextFont

```
procedure SetTextFont(FontName: String; Bold, Italic: Boolean);
```

Load and set the font for rendering text. Fonts are cached once loaded, hence the consequent calls with the same *FontName* do not cause performance overhead. *Bold* and *Italic* arguments can be used to modify the style of the font.

11.8.2 SetTextSize

```
procedure SetTextSize(Size: Integer);
```

Change the size of the currently selected font. The *Size* is in pixels, but may slightly vary for some fonts.

11.8.3 DrawText

```
procedure DrawText(X, Y: Integer; Text: String);
```

Render the line of text on the screen going to the left from X. Y is the coordinate of the baseline. This procedure ignores newline characters and can output only a single line of text.

11.8.4 TextWidth

```
function TextWidth(Text: String): Integer;
```

The function calculates the width of the given text for the currently selected font.

11.9 Colour Calculations

11.9.1 GetAlpha, GetRed, GetGreen, and GetBlue

```
function GetAlpha(Color: Integer): Real;
function GetRed(Color: Integer): Real;
function GetGreen(Color: Integer): Real;
function GetBlue(Color: Integer): Real;
```

Get an individual component of the given colour: alpha (transparency), red, green, or blue. The value is returned as real number normalised to the range $[0, 1]$.

11.9.2 MakeRGB

```
function MakeRGB(R, G, B, A: Real): Integer;
```

Create a colour value from its individual components: R for red, G for green, B for blue, and A for alpha (transparency). The components are automatically clamped to the range $[0, 1]$. This function ignores transparency setting and always uses alpha component.

11.9.3 BlendColors

```
function BlendColors(S: Real; Color1, Color2: Integer): Integer;
```

The function interpolates between *Color1* and *Color2* proportionally to the weight S in the range $[0, 1]$. The function returns *Color1* if S is 0, *Color2* if S is 1, and blended color if S is in between.

11.10 Miscellaneous

11.10.1 FPSCount

```
function FPSCount: Real;
```

Frames Per Second (FPS) counter is a popular profiling tool for animation and games. The function returns the current FPS count. In order to get correct readings, this function must be called exactly once per frame.

11.10.2 Interpolate

```
function Interpolate(S, X1, X2: Real): Real;
```

This is a helper function that calculates linear interpolation between two values *X1* and *X2*. Weight *S* is automatically clamped to the range $[0, 1]$

Index