Project D
Dynamic Language

1. Common Description

- Object types are not specified and can change while program execution
- The language is **interpreted**
- Major notion: variable & literal (constant)
- Program structure: a sequence of declarations and statements
- Builtin types: built-in: integer, real, boolean, string
  User-defined types: array, tuple, function
- Implicit type conversions
- Statements: assignment, if/while, return, input/output

2. Informal language specification

The program is a sequence of statements. Statements can be separated by character “semicolon” or newline characters.

```
Program : { Statement ‘;’ }
```

Statements execute one after the other, starting with the first in text statement.

There are two category of statements: **declaration** of the variable and the operator. Declaration of the variable enter in the current scope new named object, which can store some value. The operators itself set a specific action, which semantic could consist of calculation of new value for some variable (assignment), or it could consist of changing the sequential order of statements or of outputting values to the console..

Declaration of a variable could be in any position within in the program. A variable introduced by a declaration is considered as active within its scope, from the point of its declaration to the end of that scope. Within the scope, only one variable with the given name is allowed.

If a certain scope contains other (nested) scopes, then variables declared in a nested scope may have names that match the names in the enclosing scope. In this case, variables from the nested scope hide variables with the same names from the enclosing scope.

```
Declaration   : var VariableDefinition
                        { ‘,’ VariableDefinition }

VariableDefinition : IDENT [ ':=' Expression ]
```

When declaring a variable, you can set its initial value. The initial value of a variable can subsequently be changed using the assignment operator. If the initial value of the variable is not set, then it is considered that the variable has the special value `empty`. Value `empty` cannot act as an operand of operations; the only action on a variable with this value is type checking using the `is` operation, see below.

Variable type is *not fixed* during declaration. This means that a variable, within its scope, can take any type of values that are allowed in the language (see “Types”). To check the type of the current value of a variable, there is a special unary operation **is**.

The language defines a common set of common **statements**: except of the declarations, which is also have a status of statements, there are assignment, conditional statement, two kinds of repeat statements and also function return operator and print values.

```
Statement     : Declaration
              | Assignment
              | If
              | Loop
              | Return
              | Print

Assignment    : Reference ':=' Expression

If            : if Expression then Body [ else Body ] end

Loop          : while Expression loop Body end
              | for [ IDENT in ] [ Expression '..'
Expression ]
                  loop Body end

Return        : return [ Expression ]

Print         : print Expression { ',' Expression }
```

Expressions – syntax constructs defining an algorithm for calculating new values. The structure of expressions is traditional and includes several types of operands and a set of infix and prefix operations on operands.

```
Expression    : Relation { ( or | and | xor ) Relation }

Relation      : Factor [ ( '<'|'<='|'>'|'>='|'='|'/=' )
Factor ]

Factor        : Term { [ '+' | '-' ] Term }

Term          : Unary { ( '*' | '/' ) Unary }

Unary         : Reference
              | Reference is TypeIndicator
              | [ '+' | '-' | not ] Primary

Primary       : Literal
              | readInt | readReal | readString
              | FunctionLiteral
              | '(' Expression ')'

TypeIndicator : int | real | bool | string
              | empty     // no type
              | '[' ']'   // vector type
```

```
        | '{' '}'   // tuple type
        | func      // functional type
```

Functions in the D language are considered as literals (similarly to integer/real constants) in the sense that they are considered as constant values that can be assigned to variables and passed as arguments to other functions. The only operation defined for functions is a *call*.

```
FunctionLiteral
        : func [ '(' IDENT { ',' IDENT } ')' ] FunBody

FunBody : is Body end
        | => Expression
```

The `Reference` rule defines *accessors* – constructs that implement elementary actions with variables: taking an array element, calling a function, and accessing elements of objects. Links can be an expression operands, and also (this distinguishes them from other operands - constants, subexpressions, etc.) act as recipients of values in assignment operators.

```
Reference : IDENT
          | Reference '[' Expression ']'
          | Reference '(' Expression { ',' Expression } ')'
          | Reference '.' IDENT
```

Types and literals. The language defines the values of four simple types, two composite types, and one special type. Simple types: integer, real, string, and boolean. Integer literals are written as a sequence of decimal digits. Real literals are formed as two sequences of decimal digits separated by a “dot”. The first sequence denotes the integer part of the real, the second sequence - its fractional part. Strings are specified as a sequence of arbitrary characters enclosed in single or double quotes. Boolean values are represented by two service words **`true`** and **`false`**.

```
Literal    : INTEGER
           | REAL
           | STRING
           | Boolean
           | Tuple      // { a := 5, b := “sss”, 12.34 }
           | Array      // [ 1, 2, 3 ]
           | empty

Boolean    : true | false
```

In addition to simple types, the language includes two composite types: arrays and tuples.

Array – is a linear composition of values of *any type*. The size of the array (number of elements) is not fixed and can be dynamically changed. The array is written as a list of element values, separated by commas and enclosed in square brackets.

```
Array : '[' [ Expression { ',' Expression } ] ']'
```

Access to the elements of the array is performed in the usual way: after the name of the array in square brackets, the serial number of the element in the form of an integer expression is specified. The numbering of the elements of the array begins with one.

Note that an array is a “real” associative array with keys representing integer values. The values of the neighboring keys in the array do not necessarily differ by one. So, for example, the following sequence of operators is possible:

```
var t := [];   // empty array declaration
t[10] := 25;
t[100] := func(x)=>x+1;
t[1000] := {a:=1,b:=2.7};
```

Tuple – it is a collection of named values of arbitrary types. Element names are unique within the tuple. A tuple is specified as a comma-separated list of pairs of the form “name - value”. The name and value are separated by an assignment character. The entire list is enclosed in braces.

```
Tuple : '{' TupleElement { ',' TupleElement } '}'

TupleElement : [ Identifier ‘:=’ ] Expression
```

The structure of tuples, unlike arrays, cannot be modified; the only way to change the composition of the tuple elements is to add another tuple to it:

```
var t := {a:=1, b:=2, c};
t := t + {d:=3};   // now t is {a=1, b=2, c, d=3}
```

Access to the tuple elements is done through dotted notation, which sets the name of the tuple variable and the name of the tuple element:

```
var x := t.b;     // now x is 2
```

Assignment of unnamed elements of a tuple is allowed. Named and unnamed elements can be present in a tuple in any order. Access to unnamed elements of a tuple is done through dotted notation, where the tuple element is identified by its serial number. The first element is always number 1.

Example:

```
var y1 := t.1   // now y1 is 1
var y2 := t.3   // now y2 has the value of c
```

Value Operations. The semantics of operations are generally typical of programming languages and depend on the type of operands (operand). Since the language is dynamic, the control of the types of operands and the selection of the appropriate operation algorithm is performed dynamically during program execution.

Below is a table with a brief description of the operations.

Addition +

Integer + Integer -> Integer

Integer + Real -> Real
Real + Integer -> Real
Real + Real -> Real
String + String -> String (string concatenation)
Tuple + Tuple -> Tuple (tuple concatenation)
Array + Array -> Array (array concatenation)
Other types of operands are not allowed.

Subtraction -

Integer - Integer -> Integer
Integer - Real -> Real
Real - Integer -> Real
Real - Real -> Real
Other types of operands are not allowed.

Multiplication *

Integer * Integer -> Integer
Integer * Real -> Real
Real * Integer -> Real
Real * Real -> Real
Other types of operands are not allowed.

Division /

Integer / Integer -> Integer (round down)
Integer / Real -> Real
Real / Integer -> Real
Real / Real -> Real
Other types of operands are not allowed.

Comparisons <, >, <=, >=, =, /=

Integer *op* Integer -> Boolean
Integer *op* Real -> Boolean
Real *op* Integer -> Boolean
Real *op* Real -> Boolean
Other types of operands are not allowed.

Unary addition and subtraction + -

op Integer -> Integer
op Real -> Real
Other types of operands are not allowed.

Logical operations and, or xor, not

Both operands must be Boolean. The result of operations is always Boolean.
Other types of operands are not allowed.

3. Grammar

```
Program      : { Declaration }

Declaration  : var Identifier [ := Expression ] ;

Expression   : Relation [ ( and | or | xor ) Relation ]

Relation     : Factor [ ( < | <= | > | >= | = | /= ) Factor ]

Factor       : Term { ( + | - ) Term }

Term         : Unary { ( * | / ) Unary }

Unary        : [ + | - | not ] Primary [ is TypeIndicator ]
             | Literal
             | ( Expression )

Primary      : Identifier { Tail }
             | readInt | readReal | readString

Tail         : . IntegerLiteral   // access to unnamed tuple element
             | . Identifier       // access to named tuple element
             | [ Expression ]     // access to array element
             | ( Expression { , Expression } )  // function call

Statement    : { Assignment | Print | Return | If | Loop }

Assignment   : Primary := Expression

Print        : print Expression { , Expression }

Return       : return [ Expression ]

If           : if Expression then Body [ else Body ] end

Loop         : while Expression LoopBody
             | for Identifier in TypeIndicator LoopBody

LoopBody     : loop Body end

TypeIndicator
             : int | real | bool | string
             | empty           // no type
             | [ ]             // vector type
             | { }             // tuple type
             | func            // functional type
             | Expression .. Expression

Literal      : IntegerLiteral   // 1, 12345, 777
             | RealLiteral      // 1.23
             | BooleanLiteral   // true or false
             | StringLiteral    // "string", 'string‘
             | ArrayLiteral
             | TupleLiteral

ArrayLiteral : [ [ Expression { , Expression } ] ]

TupleLiteral : { [ [ Identifier := ] Expression
                        { , [ Identifier := ] Expression } ] }

FunctionLiteral
             : func [ Parameters ] FunBody
```

```
Parameters    : ( Identifier { , Identifier } )
FunBody       : is Body end
              | => Expression
Body          : { Declaration | Statement | Expression }
```

4. Semantics of basic operations

Sign	Operand1	Operand2	Result	Semantics
+	Integer	Integer	Integer	Algebraic addition
	Integer	Real	Real	
	Real	Integer	Real	
	Real	Real	Real	
	Vector	Integer	Vector	Adding element to vector
	Vector	Real	Vector	
	Vector	Vector	Vector	Joining vectors
	List	Integer	List	Appending element to list
	List	Real	List	
	List	List	List	Joining lists
+=	Integer	Integer	Integer	
	Tuple	Tuple	Tuple	Memberwise addition
-	Integer	Integer	Integer	Algebraic subtraction
	Integer	Real	Real	
	Real	Integer	Real	
	Real	Real	Real	

5. Structure of a D-processor