Project D: Dynamic Language

1. Common Description

- Dynamic typing: object types are not specified and can change while program execution.
- The language assumes **interpretation.**
- Major notion: variable & literal (constant).
- Program structure: a sequence of declarations and statements.
- Builtin types:
  integer, real, boolean, string
  User-defined types: array, tuple, function.
- Implicit type conversions are supported.
- Statements: assignment, if/while/loop, exit/return, output.

2. Informal language specification

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

```
Program : { Statement [ ; ] }
```

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

There are two category of constructs: variable **declaration** and **statement**.

Declaration of the variable introduces a new named object to the current scope, perhaps with an initial value. Statements define a specific action, whose semantics could consist of calculation of new value for some variable (assignment), or 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 **`none`**. 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.

The type of the variable 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 statement and the print statement.

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

Assignment
    : Reference := Expression

If
    : if Expression then Body [ else Body ] end

IfShort
    : if Expression => Body

Body
    : Statement { [ ; ] Statement }

Loop
    : WhileLoop
    | ForLoop

WhileLoop
    : while Expression LoopBody

ForLoop
    : [ ForHeader ] LoopBody

ForHeader
    : for [ IDENT in ] Expression [ .. Expression ]

LoopBody
    : loop Body end

Exit
    : exit
```

Notice that the `ForHeader` part in the for-loop can be omitted. In that case, the loop is infinite. The only way for exiting such a loop is exit statement.

The case with two `Expressions` in the loop header means that the loop iterates over the range from *v1* until *v2*, where *v1* and *v2* are current integer values of the corresponding expressions. If the identifier is specified in the loop header then it introduced a variable that takes values from the range sequentially on each loop iteration. The scope of the variable is the loop body.

If the single Expression is specified in the loop header then its type should be either an array or a tuple. In that case, the loop iterates over all elements of the corresponding aggregate.

The exit statement can appear only within loop statements.

Examples:

```
var i := 0
loop // Infinite loop
    print "Hello"
    i := i + 1
    if i=100 => exit
end

for 1..3 loop
    print "Hello"
end

var array := [1,2,3,4,5]
var sum := 0
for i in array loop
    sum := sum+i
end
```

Return statement performs leaving a function perhaps with returning a result value to the calling context.

```
Return
    : return [ Expression ]

Print
    : print Expression { , Expression }
```

Expressions

Expression is a 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
    | FunctionLiteral
    | ( Expression )

TypeIndicator
    : int
    | real
```

```
            | bool
            | string
            | none      // no type
            | [ ]       // vector type
            | { }       // tuple type
            | func      // functional type
```

Functions in the D language are treated 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
        Literal
            : IntegerLiteral
            | RealLiteral
            | BooleanLiteral
```

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

```
        Reference
            : IDENT
            | Reference [ Expression ]   // array element
            | Reference ( Expression { , Expression } ) // call
            | Reference . IDENT             // tuple element
            | Reference . IntegerLiteral    // tuple element
```

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 the dot character. 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 keywords `true` and `false`.

```
        Literal
            : INTEGER
            | REAL
            | STRING
            | Boolean
            | Tuple    // { a := 5, b := "sss", 12.34 }
            | Array    // [ 1, 2, 3 ]
            | none
```

```
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 presented 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 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 is a fixed-size collection of possibly named values of arbitrary types. Element names, if they are specified, 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 the assignment token **:=**. 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, for example:

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

Access to the tuple elements is performed via dot notation, using the name of the tuple variable and the name of the tuple element, or by using the number of the tuple element within the tuple (numbering starts with one), for example:

```
var x := t.b; // now x is 2
x := t.2;     // the same effect
var y2 := t.3 // now y2 has the value of c+d
```

Named and unnamed elements in a tuple can follow in any order.

Operator semantics

The semantics of expressions 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 plus and minus: +, -
```
op Integer -> Integer
op Real -> Real
```
Other types of operands are not allowed.

Logical operations: and, or, xor

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

Unary negation: not

The operand must be Boolean. The result of the operator is boolean. Other types of the operand are not allowed.