Project I
Imperative Language

Common Program Structure. Entity declarations

Program text is considered as a sequence of Unicode characters. The structure of the source text corresponds to syntax rules that are presented below. The source program text is located in a single disk file.

```
Program
    : { SimpleDeclaration | RoutineDeclaration }
```

Syntactically, a program is a sequence of declarations of variables, types and subprograms (possibly empty). All entities used in the program should be declared textually before their first use. There is no support for attaching external component to programs (no “importing”).

The program starts running from invoking a subprogram. The name of the starting subprogram and its arguments (if any) are passed to the program while launching. This means that a program can start its execution from any subprogram.

Declarations are separated from each other by “new line” characters or by semicolons. (Notice that separators are not shown in grammar rules for better readability.)

The language defines three categories of program entities: **variables**, **types** and **subprograms**. This makes the language compact, easy to learn and use.

The scope of an entity starts from the point of its declarations and spans to the textual end of the scope (block) where it is declared – that is, up to the area that is defined by the `Body` grammar rule, or, for topmost entities – to the textual end of the program. If there is a nested scope, then the declaration of entity within the nested scope hides (“shadows”) the entity declaration with the same name in an outer scope.

This rule applies uniformly to all kinds of entities: variables, types and subprograms.

```
SimpleDeclaration
    : VariableDeclaration
    | TypeDeclaration
```

Simple declarations include variable and type declarations. They can be specified in any scope (including subprogram bodies and compound statements). This differs simple declarations from subprogram declarations: the latter can be declared in the topmost level only. In other words, nested subprograms are not supported.

Variable declarations

All variables used in a program should be declared. The variable declaration should textually precede its first use.

```
VariableDeclaration
    : var Identifier : Type [ is Expression ]
    | var Identifier is Expression
```

The variable declaration includes:

a) The name of the variable (identifier); it is used to refer to the variable’s value

b) The type of the variable. It determines the set of values and the set of operators on values

c) The initial value of the variable.

If the initial value is specified in the declaration then the type can be omitted. In such a case, the type can be unambiguously deduced (“inferred”) from the expression that specifies the initial value. If no initializer in the variable declaration is specified then the type specifier is mandatory.

Type declarations

```
TypeDeclaration
    : type Identifier is Type
```

Type declaration introduces a short equivalent (“name”) for some type. After such a type declaration, the identifier following the keyword `type` is considered as a synonym for the type specified after the `is` keyword. The name introduced by such a declaration can be used everywhere in its scope.

```
Type
    : PrimitiveType
    | UserType
    | Identifier
```

The language supports two categories of types: predefined and user-defined types.

Predefined types

```
PrimitiveType
    : integer | real | boolean
```

There are three predefined types: integer, real and boolean. Integer type denotes the set of (positive and negative) integer values. Real type denotes the set of (positive and negative) real values. The ranges of those types (i.e., their minimal and maximal values) are not defined in the language; each implementation should define its own restrictions for those types as well as their internal representation format.

Boolean type includes two values: true and false together with conventional set of operators: logical “and”, logical “or”, logical “xor”, and negation. Boolean values are represented in programs by keywords `true` and `false`.

User-defined types

The language introduces two kinds of user-defined types: **records** and **arrays**.

User-defined types are considered as “reference types”. This means that a variable of a user-defined type is actually the reference to an object of the type. So, the variable serves as a “representative” of the object. Therefore, any assignment to a variable of an array or record type means copying the reference to the aggregate but not copying of the aggregate itself.

```
UserType
    : ArrayType | RecordType
```

Record type is for grouping logically related variables into a single construct. An arbitrary numbers of variables (“members”) can be grouped into a record. Record members can have any type; arrays and nested records are also permitted. Conventional dotted notation is used for accessing record members.

```
RecordType
    : record { VariableDeclaration } end
```

Array type is for grouping variables (“array elements”) of the same type together. The array size (i.e., the number of array elements) is specified in the array type declaration as a compile-time constant (using syntax of expression).

```
ArrayType
    : array [ [ Expression ] ] Type
```

Array elements are numbered by integer values starting with 1. Access to array elements is organized in a conventional way, by indexing: the index (number) of the array element enclosed by square brackets following the name of the array. The element index is specified dynamically (using the syntax of expression). This allows organizing array processing in loops.

Array size can be omitted in the array type declaration. Sizeless declaration can be used only for specifying subroutine parameters. In that case, the information about actual size of the argument array size is accessible by dotted notation of a special kind (see below).

Statements

The language defines a minimal set of conventional statements that perform some actions on program objects (variables).

```
Statement
    : Assignment
    | RoutineCall
    | WhileLoop
    | ForLoop
    | IfStatement
    | PrintStatement
```

Assignment performs evaluating the value of the expression specified in the right part of the statement, and copying it to the variable whose name is specified in the left part.

```
Assignment
     : ModifiablePrimary := Expression
```

Types of the name from the left part should conform the type of the expression. For predefined types, the conformance is quite obvious and natural, and is defined by the following table:

Left part type	Right part type	Assignment semantics
integer	integer	direct value copying
integer	real	copying with rounding to nearest integer
integer	boolean	`true` is converted to integer 1, `false` is converted to integer 0
real	real	direct value copying
real	integer	direct value copying
real	boolean	`true` is converted to real 1.0 `false` is converted to to real 0.0

`boolean`	`boolean`	direct value copying
`boolean`	`integer`	if integer is 1, it is converted to `true`, if it is 0, it's converted to `false`; otherwise, assignment is treated as erroneous.
`boolean`	`real`	assignment is illegal

For variables of user-defined types, the conformance means the full name identity of the types from left and right parts of the assignment. In other words, both parts should have the type that was previously introduces by the same type declaration.

Notice that user-defined types are reference types. This means the assignment actually copies the reference to a variable, but not the object itself. Therefore, it's possible to have multiple references to the same object of a user-defined type.

Notice also that passing arguments to subroutines is treated as assignment, and the same rules apply for passing arguments.

```
RoutineCall
    : Identifier [ ( Expression { , Expression } ) ]
```

The `RoutineCall` specifies the call (invocation) of a subroutine that is denoted by the `Identifier`. If the subroutine being activated is declared with parameters, the corresponding routine call should contain the list of arguments. Arguments are expressions separated by commas and enclosed in parentheses. The number of arguments in the list should be equal to the number of parameters in the declaration, and argument types should conform the types of parameters (see "Assignment" section above).

The `RoutineCall` statement can invoke a subroutine that returns a value ("function"). In that case, the returning value is discarded.

```
WhileLoop
    : while Expression loop Body end
```

While-loop is actually a loop with a precondition. It specifies repeated execution of statements from its `Body`. The expression from the loop header should be of a type that conforms **`boolean`**.

The while-loop execution starts with evaluating the expression from its header. If the value is **`true`** then the statements from the loop body are executes, and the control flow goes back to the expression evaluation and checking its value. If the value of the expression (on the very first or on any following iteration) is **`false`**, then the loop statement exits. Such semantics assumes that the loop body can be executed zero or more times.

```
ForLoop
    : for Identifier in Range [ reverse ] loop Body end

Range
    : Expression [ .. Expression ]
```

The for-loop also implements repeated execution of its body. However, in for-loops the number of iterations is specified explicitly. On each iteration the next value from within the `Range` is taken and assigned to the loop variable specified as `Identifier`. The very first value is evaluated from the first `Expression` (or from the last `Expression` if the keyword **`reverse`** is specified). Before each iteration

after the very first, the value of the loop variable gets increased (or decreased if keyword **reverse** is specified) by one. If the current value exceeds the value of the last Expression (or is less than the value of the first Expression if the keyword **reverse** is specified), then the execution of the for-loop exits.

The loop variable is treated as implicitly declared in the scope of the loop, and its type is **integer**. After completing the loop this variable (as well as all other entities declared within the loop body) disappears. Assignments to the loop variable are prohibited (in other words, the loop variable is read-only).

Types of expressions in the Range should conform the integer type. If the value of the first expression is greater than the value from the second expression, the loop body is executed zero times. The values of both expression from the Range are evaluated once before the very first iteration.

If the second Expression in the loop header is omitted then the first Expression should denote an array. In that case, the loop iterates over all elements of the array, and on each iteration the loop variable takes the value of the current array element, starting with its very first element, or starting from the last element, if the **reverse** keyword is specified.

For loops are conventionally used for processing arrays and working with array elements.

```
IfStatement
    : if Expression then Body [ else Body ] end
```

The conditional statement specifies the choice: to execute one sequence of statements or another. Choosing the sequence is made by evaluating the expression from the if header. The type of the expression should conform boolean type. If the value of the expression equals **true**, then the first sequence (specified after the keyword **then**) is executed: otherwise (i.e., if the value of the expression is **false**) the second sequence (i.e., after the keyword **else**, if any) is executed.

```
PrintStatement
    : print Expression { , Expression }
```

Routine declarations

Routine declaration introduces the conventional procedure (or function, or subprogram etc.) to the current context. Notice that the language supports only “global” routines; that is, nested routines are not supported.

```
RoutineDeclaration
    : RoutineHeader [ RoutineBody ]

RoutineHeader
    : routine Identifier ( Parameters ) [ : Type ]

RoutineBody
    : is Body end
    | => Expression

Parameters
    : ParameterDeclaration { , ParameterDeclaration }

ParameterDeclaration
    : Identifier : Type

Body
    : { SimpleDeclaration | Statement }
```

Parameter passing by definition has the same semantics as usual assignment (see “Statements” section above).

If the type after the parameters section is specified then the function returns a value; therefore, its call can present in expressions. In that case, the value to be returned by the routine should specified in the routine body by the special return statement.

The `RoutineBody` can be omitted in the `RoutineDeclaration`. In that case, this is the **forward routine declaration**. If the routine is declared as forward, then it could be called somewhere below from another routine. However, the full routine declaration (with `RoutineBody`) should be specified somewhere in the program with the same name and with the same signature.

Forwarding is sometimes used in case a routine calls another routine which is not declared yet.

Expressions

Expression are formulas for calculating values. Expressions are constructed following conventional expression rules in accordance with operator preferences (“priorities”).

```
Expression
    : Relation { ( and | or | xor ) Relation }
Relation
    : Simple [ ( < | <= | > | >= | = | /= ) Simple ]
Simple
    : Factor { ( * | / | % ) Factor }
Factor
    : Summand { ( + | - ) Summand }
Summand
    : Primary | ( Expression )
Primary
    : [ Sign | not ] IntegerLiteral
    | [ Sign ] RealLiteral
    | true
    | false
    | ModifiablePrimary
    | RoutineCall
Sign
    : + | -
```

The `ModifiablePrimary` grammar rule defines possible constructs in the left parts of assignments. The rule determines that only simple variables, array elements and record members can be specified in the left parts.

```
ModifiablePrimary
    : Identifier { . Identifier | [ Expression ] }
```