Project O
Object-oriented language

1. Informal language specification O

An O program is a sequence of class declarations (possibly empty). The entry point to the program, by definition, can be the constructor of any of its class. The name of the initial class is sets when the program is activated, together with the arguments of the constructor (if necessary).

```
Program
    : { ClassDeclaration }
```

The semantics of program initialization is as follows. Using the class constructor with the parameters corresponding to the arguments passed, an unnamed object of this class is created, the arguments are assigned from the launch command line to the corresponding constructor parameters, after which control is transferred to the constructor body. The completion of this body means the completion of the entire program.

```
ClassDeclaration
    : class ClassName [ extends ClassName ]
            is { MemberDeclaration } end
```

The concept of a class is the fundamental (and, in fact, the only) concept of the type system of the O language. In other words, the only way to specify the types of objects (variables) in the O language is to define a certain class. Thus, we can assume (similarly to other object-oriented languages) that "**a class is a type**".

In the simplest case, a class is a collection of logically related resources — simple class *members* and class *methods*. The simple members of a class collectively determine the current state of a class instance; class methods specify the behavior of the objects of this class. The class can also specify "special" methods used to create objects (instances) of the class – *constructors*.

For classes, *inheritance* relationship is defined. This means that a certain class may include, along with its own (defined in the class itself) entities, also the entities of some other class, which in this case is called the *base class* of this class. A class that inherits the properties of a base class is called a *derived class*. Thus, it is possible to build hierarchies of class types in which the properties of a derived class are a composition of their own properties and the properties of the base class.

The inheritance relation makes it possible to work with objects of derived classes as with some "extensions" of objects of base classes. In particular, objects of base classes can refer to objects whose type is a derived class.

The inheritance relation is transitive: if some class A is base for class B, and class B is base for class C, then class A is also considered base class for class C. The inheritance relation determines the polymorphic behavior of class methods. Polymorphism can be defined as follows: a method of a derived class whose signature matches with the method signature from the base class *overrides* the method of the base class. In other words, during the execution of the program, the choice of the particular method being called is determined by the dynamic type of the object.

```
ClassName
     : Identifier // [ [ ClassName ] ]
```

/*****

The following language feature is not a subject for implementation within the Compiler Construction course:

The language defines the concept of generalized classes ("generics"). This means that a class declaration can specify one or more "abstract" types (parameter types) that can be used to specify class variables and methods. An obvious example of a generalized class is an array in which the type of elements is not fixed and is set as "abstract".

The use of a generalized class is possible after its instantiation, in which the names of real types are set for all abstract parameter types (in the form of a list of types separated by commas and enclosed in square brackets). When processing a generalized class, the compiler implements the substitution of real types instead of abstract types and generates a real ("customized") class for the substitution result.

*****/

```
MemberDeclaration
    : VariableDeclaration
    | MethodDeclaration
    | ConstructorDeclaration
```

As already mentioned, a class is defined as a set of member variables of a class, class methods, and special class constructor methods.

Class member variables are considered read-only accessible from outside the class; It is not possible to directly change the current value of a member variable (for example, through assignment). Access to member variables is allowed only from class methods (or from methods of derived classes).

```
VariableDeclaration
    : var Identifier : Expression
```

Each entity in the program should be declared. The relative order of declaration of a variable and its use is described below. The syntax and semantics of variable declarations is somewhat different from the usual form. To declare a variable, you should specify two required components: the name of the variable and its initial value. The O language is typed, so the type of the declared variable is uniquely determined from the type of the initializing expression.

```
MethodDeclaration
    : MethodHeader [ MethodBody ]

MethodHeader
    : method Identifier [ Parameters ] [ : Identifier ]

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

Class methods determine all aspects of the external behavior of the class and the scheme for changing its state. Access to class member variables for modifying their values is only possible through class methods.

Method declaration begins with a keyword **method**, after which the identifier of the method name is introduced and, possibly, a list of method parameters enclosed in parentheses. If the method returns some value (and, therefore, its call can be a part of expressions), then the type of the return value should follow the parameters of the method. Next, after the keyword **is** the sequence of operators of the method body should be located. The body of the method ends with a keyword **end**.

For small methods, whose body is the single return statement, a short version of the method body is possible: just the single expression after the **=>** delimiter.

The MethodBody can be omitted in the MethodDeclaration. 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 MethodBody) 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.

In the class, several methods of the same name can be declared. All methods with the same name should differ in the number and types of parameters. The choice of a particular method to be called is determined by the compiler based on a comparison of the call arguments and method parameters with the given name.

```
Parameters
    : ( ParameterDeclaration { , ParameterDeclaration } )

ParameterDeclaration
    : Identifier : ClassName

Body
    : { VariableDeclaration | Statement }

ConstructorDeclaration
    : this [ Parameters ] is Body end

Statement
    : Assignment
    | WhileLoop
    | IfStatement
    | ReturnStatement
```

The language defines the minimal set of operators necessary for real programming: assignment, loop, conditional operator, and return operator from a method.

```
Assignment
    : Identifier := Expression

WhileLoop
    : while Expression loop Body end
```

The language defines the only (most general) form of the loop operator: a conditional loop. The body of the loop is executed as long as the value of the expression after the keyword **while** is **true**. The expression in the condition must be of boolean type and evaluated before each iteration of the loop. Thus, a cycle can be executed zero or more times.

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

Conditional operator makes the execution of a certain sequence of statements dependent on the fulfillment of a certain condition. The condition is specified as an expression, which must be of boolean type, after the keyword **if**. If the value of the expression is **true**, then the sequence of statements specified after the **then** keyword is executed, otherwise, the sequence of statements after the **else** keyword (if this part of the statement exists).

The **end** keyword ending the conditional statement is used to fix the textual completion of the statement and prevents possible ambiguities in the case of nested conditional statements.

```
ReturnStatement
    : return [ Expression ]
```

The return statement implements the completion of the method and the return of control to the context of the calling method. If the method with the return statement defines the type of the return value, then the statement must contain an expression defining the value to be returned.

```
Expression
    : Primary
    | ConstructorInvokation
    | FunctionCall
    | Expression { . Expression }

ConstructorInvokation
    : ClassName [ Arguments ]

FunctionCall
    : Expression [ Arguments ]

Arguments
    : ( )
    | ( Expression { , Expression } )
```

Expressions are intended to specify how values are calculated. The structure of expressions in the O language is much simpler than in many other languages and contains essentially two basic constructions - access to a class member and method invocation.

ConstructorInvokation specifies creation of objects (instances of classes). The class name is either just an identifier – for the case of a simple (non-generic) class, or the identifier of a generalized class, after which the real types with which the generalized class is configured are indicated in square brackets. Types inside square brackets are separated by commas.

Access to a class member (variable or method) is represented using dotted notation of the form *object-name*.*member-name*. A method call is formed in the traditional way: the name of the method (which, in turn, can be specified as a dotted notation), after which a comma-separated list of method arguments can follow in parentheses. Since a method call can return some object (variable), the call construct can serve as the left part of the member access construct. Thus, compound structures formed by a superposition of calls and / or accesses to members are allowed.

```
Primary
    : IntegerLiteral
    | RealLiteral
    | BooleanLiteral
    | this
```

The basic components of expressions are primary. These are either literals of library types (their lexical syntax is up to implementers; it should be very simple and easyto-read), or the keyword `this`, which in the body of the method denotes the object (variable) for which the method is called.

The O language defines a small number of library (predefined) classes. Using these classes in programs does not require any explicit ways to connect them. In other words, library class declarations are considered to be present in any program by default.

Implementation details of library classes in a language are not defined. An informal description of the semantics of these classes is given below.

2. Standard O language classes

The complete hierarchy of library classes is as follows:

```
class Class is ... end
   class AnyValue extends Class is ... end
      class Integer extends AnyValue is ... end
      class Real extends AnyValue is ... end
      class Boolean extends AnyValue is ... end
   class AnyRef extends Class is ... end
      class Array extends AnyRef is ... end
      class List extends AnyRef is ... end
```

Library classes `Integer`, `Real` and `Boolean` implement conventional objects of integer, real, and Boolean types, respectively. Literals of the listed types can be specified in the traditional way.

For these library classes, methods are defined that implement the operations traditional for these types of operations — arithmetic, logical, and relational operations, as well as type conversion operations. All these operations are presented **in the form of appropriate methods**. Thus, the traditional infix notation of expressions is not supported in the language. (Perhaps in future versions of the language, the familiar form of writing expressions will be also implemented.)

```
class Integer extends AnyValue is
    // Constructors
    this(p: Integer)
    this(p: Real)

    // Features
    var Min : Integer
    var Max : Integer

    // Conversions
    method toReal : Real
    method toBoolean : Boolean

    // Unary operators
    method UnaryMinus : Integer

    // Integer binary arithmetics
    method Plus(p:Integer) : Integer
    method Plus(p:Real) : Real
    method Minus(p: Integer) : Integer
    method Minus(p: Real) : Real
    method Mult(p: Integer) : Integer
    method Mult(p: Real) : Real
    method Div(p: Integer) : Integer
    method Div(p: Real) : Real
    method Rem(p: Integer) : Integer
    // Relations
    method Less(p: Integer) : Boolean
    method Less(p: Real) : Boolean
    method LessEqual(p: Integer) : Boolean
    method LessEqual(p: Real) : Boolean
    method Greater(p: Integer) : Boolean
    method Greater(p: Real) : Boolean
    method GreaterEqual(p: Integer) : Boolean
    method GreaterEqual(p: Real) : Boolean
    method Equal(p: Integer) : Boolean
    method Equal(p: Real) : Boolean
end

class Real extends AnyValue is
    // Constructors
    this(p: Real)
    this(p: Integer)

    // Features
    var Min : Real
    var Max : Real
    var Epsilon : Real

    // Conversions
    method toInteger : Integer

    // Unary operators
    method UnaryMinus : Real

    // Real binary arithmetics
    method Plus(p:Real) : Real
    method Plus(p:Integer) : Real
    method Minus(p: Real) : Real
    method Minus(p: Integer) : Real
    method Mult(p: Real) : Real
    method Mult(p: Integer) : Real
    method Div(p: Integer) : Real
    method Div(p: Real) : Real
    method Rem(p: Integer) : Real

    // Relations
    method Less(p: Real) : Boolean
    method Less(p: Integer) : Boolean
    method LessEqual(p: Real) : Boolean
    method LessEqual(p: Integer) : Boolean
    method Greater(p: Real) : Boolean
    method Greater(p: Integer) : Boolean
    method GreaterEqual(p: Real) : Boolean
    method GreaterEqual(p: Integer) : Boolean
    method Equal(p: Real) : Boolean
    method Equal(p: Integer) : Boolean
end

class Boolean extends AnyValue is
    // Constructor
    this(Boolean)

    // Conversion
    method toInteger() : Integer

    // Boolean operators
    method Or(p: Boolean) : Boolean
    method And(p: Boolean) : Boolean
    method Xor(p: Boolean) : Boolean
    method Not : Boolean end

class Array[T] extends AnyRef is
    // Constructor
    this(l: Integer)

    // Conversion
    method toList : List

    // Features
    method Length : Integer

    // Access to elements
    method get(i: Integer) : T
    method set(i: Integer, v: T)
end

class List[T] extends AnyRef is
    // Constructors
    this()
    this(p: T)
    this(p: T, count: Integer)

    // Operators on lists
    method append(v: T) : List
    method head() : T
    method tail() : List
    ....
end
```

3. Examples of O language constructs

```
var a : Array[Integer](10)
a.set(i) := 55
x := a.get(i.Plus(1))

var i is 1
while i.LessEqual(a.Size) loop
    x := a.get(i)
    a.set(i,x.Mult(x))
    i := i.Plus(1)
end

class C[T] is
    var m : T
end

var y : Array[Integer](10)
var k : List[Real]

var m : Integer(1)

var b : true
var b : Boolean(true)

var x : Base(1,2)
x := Derived(3)

method MaxInt(a: Array[Integer]) : Integer is
    var max : Integer.Min
    var i : Integer(1)
    while i.Less(a.Length) loop
        if a.get(i).Greater(max) then
```

```
            max := get(i)
        end
        i := i.Plus(1)
    end
    return max
end
```