# Grammar

**In actual use, please use "e" instead of "exp" and "re" instead of "return"**

## FUNCTION

## DEFINE FUNCTION

In COOL, a function consists of a function declaration and a body of functions. A function representing addition can be defined as code 1:

|  |
| --- |
| **CODE 1** Example function declaration |
|  |
|  |
|  |

Where '@' modifies the subsequent as a function declaration rather than a function call. You need to remove '@' when calling a function:

|  |
| --- |
| **CODE 2** Function call example |
|  |

Functions are called with preference for passing in references to actual parameters rather than copying, for example, code 3:

|  |
| --- |
| **CODE 3** Example function call |
|  |
|  |
|  |
|  |

Increase by 1 after code 3 is executed. In this code, function parameters are allowed to be embedded in function name (predicate) string to enhance expressiveness.

In addition, COOL provides built-in functions that can be used without definition to provide basic mathematical operations.

## FUNTION RETURNING EXPRESSION and FUNCTION RETURNING VALUE

The attribute "exp" is added to the function declaration to indicate that the return of this function is an expression. This kind of function is used to express the transformation rules of the expression. Users cannot directly call the function returning expression.

For example, code 4 describes the inverse operation of the distributive law of multiplication by function:

|  |
| --- |
| **CODE 4** Example of function returning expression |
|  |
|  |
|  |

The pair of curly braces immediately following "@" and the expression within them are called function declaration scope, and the only internal expression is called function declaration expression. In fact, in the function example in section 1.1.1, the function name is also in its function declaration scope, but the curly brackets on the scope boundary are omitted for writing convenience.

If the function declaration has no attribute "exp", the function returns the computed value. The user can only call value returning function directly.

## FORWARD FUNCTION and REVERSE (BACKWARD) FUNCTION

Both forward and reverse functions are for value returning functions; Functions returning expression do not have this property.

Forward function refers to the function whose parameters are determined and whose return value is undetermined. The execution process of forward function is to deduce the return value of the function by using the input parameters, such as code 1.

Reverse function refers to the function whose return value is determined and whose input parameters have pending parameters (parameters with undetermined values). The execution process of reverse function is to use the return value of the function and the determined input parameters to calculate the pending input parameters.

For example, code 5 provides the reverse function required to calculate a solution of a quadratic function:

|  |
| --- |
| **CODE 5** Example reverse function |
|  |
|  |
|  |
|  |
|  |
|  |

The "$" symbol acts on parameter x, indicating that parameter x is pending in the expression. For variables that appear multiple times in an expression, you only need to use "$" to decorate them once. The return value of the function represented by the variable "ans" appearing in the function body is a known parameter, which you can use or not use.

When calling the reverse function, the user needs to modify the undetermined variable to be derived with "$", for example, solve the univariate quadratic equation with as the unknown number (code 6):

|  |
| --- |
| **CODE 6** Example for calling reverse function |
|  |

needs to be decorated with "$".

## FUNCTION WEIGHT

The function weight is used to control the reasoning direction of the reasoning system. Users can give greater weight to transformations (i.e. functions) that are easier to lead to correct reasoning results, and less weight to transformations that are not easy to lead to correct reasoning results. The weight of a function is determined at the time of function declaration, between "@" and the scope of function declaration. For example, code 7 defines a function with a weight of 10:

|  |
| --- |
| **CODE 7** Examples of function with weight |
|  |
|  |
|  |

A function that is not weighted (such as code 1) has a weight of 0.

## FUNCTION INVERSION

Function backstepping is a special way to define reverse function through forward function. Example code 8:

|  |
| --- |
| **CODE 8** Example of using forward function to define reverse function |
|  |
|  |
|  |

Where "=>" indicates derivation. The necessary conditions for using this method to define the reverse function in the algorithm to realize this function in this paper are: the names of all parameters in the function declaration of the reverse function must correspond to the names of all parameters in the function declaration of the forward function one by one; The undetermined parameter in the reverse function and the variables that depend on this parameter only participate in the sequential structure of the forward function, and do not participate in the loop and branch structure of the function body; The function body code of forward function does not modify the parameters outside the scope; Variables with the same name and different scopes do not exist in the function body of forward function. Since it is a necessary condition, it also means that even if these requirements are met, the backstepping may not be completed for other reasons.

## VARIABLE

## DECLARATION and TYPE of VARIABLE

Non temporary variables of COOL must be declared before use. An example of declaring variable and assigning an initial value of 1 is shown in code 9:

|  |
| --- |
| **CODE 9** Example of variable declaration |
|  |

The type of the variable is determined by the last assigned type. The basic data types supported by COOL are floating point numbers and strings; When a variable is not assigned a value, its type defaults to floating number.

## ACCESS to VARIABLES

Variables can be accessed from the point where they are declared to the end of their scope. When an expression needs to use a variable, it takes precedence over the variable in the current scope. Users can use "out" to modify this variable to force the expression to use variables from the upper scope. For example, code 10:

|  |
| --- |
| **CODE 10** "out" usage example |
|  |
|  |
|  |
|  |
|  |

The result of calculation (the final value of in the inner scope) is 2.

When the "out" modifier parameter is used in a function declaration scope, it is no longer a formal parameter but an actual parameter outside the scope of the function declaration. For example, in the function declaration of code 11:

|  |
| --- |
| **CODE 11** Example use of "out" in a function declaration |
|  |
|  |
|  |
|  |
|  |

## CONDITIONAL STATEMENT

## LOOP

COOL supports “while” loop structure, such as code 12:

|  |
| --- |
| **CODE 12** “while” loop example |
|  |
|  |
|  |

## BRANCH

Branch structure of COOL, example code 13:

|  |
| --- |
| **CODE 13** Branch structure example |
|  |
|  |
|  |
|  |
|  |
|  |
|  |

## COMMENT STATEMENT

The comment style is the same as C, for example, code 14:

|  |
| --- |
| **CODE 14** Example of comment |
|  |
|  |

## COMPLETE CODE EXAMPLE 1

Solve the following mathematical problems:

For two numeric quantities and , do the following in turn:

1. Sum and and record the result as ;

2. Modify the value of so that it satisfies the value of equal to ;

3. Solve the variable , where equals 100;

4. Sum , , to get the final result;

Given that the result from the fourth step is 50 and that has an initial value of 3, what is the initial value of ?

The complete COOL code for solving the problem, such as code 15, involves function returning expression and value returning function, forward and reverse functions, function weights, and function inverse.

|  |
| --- |
| **CODE 15** Complete code example 1 |
|  |
|  |
|  |
|  |
| *Defines the law of additive exchange, and the function declaration is modified by attribute ”exp”, indicating that this function is a function returning expression. For the function returning expression, the function parameters and b are modified by” #”, indicating that the law of additive exchange can be applied regardless of whether or is undetermined.* |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
| “”  *\*/* |

## CLASS

Rules for solving similar problems can be encapsulated into classes. Users can reuse, modify and expand rules more flexibly through inheritance, so as to realize the division and treatment of complex problems and the modular development of programs.

## DEFINE CLASS

In COOL, a class consists of a declaration and the scope of the class (hereinafter referred to as the "class body"), such as code 16:

|  |
| --- |
| **CODE 16** Example of class |
|  |
|  |
|  |

Where, "system" is the keyword of the declaration of class; "Operationlaw" is the name of the class.

## INHERIT

When defining a class, you can make it inherit from other classes to use its member functions and variables:

|  |
| --- |
| **CODE 17** Class inheritance example |
| *,* |
|  |
|  |

Where "<<" means inheritance. When a class inherits multiple classes, the names of the inherited classes are separated by commas. If a variable with the same name or a function with the same declaration in the parent class exists in the current class, the member variable or function of the current class will be used first by default; If multiple parent classes have the same member, the member of the class on the left at the time of declaration is preferred (for example, if the class has the same member as the class , the member of is preferred); If the name of the class is specified, the member function of the specified class is used.

## CLASS INSTANCE INITIALIZATION

An instance of a class also belongs to a variable. The way to declare it is shown in code 18:

|  |
| --- |
| **CODE 18** Class instance declaration example |
|  |

Its initialization is similar to a function call. After entering the scope of the class, it creates an active record and executes the code in the scope of the class in turn. However, when leaving the scope of the class, it does not destroy the active record, but takes the active record as the value of the corresponding variable.

## ACCESS MEMBERS

By "." Operator to access member variables or call member functions, such as code 19:

|  |
| --- |
| **CODE 19** Example of access members |
|  |
|  |

## COMPLETE CODE EXAMPLE 2

Code 20 shows the combined use of cool's classes.

First, two classes are defined. The class contains the transformation rules of some common operations. The class contains a formula for solving the univariate quadratic equation. Then, the main program class inherits the two classes previously defined, and defines two functions in the program to solve the univariate quadratic equation and modify the member variables:

|  |
| --- |
| **CODE 20** Complete code example 2 |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |