COOL SYNTAX

This section briefly introduces the syntax of COOL, including functions, variables, control structures, and classes through examples. This is an early version created in 2022, and is modified in 2024.

1 FUNCTION STRUCTURE

Functions are the most important part of COOL. This section first provides some simple code to familiarize readers with the style of COOL functions, then introduces functions that return expressions and return operands, functions with additional user prompts, rule and fact functions, forward and inverse functions. These constitute the COOL inference framework.

1.1 FUNCTION DEFINITION

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

CODE 1 Example function declaration

```
@add(a, b){
    return: a + b;
}
```

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

CODE 2 Function call example

```
add(1,2);
```

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

CODE 3 Example function call

```
@add(a)to(b){
    b = b + a;
}
add(1)to(x);
```

Increase x 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.

1.2 RULE FUNCTION and FACT FUNCTION

The attribute "expr" is added to the function declaration to indicate that the return of this function is an expression. Such functions serve as "rules" in logic programming and are called “rule functions”, while the functions in Section 1.1 returning the computed values play the role of "facts" and are called “fact functions”.

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

CODE 4 Example of function returning expression

```
expr: @{a * c + b * c}{
        return: (a + b) * c;
}
```

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, the function name is also in its function declaration scope, but the curly braces on the scope boundary are omitted for writing convenience.

1.3 FORWARD FUNCTION and INVERSE FUNCTION

Both forward and inverse functions are for fact functions; rule functions 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.

Inverse function refers to the function whose return value is determined and whose input parameters have undetermined parameters. The execution process of inverse function is to use the return value of the function and the determined input parameters to calculate the undetermined input parameters.

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

CODE 5 Example reverse function

```
@{a * $x^2 + b * x + c}{
        x = (-b + (b^2 - 4 * a * (c - ans))^0.5)/(2 * a);
}
@{$a == b}{
      a = b;
}
```

The "$" symbol acts on parameter x, indicating that parameter x is undetermined in the expression. For variables that appear multiple times in an expression, only need to use "$" 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 users can choose to use or not.

When invoking inverse functions, the user needs to modify the undetermined variable to be decorated with "$" (code 6):

CODE 6 Example for calling inverse function

```
1 * $a^2 + (-2) * a + 1 == 0;
```

a needs to be decorated with "$".

1.4 PROCESS CONTROL PROMPT

The process control prompt is used to control the reasoning process of the reasoning system. Users can control the reasoning process through a vector in the function. The dimensions of the vector represent the total number of stages that the user estimates will be required to handle the problem. Non-zero bits represent the stage at which this function can be called, along with the reward obtained by the compiler when calling this function. A positive and larger reward indicates that the user is more inclined to suggest that the compiler call this function multiple times at this stage. Conversely, a negative and smaller reward suggests that the user is less inclined to recommend the

compiler to call this function multiple times at this stage. For example, code 7 defines a function with a process control prompt of (10,0), indicating that the function can only be invoked at the first stage, and the user strongly recommend that the compiler call this function as often as possible:

CODE 7 Examples of function with the user prompt

```
@(10,0){$a == b}{
    a = b;
}
```

A function without a process control prompt can be invoked at any stage with a default reward.

1.5 LOGIC GUIDED FUNCTION DERIVATION

Function derivation is a special way to define inverse functions through forward functions. Example code 8:

CODE 8 Example of using the forward function to define the reverse function

```
@ price of buying (a) kg of apple unit price (b){
    return: a * b;
} => @apple unit price (b) can be bought ($a) kg;
```

Where "=>" indicates derivation. The necessary conditions for using this method to define the inverse function are: the names of all parameters in the function declaration of the inverse function must correspond to the names of parameters in the function declaration of the forward function one by one; the undetermined parameter in the inverse 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 of the 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 derivation may not be completed for other reasons.

2 VARIABLE

2.1 DECLARATION and TYPE of VARIABLE

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

CODE 9 Example of variable declaration

```
new: a = 1;
```

The type of a 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 a floating-point number.

2.2 VARIABLE ACCESS

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 the keyword 'out' to modify this behavior, allowing the variable to refer to the variable from the upper scope. Code 10:

CODE 10 "out" usage example

```
new: a = 1;
```

```
{
        new: a = 0;
      a = out: a + 1;
}
```

The result of calculation (the final value of a 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 function declaration scope. For example, pi in the function declaration of code 11:

CODE 11 Example use of "out" in a function declaration

```
new: pi = 3.14159;
   …… 
exp: @{sin(out: pi/2 − a)}{
        return: cos(a);
}
```

3 CONDITIONAL STATEMENT

3.1 LOOP

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

CODE 12 “while” loop example

```
while(a > 0){
        …… 
}
```

Use $continue$ to end the current loop and $break$ to end the entire loop.

3.2 BRANCH

Branch structure of COOL in code 13:

CODE 13 Branch structure example

```
if(a == 0){
        …… 
}elif(a > 0){
        …… 
}else{
        …… 
}
```

4 COMMENT STATEMENT

The comment style follows that of C, as shown in code 14:

CODE 14 Example of comment

```
//Single − line comments
/* One or more lines of comments */
```

COMPLETE CODE EXAMPLE 1

Solve the following mathematical problems:

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

1. Sum x and y and record the result as a;

2. Modify the value of x so that it satisfies the value of $x + 1$ equal to y;

3. Solve the variable z, where $z^2 + x * z + y$ equals 100;

4. Sum a, x, z to get the final result;

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

The complete COOL code for solving the problem, such as code 15, involves rule functions and fact functions, forward functions and reverse functions, process control prompts, and function derivation.

CODE 15 Complete code example 1

```
/* Function for solving quadratic equations */
@(100){a * $x^2 + b * x + c}{
      x = (-b + (b^2 - 4 * a * (c - ans))^0.5)/(2 * a);
}
/* Defines the law of additive exchange a + b→b + a. For the rule function, the
function parameters 'a' and 'b' are modified by" #", indicating that the law of
additive exchange can be applied regardless of whether a or b is determined or not.
*/
expr: @(-1){#a + #b}{
   return: b + a;
}
/* Define addition - subtraction conversion :
      a - b→a + (-b);*/
exp: @(-1){#a - #b}{
      return: a + (-b);
}
/* Define additive reverse function */
@(10){$a + b}{
       a = ans - b;
}
/* Define inverse function of equation */
@(10){$a == b;}{
       a = b;
}
/* Define a forward function for deriving the reverse
function based on the problem */
@get result from (x) and (y){
      new: a = x + y;
      $x + 1 == y;
      new: z = 0;
      1 * $z^2 + x * z + y == 100;
      return: a + x + z;
} ⇒ @get result from ($x) and (y);

new: x = 0;
new: y = 3;
get result from ($x) and (y) == 50;
x -> "#FILE(SCREEN)";/* "-- >" indicates output, which means that
the x value is output to the screen. The string in "#FILE(file name)"
will be considered as a file*/
```

5 CLASS

Rules for solving similar problems are considered to belong to one knowledge domain. They can be encapsulated into a file or a class. By creating classes, users can reuse, modify, and expand groups of rules more flexibly through inheritance. This enables the division and treatment of complex problems and facilitates the modular development of programs.

5.1 CLASS DEFINITION

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

```
class : OperationLaw{
        … …
}
```

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

5.2 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

```
class: MainProcess << OperationLaw,QuadraticEquation {
        … …
};
```

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.

5.3 CLASS INSTANCE INITIALIZATION

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

CODE 18 Class instance declaration example

```
MainProcess: m;
```

Its initialization is similar to a function invocation. After entering the scope of the class, it creates an active record and executes the code in the scope of the class. 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.

5.4 MEMBER ACCESS

Using "." operator to access member variables or member functions, such as code 19:

CODE 19 Example of access members

```
m.x = 1;
m.constructor( );
```

2.7 COMPLETE CODE EXAMPLE 2

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

First, two classes are defined. The class *Operationlaw* contains the transformation rules of some common operations. The class *Quadricequation* contains a formula for solving the univariate quadratic equation. Then, the main program class *Quadricequation* 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

```
//Define the class composed of operation laws
class:OperationLaw{
    //transposition of terms
    expr:@(-10){$a == b}{
        return:a - b == 0;
    }
    //Addition and subtraction conversion
    expr:@(-10){#a - b}{
        return:a + (-b);
    }
};
//Define the class for solving quadratic equations
system: QuadraticEquation{
        /* Functions for solving standard quadratic
    equations of one variable */
    @(-100){ a * $x^2 + b * x + c == 0;}{
        x = (-b + (b^2 - 4 * a * c)^0.5)/(2 * a);
    }
}
/* "<<" indicates the MainProcessclass
inherits the OperationLaw class and the
QuadraticEquation class. It can access the members of
 the two classes at the same time */
system: MainProcess <<
            OperationLaw, QuadraticEquation {
    new: x = 1;
    @constructor(){
        1 * $x^2 + 4 * x == 100;
        x -- -> 0;
    }
    @increase(n){
        x = x + n;
    }
};
MainProcess: m;//create class objects
m.constructor();//Call member function
m.increase(10);
m.x--→ "FILE(SCREEN)";//Output the value of the member variable
```

6 CONTAINER

The types of COOL's container include list, map, multimap, set, multiset and support pop, push, insert, erase, find, count, and clear operations.

Pop：

x.popFront(), x.popBack(): Pop the header or tail element of the container. x can be a list or a string

Push：

x.pushFront(arg), x.pushBack(arg): Pushes elements into the head or tail of the container. x can be a list or a string

Insert:

x.insert(element): Insert a specific element into the container. x can be a set, a multiset, or a string; insert a specific element to the end of the container. x can be a list or a string

x.insert({key,value}): Insert a specific key-value pair into the container. x can be a map or a multimap;

x.insert({position,element}): Inserts a specific element into the container at the specified position. x can be a list or a string.

Erase：

x.erase(position): Removes the element at the specified position. x can be a list or a string.

x.erase(key): Delete the key-value pair corresponding to the specified key. x can be a map or a multimap.

x.erase(element): Delete the specific element in the container. x can be a set or a multiset.

Find：

x.find(element): Return the position of the element in the container. x can be a list, a set, a multiset or a string.

x.find(key): Return the value corresponding to the specified key. x can be a map or a multimap.

Count：

x.count(element): Return the number of occurrences of an element in the container. x can be a list, a set, a multiset or a string.

x.count (key): Return the number of occurrences of the specified key in the query container. x can be a map or a multimap

[ ]：

x[position]: Return the reference or the duplication for the element at the specified position. x can be a list or a string. x[1,2,3] is equivalent to x[1][2][3]

x[key]: Return the reference to the value corresponding to the specified key. x can be a map.

Clear：

x.clear(): Clears all elements of a container or string.

Length：

x.length: Get the length of the container or string.

Typename：

x.typename: Get the type string of a variable.

Initialization:

Initialized by the `map()`, `multimap()`, `set()`, `multiset()` functions, and by the `{x,...}` syntax, initializes the array.

7 OPERATOR

Precedence	Operator	Description	Grouping	Attention
The highest	.	Member access	→	
	$	Identification parameters to be undetermined		
	#	Identification parameters are either to be determined or confirmed		
	:	Assign a value attribute	←	
	[]	The subscript access	→	
	!	Logical NOT	←	
	-- ++	Prefix increment / decrement		

	-- ++	Postfix increment / decrement		
	-	Minus sign	←	
	^	Exponentiation or intersection	→	
	* / %	Multiply/divide/mod	→	
	+ -	Add or take union/subtract or take difference	→	
	> < <= >=	Comparison operators	→	
	== !=	Equality/inequality	→	
	&& ||	Logical AND/logical OR		
	= += -= *= /= %=	Assignment/compound assignment		
	-->	Output	→	
	<<	Inheritance		
	=>	Derivation		
	,	Comma separator	→	
	;	Semicolon separator	→	
The lowest	@	Function declaration		
	@()	Function declarations with prompts		

* Why functional instead of declarative code style?

1. The most important advantage of using functions to represent rules or facts is that they are more concise and readable when dealing with complex logic in rules, although they are still declarative in nature.

   Here is an is a simple simplification rule:

   When ‘a’ is an unknown number and ‘b’ is an immediate number, if ‘b’ is zero, simplify ‘a-b’ to ‘a’, if ‘b’ is not zero, simplify to ‘a+(-b)’.

   Written in a purely declarative style, it is not easy to quickly understand the purpose of this set of rules:

   CODE Example of Prolog (Declarative)

```
simplify_condition(a, b, a) :- immediate(b), b =:= 0.
simplify_condition(a, b, result) :- -immediate(b), b < 0, result is a +
(-b).
simplify(a, b, result) :- simplify_condition(a, b, result).
```

   When written in a function-like style, the processing logic can be understood relatively easily:

   CODE Example of Prolog’s code (Function-like style)

```
simplify(a + b, result):-
    immediate(b),
    (b == 0
        -> result = a
    ; b < 0
        -> tmp is 0 - b,
        result = a + tmp
    ).
```

   In COOL, the style is further functionalized. In addition to explicitly adopting keywords common in GPLs (C++, Java, etc.), it is also allowed to present constraints on variables and rules/facts in a manner similar to variable attributes.

   Furthermore, expressions are allowed to be used directly as function names, which improves the expressivity of rules and saves part of the work of naming rules/facts.

   CODE Example of COOL’s code (Functional style)

```
expr: @(2,2,2,2,2){$a - immediate: b}{
    if(b == 0){
        return: a;
    }else{
        new: tmp = 0 - b;
        return: a + tmp;
    }
}
```