COOL Syntax Manual

For Domain-Specific Language Programmers

First Edition

CHAPTER 1

FUNCTION

COOL features a clean, parenthesis-free syntax. Rules are presented in the form of functions, which are central to COOL. Mastering writing COOL functions will unlock the powerful potential of Domain-Specific Language (DSL) programming.

1 Declaration and Invocation

In COOL, a function consists of a function declaration and a body. A function representing addition can be defined as Code 1.1:

Code 1.1: Function Declaration

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

where `@` modifies the subsequent `add(a,b)` as a function declaration rather than a function call. We need to remove `@` when calling a function, as shown in Code 1.2:

Code 1.2: Function Invocation

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

Functions prioritize passing references to actual parameters rather than making copies. This practice promotes efficiency and clarity in function calls:

Code 1.3: Reference Passing in Function Invocation

```
@add (a) to (b){
    b = b+a;
 }
 new:x = 0;
 add (1) to (x);
```

In Code 1.3, when `add (1) to (x)` is executed, the value of `x` is increased by 1 as a result of the operations defined in the `add (a) to (b)` function. Notably, function parameters can be embedded directly within the function name string, enhancing the expressiveness of the code.

2 Expression-Returning Function and Value-Returning Function

COOL functions can be divided into two categories according to whether they return an expression or a value. **Expression-returning functions** serve as tree operation rules in DSL reasoning and are called by the DSL solver automatically, while **value-returning functions** generate computed data, and programmers can invoke them directly.

Difference: Expression-returning functions are declared with the attribute `expr`, while value-returning functions are not.

2.1 Expression-Returning Function

Code 2.1 is an expression-returning function, describing the inverse operation of the distributive law of multiplication:

Code 2.1: Expression-Returning Function

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

Where the pair of curly braces immediately following `@` delimits a scope termed as **function declaration scope**. And the internal expression within it is termed as **function declaration expression**. In fact, in Code 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.

2.2 Forward Function and Inverse Function

Value-returning functions can be further categorized into forward functions and inverse functions, depending on whether the function has undetermined input or undetermined output:

- **Forward Function:** A forward function is characterized by having predetermined input parameters, while its return value remains undetermined. The invocation process of a forward function involves deducing the return value based on the provided input parameters. For instance, consider Code 1.1.
- **Inverse Function:** Conversely, an inverse function has a determined return value but contains input parameters with values that are yet to be defined (hole). The execution process of an inverse function involves using the known return value and fixed input parameters to calculate the undetermined input parameters. For example, Code 2.2 illustrates an inverse function required to compute the roots of a quadratic equation.

Code 2.2: Inverse Function

```
@{a*$x^2+b*x+c}{
    new:x1 = (-b + (b^2 - 4*a*(c - ans))^0.5)/(2*a);
    new:x2 = (-b - (b^2 - 4*a*(c - ans))^0.5)/(2*a);
    x = {x1, x2};
 }
```

In Code 2.2, the `$` symbol indicates that the parameter `x`'s value is yet to be determined in the expression. For a variable that appears multiple times in an expression, the `$` decoration is required only once. Additionally, the variable `ans` represents a known return value, which can be accessed within the function body.

When calling an inverse function, we need to specify the undetermined variable using the `$` decoration. For example, Code 2.3:

Code 2.3: Solving a Quadratic Equation where x is Unknown

```
2*$x^2 - 9*x + 7 == 0;
```

COOL supports automatically inferring the inverse function based on the corresponding forward function. This process is illustrated in Code 2.4:

Code 2.4: Automatic Function Inversion

```
@price of (a) kg apple unit price (b){
    return:a*b;
 =>@apple unit price (b) for ($a) kg;
```

Where `=>` indicates derivation, signifying we want the solver to generate the logic of function `@apple unit price (b) for ($a) kg` based on `@price of (a) kg apple unit price (b)`. It is important to ensure that the parameter identifiers for the derived function — `(a, b)` — remain consistent with those of the original function.

Furthermore, Code 2.5 demonstrates that COOL supports the inversion of multi-step forward functions that involve intermediate variables, provided that the functions necessary for each step's inversion have been defined:

Code 2.5: Multi-Step Function Inversion

```
@{a * $x^2 + b * x + c} {...}
@{$x + b} {...}
@{$x == b} {...}
...
@forward(x, y) {
   new:  a = x + y;
   $x + 1 == y;
   new:  z = 0;
   3 * $x^2 + x * z + y == 100;
   return:  a + x + z;
} => @inverse($x, y);
```

3 Chain-of-Logic

The Chain-of-Logic is a framework designed for managing function invocations, utilizing heuristic vectors and keywords.

- **Heuristic Vector:** Heuristic Vectors facilitate the segmentation of function calls within the same scope (such as a class or library) into multiple stages. Functions that are modified by a specific heuristic vector can only be invoked in the stage dictated by the non-zero bits of that vector. For example, in Code 3.1, the functions

modified by the heuristic vector `(9)` and `(0, 0, 5)` can only be called in `Stage 1` and `Stage 3` respectively. The non-zero values within heuristic vectors represent rewards associated with function invocations at different stages, influencing the priority and frequency with which the function is called during the reasoning process of the solver.

- **Keyword:** Chain-of-Logic keywords include `return`, `logicjump`, and `abort`, which control the flow of function calls by managing the stage advancement, conditional jumping, and termination, respectively.

Code 3.1: Family Relationship Reasoning with Chain-of-Logic

```
//Stage 1:  Separate Relations and Genders
 expr:@(9){(a) is (b)s brother}{
    return:(a) is male & (a) is (b)s sibling & (b) is (a)s sibling;
 }
 ...
 //Stage 3:  Reason Indirect Relations
 expr:@(0,0,5){(a) is (b)s sibling}{
    if(a == b){
       abort;
    }
    if(this expr.exist subexpr{(a) is male} == false && this expr
.exist subexpr{ (a) is female } == false ){
       logicjump(1);
    }
    placeholder:p1;
    while(this expr .find subexpr{ (p1) is (a)s sibling }){
       if(this expr.exist subexpr{ (p1) is (b)s sibling } == false
&& p1 != b){
          return:  (a) is (b)s sibling & (p1) is (b)s sibling;
       }
       p1.reset();
    }
    p1.reset();
    ...
 }
 ...
```

In Code 3.1, the logic of the second function asserts that if `a` and `b` are siblings, how the relationships of `b` are explored based on those of `a`:

1. First, the reasoning fails if `a` is found to equal `b`, prompting an exit with `abort`.
2. Second, the reasoning stage will jump to `Stage 1` through `logicjump(1)` when the gender of `a` is not directly known.

3. After that, if `p1` is identified as a sibling of `a`, and it differs from `b` with an undetermined relationship to `b`, then `p1` can be concluded to be a sibling of `b`.

In this context, `p1` serves as a placeholder for pattern matching within the expression, while `this expr` refers to the expression impacted by the expression-returning function. Both `find subexpr{expression}` and `exist subexpr{expression}` perform pattern matching but differ in their functionality: the former includes an internal iterator, enabling traversal through matches. The `reset` function is used to revert the placeholder to its initial state.

(For detailed usage, please refer to the paper COOL: Efficient and Reliable Chain-Oriented Objective Logic with Neural Networks Feedback Ccontrol for Program Synthesis.)

CHAPTER 2

VERIABLE

This chapter introduces the concept of variables in COOL, highlighting their declaration, types, and access. Understanding how variables operate is essential for effective programming in COOL, as it lays the foundation for data manipulation and function execution within the language.

1 Declaration and Type

Variables in COOL must be declared with `new` or a specified type before use. During the life cycle of a variable, COOL doesn't allow implicit type conversion, therefore the type of a variable is determined by the most recent value assigned to it. An example of variable declaration, type query, and assignment is shown in Code 1.1:

Code 1.1: Variable Declaration and Type

```
new:a;
a.typename->"#FILE(SCREEN)"; //"number"
a = "hello world";
a.typename->"#FILE(SCREEN)"; //"string"
string:b = "123";
a = b.toNum();
a.typename->"#FILE(SCREEN)"; //"number"
```

In Code 1.1, we can observe that variable declarations primarily establish the initial value of the variable. When a variable is declared using the `new`, it defaults to a floating-point value of zero. The `typename` is utilized to query the current type of the variable, while the `-->` operator is used for exporting data to the file system. COOL also offers basic type conversion functions, including `toInt`, `toNum`, and `toStrg`.

2 Access

Variables can be accessed from their declaration until the end of their defined scope. When an expression needs to refer to a variable, it prioritizes the variable in the current scope. To alter this behavior, we can utilize the `out` modifier, which allows expressions to reference variables from the upper scope. For example, see Code 2.1:

Code 2.1: Variable Access

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

In this example, the final value of `a` defined in the inner scope is 2.

When the `out` modifier is applied in a function declaration scope, the parameter no longer serves as a formal parameter but rather as an actual parameter external to the function, as illustrated in Code 2.2:

Code 2.2: “out” in Function Declaration

```
new:pi;
expr:@{sin(out:pi/2-a)}{
   return:cos(a);
}
```

Where the function represents the trigonometric transformation `sin(pi/2 - x) = cos(x)`, and `out:pi` in the function declaration refers to the global variable `pi`, not a formal parameter.

CHAPTER 3

CONDITIONAL STATEMENT

In this section, we will explore the conditional statements in COOL, which include branching and loop structures providing essential control flow capabilities.

1 Branching

The branching structure in COOL allows for conditional execution of code blocks, as shown in Code 1.1:

Code 1.1: "If-Else" Branching

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

2 Loop

COOL supports the "while" loop structure, as demonstrated in Code 2.1:

Code 2.1: "While" Loop

```
while(a > 0){
    ++a;
    ...
    if(a % 2 == 0){
       continue;
    }
    ...
    if(a > 100){
       break;
    }
    ...
 }
```

Within this loop, the `continue` statement is used to skip the current iteration, while the `break` statement exits the loop.

CHAPTER 4

ENCAPSULATION

Encapsulation is a key concept that allows programmers to group multiple DSLs into libraries and classes, which can be easily loaded and utilized through specific instructions. This promotes modularity and reusability in programming.

1 Library

To facilitate the management of DSLs, programmers can encapsulate multiple DSLs into libraries and call them using the `load` instruction, as demonstrated in Code 1.1:

Code 1.1: Library Loading

```
  #load(quadratic) // Load the DSL library for quadratic solving
  #load(family) // Load the DSL library for family relationship
reasoning
  new:x = 1;
  $x^2 - 4*x == 6;
  ...
  new:relation = "";
  new:Lynn = "Lynn";
  new:Joshua = "Joshua";
  new:Don = "Don";
  new:Dolores = "Dolores";
  (Joshua) is (Lynn)s husband & (Don) is (Joshua)s son & (Dolores) is
(Don)s wife & (Dolores) is (Lynn)s ($relation);
  ...
```

2 Class

When developing DSLs with COOL, we can encapsulate related functions (rules) for solving similar problems within classes. This structure allows us to reuse, modify, and expand their code more flexibly through inheritance, promoting better organization and modular development of complex programs. [1]

2.1 Definition

A class in COOL consists of a declaration with `class` followed by its body, which contains the scope of the class. This is illustrated in Code 2.1:

[1] Currently, Object-oriented programming in COOL is not fully compatible with Chain-of-Logic.

Code 2.1: Variable Access

```
class:OperationLaw {
   expr:@{$a == $b}{
      return:a - b == 0;
   }
   ...
}
class:QuadraticEquation {
   @{a * $x^2 + b * x + c}{...}
}
```

Where we define two classes named `OperationLaw` and `QuadraticEquation`.

2.2 Inheritance

Classes in COOL can inherit from other classes, allowing them to access member functions and variables from their parent classes. See Code 2.2:

Code 2.2: Inheritance

```
class:MainProcess << OperationLaw, QuadraticEquation {
   new:x = 1;
   @problem() {
      2 * $x^2 + 4 * x == 100;
      x -> "#FILE(SCREEN)";
   }
}
```

In this code, the class `MainProcess` inherits classes `OperationLaw` and `QuadraticEquation`, where the `<<` operator indicates inheritance. Besides, the members in `OperationLaw` are accessed first, which is consistent with the inheritance order (left to right).

2.3 Instantiation

Instances of a class are created as variable declarations with class name as the type name, as shown in Code 2.3:

Code 2.3: Instantiation

```
MainProcess:m;
```

The instantiation process behaves similarly to a function call. Upon entering the class's scope, an activation record is created, and the code within the scope executes sequentially. However, when exiting the class scope, this activation record is retained as the value of the corresponding instance rather than being destroyed.

2.4 Accessing Members

Members of an instance can be accessed using the `.` operator, enabling us to reach member variables or invoke member functions, as illustrated in Code 2.4:

Code 2.4: Accessing Members

```
m.x = 3;
m.solveProblem();
```

CHAPTER 5

CONTAINER

COOL enhances the management and organization of collections by packaging C++ containers, which include lists, maps, multimaps, sets, and multisets.

1 Initialization

Containers in COOL can be initialized using specific functions:

- `map()`: Creates and initializes an empty map.
- `multimap()`: Creates and initializes an empty multimap.
- `set()`: Creates and initializes an empty set.
- `multiset()`: Creates and initializes an empty multiset.
- `{x,...}`: Initializes a list with the specified values.

2 Operation

COOL supports a comprehensive set of operations to facilitate container manipulation:

2.1 Pop

- `x.popFront()`: Removes the first element (header) from the container. This operation can be applied to both lists and strings.
- `x.popBack()`: Removes the last element (tail) from the container, applicable to lists and strings.

2.2 Push

- `x.pushFront(element)`: Adds an element to the front (head) of the container. This operation is applicable to both lists and strings.
- `x.pushBack(element)`: Adds an element to the back (tail) of the container, usable with lists and strings.

2.3 Insert

- `x.insert(element)`: Inserts a specific element at the end of the container; applies to sets, multisets, or strings.
- `x.insert({key, value})`: Adds a key-value pair to the container, applicable for maps and multimaps.
- `x.insert({position, element})`: Inserts an element at a specified position within the container; suitable for lists and strings.

2.4 Erase

- `x.erase(position)`: Removes the element at the specified position in the container, applicable to lists and strings.
- `x.erase(key)`: Deletes the key-value pair associated with a specified key, applicable for maps and multimaps.
- `x.erase(element)`: Removes a specific element from the container, applicable for sets or multisets.

2.5 Find

- `x.find(element)`: Searches for the position of a specified element in the container, applicable to lists, sets, multisets, and strings.
- `x.find(key)`: Retrieves the value corresponding to a specified key, applicable to maps and multimaps.

2.6 Count

- `x.count(element)`: Determines the number of occurrences of an element in the container, applicable to lists, sets, multisets, and strings.
- `x.count(key)`: Finds the number of occurrences of a specified key in the container, applicable to maps and multimaps.

2.7 Indexing

- `x[position]`: Returns a reference or a copy of the element located at the specified position. This operation can be performed on lists or strings. For nested indexing, `x[1, 2, 3]` is equivalent to `x[1][2][3]`.
- `x[key]`: Provides a reference to the value associated with the specified key, applicable for maps.

2.8 Clear

- `x.clear()`: Empties all elements from a container or string, effectively resetting it to an empty state.

2.9 Length

- `x.length`: Retrieves the total number of elements in the container or the length of a string.

2.10 Container Type Query

- `x.typename`: Returns the type name of the container.

CHAPTER 6

OPERATOR

This chapter provides an overview of the built-in and custom operators, which provide essential functionality for coding and program execution in COOL.

1 Built-in Operator

COOL features a variety of operators with different precedences and uses. Below is a summary of the built-in operators, their precedence, and descriptions:

Precedence	Operator	Description	Grouping
Highest	.	Member access	→
	User-defined operator		
	&, |		
	$	Parameters are undetermined	No continuous use
	#	Parameters are either to be undetermined or confirmed	No continuous use
	:	Declaration	←
	[ ]	Subscript access	→
	!	Logical NOT	←
	- -, ++	Prefix increment/decrement	No continuous use
	- -, ++	Postfix increment/decrement	No continuous use
	-	Minus sign	←
	^	Exponentiation or intersection	→
	*, /, %	Multiply/divide/mod	→
	+, -	Add or take union/subtract or take difference	→
	>, <, <=, >=	Comparison operators	→
	==, !=	Equality/inequality	→
	&&, ||	Logical AND/logical OR	
	=, +=, -=, *=, /=, %=	(Compound) assignment	
	- ->	Output	→
	< <	Inheritance	No continuous use
	= >	Derivation	No continuous use
	,	Comma separator	→
	;	Semicolon separator	→
	@	Function declaration	No continuous use
Lowest	@( )	Function declaration with a heuristic vector	No continuous use

2 Custom Operator

In addition to built-in operators, COOL allows us to define custom operators with different associativity to suit specific programming needs:

- `~*`: This operator is left-associative and can be either unary or binary; it must have an operand on the right.
- `~~*`: This operator is right-associative and can be either unary or binary; it must also have an operand on the right.
- `~~~*`: This operator is left-associative and is strictly unary; it must have an operand on the left.

Note that for defining custom operators, `*` can include any ASCII symbols except for double quotes, commas, various types of brackets, `@`, `#`, and `~`.

CHAPTER 7

MISCELLANEOUS

This section introduces syntax that assists programming with COOL.

1 Comment Statement

In COOL, the way to add comments is consistent with that of the C programming language (Code 1.1). The two styles of comments provided are as follows:

- **Single-line Comment:** A single-line comment begins with `//` and extends to the end of the line.
- **Multi-line Comment:** A multi-line comment begins with `/*` and ends with `*/`. This style can encompass one or more lines.

Code 1.1: Comment

```
// This is a single-line comment
...
/* This is a multi-line comment.
It can span multiple lines.  */
...
```