

Dao built-in methods for functional-style programming

September 19, 2009

(For version 1.1)

1 General Syntax

The general syntax for such built-in methods is the following,

```
RESULT = METHOD( PARAMETER ) -> |VARIABLE| { EXPRESSION }
```

where METHOD can be any of,

1. map
2. reduce/fold
3. unfold
4. select
5. index
6. count
7. each
8. repeat

It can also be **string** , **array** or **list** for data construction with such syntax. Another two methods adopted such syntax are **sort** and **apply** .

Depends on the METHOD, the PARAMETER can be one or more list, array, or any other types. Optional VARIABLE can be declared according to the METHOD and PARAMETER, if |VARIABLE| is not present, **x,y,z** and/or **i,j** will be declared automatically when necessary.

The EXPRESSION placed inside the braces is the (inlined) function to be applied to the PARAMETER or the items of PARAMETER if it is a list or array, which is passed to the EXPRESSION through parameter(s) declared in VARIABLE or declared automatically.

A simple example,

```
a = { 1, 2, 3 }  
b = map( a ) -> { 10*x } # produce {10, 20, 30 }
```

which is equivalent to,

```
a = { 1, 2, 3 }  
b = map( a ) -> |x| { 10*x }
```

When multiple lists/arrays are used as parameters, the variables can be declared accordingly as the items of the lists/arrays,

```
a = { 1, 2, 3 }  
b = { 11, 22, 33 }  
c = map( a, b ) -> |x,y| { x + y }
```

in this example, *x* is declared as the item of *a* and *y* as the item of *b* . Here $|x,y|$ can be omitted.

Multiple expressions can also be used and must be separated by commas, and the result of the expressions will be a tuple,

```
a = { 1, 2, 3 }  
b = map( a ) -> { 10*x, x+10 } # produce {(10, 11), (20, 12), (30, 13) }
```

To use statements inside the braces, it is necessary to place a **return** keyword before EXPRESSION as the following,

```
... -> |VARIABLE| { BLOCK return EXPRESSION }
```

for example,

```
a = { 1, 2, 3 }
```

```

b = map( a ) -> |x| {
  u = 10*x;
  v = x+10;
  if( u < v ) u += v;
  return u*v
}

```

No other **return** keyword can be used.

Function composition can be done by appending more `->|...|{...}` structures one after another,

```

RESULT = METHOD( PARAMETER )
-> |VARIABLE| { EXPRESSION }
-> |VARIABLE| { EXPRESSION }
...

```

For the VARIABLE in the `->|...|{...}` structures after the first, variables can be declared according to the EXPRESSION in the previous structure. For example,

```

a = { 1, 2, 3 }
b = { 11, 22, 33 }
c = map( a, b ) -> |x,y| { x + y, x - y } -> |u,v| { u * v }

```

Here, u is declared as the result of $x+y$ and v as the result of $x-y$.

2 map, reduce/fold, unfold

2.1 map()

`map()` take one or more lists/arrays as parameter(s), and apply a (inlined) function to each of the items of the lists/arrays, then return the results of the function applications as a list/array. When more than one list/array are used as parameters, they should have the same size.

As mentioned, `map()` can also take array(s) as parameters, and return an array as result. If a simple scalar type is returned as result in the (last) braces, an array of the same shape as the (first) parameter array. If a tuple, list, or array etc. is returned as result, the final resulting array has additional dimensions over the (first) parameter array, such that, the result of each function application is used to set the value of the corresponding slice of the final array.

Some examples on list have been given in the previous section, here some examples on array will be shown,

```

a = [ 1, 2; 3, 4 ]

```

```
b = map( a ) -> { 10*x } # produce [ 10, 20; 30, 40 ]
```

```
a = [ 1, 2; 3, 4 ]  
b = [ 5, 6; 7, 8 ]  
c = map( a, b ) -> { x, y }  
# or  
c = map( a, b ) -> { [x, y] }  
# or  
c = map( a, b ) -> { {x, y} }  
# produce 3d array: [ [ [1, 5], [2, 6] ], [ [3, 7], [4, 8] ] ]
```

The zip() function which is often seen in some other languages is not supported in Dao, because it can easily achieved by map(),

```
a = { 1, 2, 3 }  
b = { 11, 22, 33 }  
c = map( a, b ) -> { x, y }
```

which will return a list of tuples "zipping" the two list. Clearly it can also be used to zip multiple lists.

2.2 reduce/fold()

fold() with its alias reduce(), takes a list/array and an initial value (the accumulator) as parameter, and apply the (inlined) function to each item of the list/array and the accumulator, and then the value of the accumulator after the last function application is returned. If the second parameter is omitted, the first item of the list is used as the initial value, and the function is applied to the rest of items.

By default, the item of the list/array is passed as parameter/variable **x** and the initial value (accumulator) as **y**, to the function application. Other parameter/variable names can also be used explicitly as described before.

```
a = { 1, 2, 3 }  
b = fold( a, 10 ) -> { x + y } # produce 16  
b = fold( a ) -> { x + y } # produce 6
```

2.3 unfold()

unfold() takes an initial value as parameter, and generates a list from this initial value, by applying the function to the initial value and then to the result of the previous function

application until a condition become false,

```
a = unfold( 5 ) -> |x| { while x > 0 return x - 1 }  
# produce: {4, 3, 2, 1, 0 }
```

where the condition expression must be placed after the **while** keyword, and the **return** keyword must also be used after the condition expression and before the expression(s) to generate results.

Statements can also be used before the **while** keyword.

```
a = unfold( 5 ) -> |x| {  
  stdout.println(x)  
  while x > 0 return x - 1  
}
```

3 select, index, count

select, **index**, **count** are the methods test a condition as function application over the items of the parameter lists/arrays, and select, or index, or count the items that makes the condition true.

```
a = { 10, 20, 30 }  
b = select( a ) -> { x >= 20 } # produce: {20, 30 }  
c = index( a ) -> { x >= 20 } # produce: {1, 2 }  
d = count( a ) -> { x >= 20 } # produce: 2
```

These methods can also be applied to multiple lists, or to array(s),

```
a = { 10, 20, 30 }  
b = { 'a', 'b', 'c' }  
c = select( a, b ) -> { 1 }  
# effectively as zip, produce: {( 10, a ), ( 20, b ), ( 30, c ) }
```

4 each, repeat

each() is the same as **map()** except that no result is returned.

repeat takes an integer number as parameter, and evaluate the BLOCK/EXPRESSION that number of times.

5 string, array, list

Based on the same idea as the above functional methods, **string**, **array**, **list** are provided as convenient way to construct complex string, array, list with simple and flexible codes.

These methods take a number as parameter, and evaluate the (inlined) function this number of times, and return the results as concatenated string, list or array. By default, the index of the function evaluation is passed as **i** to the function. Of course the variable for index can also be declared explicitly.

```
a = string( 5 ) -> { (string)i + 'a' }  
# produce: 0a1a2a3a4a  
  
b = list( 5 ) -> { (string)i + 'a' }  
# produce: {0a, 1a, 2a, 3a, 4a }  
  
c = array( 5 ) -> |x| { 10 * x }  
# produce: [ 0, 10, 20, 30, 40 ]
```

Slightly more complex examples:

```
d = list( 3 ) -> { { 1 : (i+2) } }  
# produce: {{1, 2 }, {1, 2, 3 }, {1, 2, 3, 4 }}  
  
e = array( 3 ) -> { [ 10*i+1, i+2 ] }  
# produce matrix: [ [1, 2], [11, 2], [21, 3] ]
```

6 sort, apply

6.1 sort()

sort() sorts a list according to a comparison function, it takes a list as the first parameter and optionally a number as the second parameter. If the second parameter is presented, the list will be partially sorted such that only the first some number of items will be properly sorted, which can be used to search for the largest/smallest *K* items among a list.

```
a = { 3, 2, 15, 8, 25, 16, 3, 7, 1, 12, 17, 11, 0 };  
sort( a, 6 ) -> { x < y }
```

where *x,y* are declared as the variables representing the two items to be compared during sorting. These variables can be declared explicitly as well.

6.2 apply()

apply() evaluates a function for each element of an array, and replace the value of the element by the result of the evaluation. By default the variable representing the element value is declared automatically as x , and the indices as variables i,j . The can also be declared explicitly as well. For arrays with dimension higher than 2, one must declare variables explicitly to get all components of the multi-dimensional index passed in.

```
a = [ 10, 20, 30 ]
apply( a ) -> { x + i }
# a becomes: [ 10, 21, 32 ]

b = [ [ 1, 2; 3, 4 ] : 3 ];
apply( b ) -> |x,i,j,k| {
  stdout.println( x, i, j, k );
  return x + i*j*k;
}
```