SCIC - Scientific Calculation Language

Yucen Sun (ys3393), Zhengyi Li (zl3029), Zhengyuan Dong (zd2216)
February 3, 2021

1 Introduction

The SCIC language is a statically scoped, statically typed functional language for science. Types for arguments and return data are explicitly specified. It mainly focuses on arithmetic operations and formula calculations with units. We allow users to define units and formula based on our basic units and operators. We support automatic unit conversion. The string support is limited.

2 Language Details

2.1 Data Types

(1)

Data Type	Description
int	32-bit signed integer
long	64-bit unsigned integer
float	single-precision 32-bit floating point
boolean	true or false
char	16-bit Unicode character
String	immutable array of characters

2.2 Data Structure

(2)

Data Structure	Description
int[] / float[]	list of integer / float

2.3 Operations

(3)

Arithmetic Operators	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder
^	Power

(4)

Unary Operators	Description	Usage
++	Increment	i++
−−	decrement	i−−

(5)

Relational Operators	Description
==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to
<	Less than
<=	Less than or equal to

2.4 Units

We provide most commonly used units in physics equations. Users are allowed to customize units. But users need to define them based on the primitive types, and provide conversion formula to primitive ones.

(6)

Primitive Unit	Description	Quantity
'm	Meter	Length
'N	Newton	Force
'kg	Kilogram	Mass
's	Second	Time
'{m}/{s}	Meter per second	Velocity

m

2.5 Control Flow

(7)

Control Flow	Description
if / else if / else	Conditional expression
for loop	Loop expression

```
if (x > 0) {
    print_type(x);
} else if (x < 0){
    print("x is negative!");
} else {
    print("0");
}
```

```
for (int i = 0; i < 10; i++) {
    print(i);
}
```

2.6 Functions

Function declaration is a key part of the SCIC language. There are two style of function declarations. The first one is similar to the C++ function declaration, and the second one is ′equation-like function′.

Type and Unit should be defined in the arguments of the function.

```
float 'm func dist_speed(float'm X1 , float '{m}/{s} Va , float 's T ) {
    return X1 + Va * T;
}
```

One can also define function of non-unit arguments (go by default unit constant 1). However, it's not allowed that some arguments have units and others not. For example, the following is allowed.

```
int func add2 (int a) {
    return a + 2;
}
```

Due to the equation nature of many scientific calculation, we suggest a new equation-like functions. Take the distance and speed equation for example, it's by nature a equation of the four arguments and we should be able to derive any one argument given the other three. There is no 'return' keyword for this type of function. Therefore, the equation-like function is declared as the following:

```
func dist_speed(float'm X1 , float'm X2  float'{m}/{s} Va , float's T) {
    X2 - X1 = Va * T;
}
```

When calling the equation-like functions, we need to specify each arguments so that the only one missing argument will be returned.

```
//one way to call equation-like function
float 'm x1 = 11.2;
float '{m}/{s} va = 0.5;
float 's t = 7;
float 'm res = dist_speed(X1=x1, Va=va, T=t);
```

```

// another way without pre-defined values
float res 'm = dist_speed(X1=11.2'm, Va=0.5 '{m}/{s}, t=7.0's);

/* if unit conversion is needed, simply specify the desired units
and arguments units: (units should be built-in or user pre-defined)
*/
float 'cm res = dist_speed(X1=3205'mm, Va=0.5 '{m}/{s}, t=7000'ms);
```

A function can call other functions within it.

```
func sub_dist (float X 'cm, float X0 'cm) {
    X = X0 - dist_speed(X1=1.0'm, Va=0.3'{m}/{s}, T=6 's);
}
```

2.7 Comments

SCIC language supports comments.

```
// in-line comments

/*
  multi-line comments
  multi-line comments
*/
```

3 Features

1. Users can define formula with units as a function. The function will calculate and return an unprovided parameter value given other parameters.

```
equa acceleration (float 'm/s v0 ,float 'm/s v1, float 's dt, float 'm/{s^2} a) {
    a = (v1 - v0)/dt;
}
float 'cm/s v1 = ceil(acceleration(v0 = 5, dt = 5, a = 3));
/* v1 = 1500 */
```

---

2. Pass a function as a parameter. Users can call formulation functions with parameters as another formulation function.

```
equa v_sum (float 'm/s v0 , float 'm/s v1 , float 'm/s delta_v) {
    delta_v = v0 + v1;
}
equa acceleration(float 'm/s delta_v, float 's t, float 'm/{s^2} a) {
    a = delta_v/t;
}
int 's new_t = acceleration(delta_v=delta_v(v0=5,v1=10), a=3);
// new_t = 5
```

---

3. A list of values with one unit.

```
int[] v0 = [1,2,3,4,5] '{m}/{s} ;
```

4. Pass a list as a parameter. The formulation functions will return a list of calculated unprovided parameter values.

```
int[] v0_data = [1, 2, 3, 4] '{m}/{s};
int[] v1_data = [5, 4, 3, 2] '{m}/{s};

equa v_sum (int[] 'm/s v0 ,int[] 'm/s v1 ,int[] 'm/s delta_v ) {
    delta_v = v0 + v1;
}

int[] '{m}/{s} delta_v_list = v_sum(v0=v0_data, v1=v1_data)
// delta_v_list will be a list [6,6,6,6] '{m}/{s}
```

5. Pipeline like linux. Allow data flows from left to right through the pipeline. In this example, we first calculate Δ speed, pass it to acc(x) function to get current speed, and pass it to displacement(t) function to get displacement.

```
int 'm distance  = speed(a=4, t=3) | acc(x=1, v=&) | displacement(t=2, a=&)
// variable v in function acc is pipelined from returned value of function speed
// variable a in function displacement is
// pipelined from returned value of function acc
```

4 Examples

```
// observation of kinetic energy transform
// Rotational energy transform from kinetic energy

// define formula function of kinetic energy from movement
equa kinetic_energy_t (float[] 'g m, float[] 'm/s v, float[] 'j K) {
  K = (1/2) * m * (v^2);
}

// define rotational energy
equa kinetic_energy_r (float[] 'kg*m^2 I ,float[] 'r/s w□float[] 'j K) {
    K = (1/2) * I * (w^2);
}

// unit transform define
|'kg = 1000 * 'g|

// experiment data
float[] m_data = [20.2, 30.2, 40.1, 50.4] 'g;
float[] v_data = [5.1, 6.3, 7.3, 8.2] 'm/s;
float[] I_data = [10.6, 12.7, 14.8, 16.1] 'kg*m^2;
float[] w_data = [3.1, 4.5, 6.7, 10.9] 'r/s;

// there multiple way to calculate list of angular speed w
// 1. get kinetic energy first, then pass by list
float[] 'j k_list = kinetic_energy_t(m = m_data, v = v_data);
float[] 'r/s w_list1 = kinetic_energy_r(I = I_data, K = k_list);
```

```
27
28  // 2. passing equa as parameter
29  float[] 'r/s w_list2 = kinetic_energy_r(I = I_data, k = kinetic_energy_t(m = m_data, v = v_dat
30  // note, the unit convert from g to kg achieved during call of functions
31
32  // 3. pipeline
33  float[] 'r/s w_list3 = kinetic_energy_t(m = m_data, v = v_data) |  kinetic_energy_r(I = I_data
34
35
36  // compare the calculate result of angular speed with experiment results
37  float[] 'r/s func compare(float[] 'r/s expr, float[] 'r/s calc) {
38      float[] 'r/s compare_list;
39      for (int i = 0; i < w_data.length; i++) {
40          compare_list.append(expr[i] - calc[i]);
41      }
42      return compare_list;
43  }
44
45  float[] 'r/s compare_data = compare(w_data, w_list1);
```
