SCIC Reference Manual

Yucen Sun, Zhengyi Li, and Zhengyuan Dong

{ys3393, zl3029, zd2216} @columbia.edu

1 Introduction

The SCIC language is a statically scoped, statically typed, C-like language for science. Data types and units are explicitly specified for each variable. SCIC supports arithmetic operations and equations solving with units. In addition, it allows users to define units based on our basic units and operators. It also supports automatic unit conversion.

2 Lexical conventions

2.1 Comments

SCIC does multi-line comments starting with /* and ending with */.

```
/* my first program in SCIC */
```

2.2 Identifiers

An identifier is composed of ascii letters, digits, and underscore ' _'; the first character must be letters.

2.3 Keywords

The following identifiers are reserved as keywords and cannot be used to define units or name variables:
int float long bool char string if else for return print true false equa function ceil floor typeof float2int int2float sizeof

2.4 Constants

Integer constants The integer constant is a sequence of digits in decimal; the first digit cannot be 0.

```
int x = 2;   /* allowed */
int x = 02;  /* not allowed to start with 0  */
```

Character constants The character constant is an ASCII characters enclosed in " ' ".

```
char c = 'a';
```

Float constants The float constant consists of 1 bit for sign, 1 bit for sign of exponent, 7 bits of expoenent and 23 bits of mantissa.

```
float t = 2.078;
```

Boolean constants The boolean constant is true or false.

```
bool x = false;
```

String constants The string constant is a sequence of characters enclosed in " " ".

```
string s = "hello world";
```

3 Data Types

Data Type	Description
int	32-bit signed integer
float	single-precision 32-bit floating point
bool	true or false
char	16-bit ASCII character
string	immutable array of characters

Data Structure	Description
int[ ]/float[ ]	integer/float array

4 Unit System

SCIC features arithmetic operations and formula calculations with units. The prototype supports a part of SI units as base units and their common conversions as built-in units.

SCIC also allows users to define units. User can define own base units and and conversion of base units as well. User-defined unit names must only contain English letters. Derived units can be defined as multiplication, division, and power of defined units. Unit declaration is global.

SCIC supports user-defined functions and equations with units. It also supports automatic unit conversion.

In SCIC, all units begin with identifier " ′ " and wrapped by curly braces {} such as '{m} and '{s}. Only defined unit identifiers and their power can go into {}, and constants shall not be placed inside. For example, derived unit $'\{m/s\}$ can be used because m and s are both defined, and we can also define cm per second using $|'\{cmps\} := 100 * '\{m/s\}|$; (more details follow). '{100 * m} or '{undefined-unit / defined-unit} is not allowed.

Base Units	Description	Quantity
'{s}	second	time
'{m}	meter	length
'{kg}	kilogram	mass
'{A}	ampere	electric current
Units	**Description**	**Quantity**
'{cm}	centimeter	length
'{g}	gram	mass
'{Hz} (=1/'{s})	frequency	times per second
'{N} (= '{kg * m * s^ (-2)})	newton	force

Variable declaration with unit:

```
float '{m} dist = 5.5;
int '{s} t1 = 1;
float[] '{m/s} speed_list = [1.09, 0.92, 0.883];
```

Define a unit based on base units. On the right of assignment symbol, constant calculation expression should be separated from the expression in the curly bracket only involving units .

```
|'{I} = '{A / s}|
/* Define current unit I as Ampere per second */

|'{uA} = 10^(-6) * '{A} |
/* Define microampere as uA */
```

Define a new base unit and units based on it:

```
|'{bowl} := 1|
/* bowl is a user-defined base unit */

|'{bucket} := 20 * '{bowl}|
/* bucket is a conversion unit of base unit bowl */

|'{bowlps} := '{bowl / s}|
/* bowlps is a derived unit of base unit bowl and second */
```

5 Conversions

5.1 Data conversion

Integer to float Integers between -16777216 and 16777216 (2^{24}) can be converted to float number without loss of precision.

Float to integer Float numbers between -32767 and 32767 can be converted to integer with truncation to 0.

5.2 Unit conversion

SCIC supports automatic unit conversion if the pre-convertion unit and post-convertion units are both defined and correctly related.

```
|'{cm/s} := 100 * '{m/s}|
float '{m/s} dist2 = 0.123;
float '{cm/s} dist1 = dist2;

/* value of dist1: 12.3, unit of dist1: cm/s */
```

6 Expressions

The precedence of expressions is descending in the order of the following subsections.

6.1 Primary expressions

identifier Identifier is a primary expression specified in its declaration.

constants Constant is a primary expression which can be an integer, a float number, a character, a string, and a boolean.

(expression) An expression with parentheses is a primary expression. The type and value of the expression remains unchanged.

primary expression [expression] The expression enclosed by square bracket after a primary expression is a primary expression. It is used to access an element in the array.

```
int[] x = [2,3,6,4];    /* initialize list */
int num = x[2];    /* the index of array starts from 0, num is at index 2, which is 6*/
x[1] = 0;  /* x becomes [0,3,6,4] */
```

primary expression (expression list) A primary expression followed by empty or a list of comma-separated expressions enclosed by parentheses is a function call. The expression lists are the parameters passed into the function.

6.2 Unary Operators

- expression The result is negative of the expression.

```
int x = -5;    /* x is negative 5 */
```

! expression The result is logically negative of the expression.

```
bool x = true;
bool y = !x;    /* y is false */
```

6.3 Multiplicative operators

The multiplicative operators are * for multiplication, / for division, evaluated from left to right. For division, if two literals are integers, the result is integer truncated to 0; if one or both literals are float numbers, the result is float.

```
int x = 2;
int y = 3;
float z = 4.0;
int mul = x * y;          /* mul = 6 */
int div = y / x;          /* div = 1*/
float res = z / x;        /* res = 2.0 */
```

6.4 Additive operators

The additive operators are + for addition, - for subtraction, evaluated from left to right. If one of operands is float, the result is float. If both operands are integer, the result is integer.

```
int x = 2;
float y = 3.0;
x + y;          /* 5.0 */
```

6.5 Relational operators

The relational operators are < (less than), > (greater than), <= (no greater than), >= (no less than). They yield true if the relation between two operands is true, otherwise, false.

6.6 Equality operators

The equality operators are == (equal to) and ! = (not equal to). They compare values of two operands and yield true if two values are the same.

6.7 Logical operators

The logical operators are && (logical and) and || (logical or). && yields true only if operands are true. || yields false only if both sides are false.

6.8 Assignment operators

Assignment is identifier = expression. Two operands should be the same type.

```
int x = 2;           /* correct */
int x = 2.0;         /* not allowed */
int x = (int) 2.0;   /* allowed, because the right operand becomes integer after conversion */
```

7 Declarations

7.1 Variable Declaration

Variable declaration follows the form
type unit identifier where type is limited to float and int.
or
type identifier without unit.

```
float 'm y;         /* declare variable y of float type and unit meter*/
y = 1.0;            /* y is 1.0 meter*/
int x = -5;         /* x is negative 5 */
```

7.2 Function Declaration

A function takes a list of arguments and returns a value. The body of a function is wrapped by curly braces. All arguments and returned value in a function declaration should have legal unit; or none of them should have units.

SCIC enforces function declaration and definition at the same time, and it follows the form:
type (unit) func identifier(arguments) {statements}
Examples of the two types of function declarations are as follows:

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

/* Function declaration with no units */
int func add2 (int a) {
    return a + 2;
}
```

7.3 Equation Declaration

Equation is a type of statement similar to function, but equation has no return value defined. Users can define relationship of the arguments in a equation statement. Either all arguments have units, or none should have units.

SCIC enforces equation declaration and definition at the same time, and it follows the form:

equa identifier(arguments) {statements}

An example of the equation declaration is as follows:

```
/* Equation declaration with units */
equa acc_equa (float '{m/s} v0 ,float '{m/s} v1, float 's dt, float '{m/s^2} a) {
    a = (v1 - v0)/dt;
}

/* Equation declaration with no units */
equa abc (int a, int b, int c) {
    b = a + 2c;
}
```

*More details of the expression to call equations in SCIC in Statements section.

8 Statements

statement are executed in sequence

8.1 Expression statement

expression

most of statement is an expression, and expression can be unit type define, variable assignment, function calls, or equation calls

8.2 Conditional Statement

if (expression) statement
if (expression) statement else statement

```
int x = 0;
if (true) x = 5;

if (true) x = 10 else x = 20;
```

the "else" is connected to the last "if" if there is ambiguity

8.3 For Statement

for (expression 1; expression 2; expression 3) statement

the first expression set up the initial state, the second expression check if loop can continue, and the third expression specifies the incrementation after each iteration

```
int x = 0;

for (int i = 0; i <= 10; i++){
    x = i+1;
}
```

int i = 0 is expression for setting up initial state. $i <= 10$ is the expression for checking loop stop. **i++** is what variable will be increment after each loop.

8.4 Return Statement

return expression;

the first statement will have no value return. the second statement will return value when function calls it.

```
/* return statement */
int func add(int x){
    return x+2;
}
```

9 Scope rules

Scope means when creating a function or equation, scope was created with hierarchy.
When **Scope** was form, then lower level of **Scope** cannot access higher level of **Scope**
The lower level scope can access variables and functions in higher level of scope
Default **Scope** is global scope, which is lowest level of scope

```
/* create a variable in global scope */
int y_2 = 0;

/* creating a function or equation will form a new scope */
int func add(int x){
    /* here is a higher level of scope */
    /* create a new variable in function scope */
    int y = 3;
    return y;
}

/* y_2 is in global scope, and y is in a function scope.*/
/* global scope is lowest scope, and cannot access variables in higher scope(function) */
/* y cannot be assigned to y_2* /
/* y_2 can be assigned to y*/
```

10 Library Functions

10.1 print

print would print any variable into the console

```
int x =10;
print(x);
/* console log: 10 */

string = "hello world";
print(string);
```

```
/* console log: "hello world"*/

int[] x = [1,2,3];
print(x);
/* console log: [1,2,3]*/
```

10.2 typeof

typeof function returns the type of identifier

```
int x =10;
typeof(x);            /* int  */

string s = "hello world";
typeof(s);           /* string */

int[] arr = [1,2,3];
typeof(arr);         /* int list*/

int '{m/s} x_w_unit = 10;
typeof(x_w_unit);    /* int '{m/s}*/
```

10.3 float2int

float2int will convert float type to int. Since the compiler cannot convert float to int automatically, it requires user to convert manually.

```
float '{m} dis = 10.2;
int '{m} new_dis = float2int(dis);   /* new_dis = 10 */
```

10.4 int2float

int2float will convert int type to float.

```
int '{m} dis = 10;
int '{s} sec = 3;
int '{m/s} speed = dis/sec;                    /* speed = 3 */
float '{m/s} new_speed = int2float(speed);     /* new_speed = 3.0 */
```

10.5 ceil

ceil(x) function returns the smallest integer greater than or equal to x.

```
float speed = 9.6;
int new_speed = ceil(speed);      /* new_speed = 10 */
```

10.6 floor

floor(x) returns the largest integer less than or equal to x.

```
float '{m/s} speed = 5.3;
int new_speed floor(speed);     /* new_speed = 5 */
```

10.7 Array Operation

find Array operation supports using array[indx] to find the value at index position

```
int[] newList = [1,2,3,4,5];
int x = newList[0];           /* x = 1 */
```

sizeof Array sizeof function returns length of the array

```
int[] nums = [1,2,3,4,5];
sizeof(nums);       /* 5 */
```

11 Examples

```
/*  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);
}

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


/* compare the calculate result of angular speed with experiment results */
int func compare(float '{r/s} expr, float '{r/s} calc) {
    int res;
    if (expr > calc) {
       res = 1;
    } else if (expr < calc) {
        res = -1;
    } else {
        res = 0;
    }
    return res;
}

int main() {
    /* experiment data */
    float[] '{g} m_data = [20.2, 30.2, 40.1, 50.4];
    float[] '{m/s} v_data = [5.1, 6.3, 7.3, 8.2];
    float[] '{m/s} k_data = [2.1, 3.3, 0.5, 8.1];

    int experiment_k;
```

```
    int res;
    for (int i = 0; i < sizeof(m_data); i++){
        experiment_k = kinetic_energy_t(m=m_data[i], v=v_data[i];
        res = compare(experiment_k, k_data[i]);
        print(res);
    }

    return 0;
}
```

References

1. C Reference Manual, https://www.bell-labs.com/usr/dmr/www/cman.pdf.
2. Single-precision floating-point format Wikiepdia, https://en.wikipedia.org/wiki/Single-precision_floating-point_format.