EasKey Language: Language Reference Manual

Keqiu Hu, Jinqi Huang, Xiaoyu Huang,

Zongheng Wang, Lizhong Zhang

1. Introduction

EasKey aims to facilitate programming of keyboard and mouse actions. It consists of most fundamental operations and functions of regular programming languages, such as branches and loops. In addition, all the potential keyboard and mouse actions are realized by EasKey built-in functions, which make the process of keyboard and mouse programming super easy for users. EasKey mainly solve three kinds of problems, repetitive operations, fast operations and customizable shortcut keys. Overall, the purpose of EasKey is to provide with great convenience and reliability both in the coding and application process.

2. Lexical conventions

Five kinds of tokens are included in EasKey: identifiers, keywords, constants, strings and expression operators. Blanks, tabs, newlines and comments can only serve to separate tokens, otherwise they will be ignored. Tokens are distinguished by adding at least one of the characters above to separate them.

2.1. Comments

Any character between a /* and the first */ after that are considered as a comment. Nested /* and */ pairs are illegible.

2.2. Identifiers (Names)

An identifier is a sequence of letters and digits; the first character must be alphabetic. The underscore ‘‘_’’ counts as alphabetic. Upper and lower case letters are considered different.

2.3. Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

int	*float*
char	*string*
boolean	*point*

color	*key*
if	*elseif*
else	*while*
for	*continue*
break	*switch*
case	*default*
function	*return*
do	*end*
true	*false*

2.4. Constants

There are several kinds of constants as follows.

2.4.1. Integer constants

An integer constant is a sequence of digits.

2.4.2. Character constants

A character constant is one or two characters enclosed in single quotes ‘‘′ ’’. Within a character constant a single quote must be preceded by a backslash “\’’. Certain non-graphic characters, and ‘‘\’’ itself, may be escaped according to the following table:

back space	\b
newline	\n
carriage return	\r
horizontal tab	\t
null byte	\0
\	\\

2.4.3. Floating constants

A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (not both) may be missing; either the decimal point or the e and the exponent (not both) may be missing.

2.4.4. Boolean constants
A boolean constant only has two possible values: true or false.

2.4.5. Point constants
A point constant is two integer constants separated by a comma “,” and enclosed in parentheses “{”, “}”, indicating the coordinates of a pixel in the computer screen.

2.4.6. Key constants
A key constant is one or more characters enclosed in “$”, indicating the most of the keys of an ordinary keyboard.

2.4.7. Color constants
A color constant is six characters enclosed in “#”, indicating the color of pixel in the computer screen. The six characters must be from ‘0’ to ‘9’ or from ‘a’ to ‘f’, representing three hexadecimal numbers in a RGB color model.

2.4.8. String constants
A string is a sequence of characters surrounded by double quotes “"”. String itself is a type, which is different from the type array-of-characters. In a string, the character “"” must be preceded by a ‘‘\’’; in addition, the same escapes as described for character constants may be used.

3. Conversions
EasKey has the following built-in functions for type conversions.

3.1.1. Characters and integers
char_of_int(int) returns a character whose ASCII value is the integer parameter. int_of_char(char) returns an integer with the ASCII value of the character parameter.

3.1.2. Float and integers
float_of_int(int) returns a float with the value of the integer parameter. int_of_float(float) returns an integer with the value of the floor of the float.

3.1.3. String and boolean
string_of_bool(bool) returns “true” of “false” depending on the boolean parameter. bool_of_string(string) returns a boolean from the given string, which must be “true” or “false”. Raise failure if the given string is not a valid representation of a boolean.

3.1.4. String and float
string_of_float(float) returns string from the given float. float_of_string(string) returns a float from the given string. Raise failure if the given string is not a valid representation of a float.

3.1.5. String and integers

string_of_int(int) returns a string from the given integer. int_of_string(string) returns an integer from the given string. Raise failure if the given string is not a valid representation of an integer.

4. Expressions

The precedence of expression operators is the same as the order of the major subsections of this section (highest precedence first).

4.1. Primary expressions

Primary expressions include the following.

4.1.1. Identifier

An identifier is a primary expression, provided it has been suitably declared as discussed below. Its type is specified by its declaration.

4.1.2. Constant

Any constant is a primary expression.

4.1.3. (*expression*)

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.

4.1.4. *primary-expression* [*expression*]

A primary expression followed by an expression in square brackets is a primary expression. It is specifically used for the accessing an element in one array.

4.1.5. *primary-expression* ($\textit{expression-list}_{opt}$)

A function call is a primary expression followed by parentheses containing a possibly empty, comma separated list of expressions which constitute the actual arguments to the function. The result of the function call is of return type of that function.

4.2. Unary operators

Expressions with unary operators group right-to-left.

4.2.1. *–expression*

The result is the negative of the expression, and has the same type. The type of the expression must be char, int, or float

4.2.2 *!expression*

The result of the logical negation operator ! is true if the value of the expression is false, false if the value of the expression is true. Note that this operator is applicable to int or char type.

4. 3 Additive operators

The additive operators + and – group left or right.

4.3.1 expression + expression

The result is the sum of the expressions. If both operands are int or char, the result is int. If both are float, the result is float. If one is char or int and one is float, the former is converted to float and the result is float.

4.3.2 expression – expression

The result is the difference of the operands. If both operands are int, char, or float, the same type considerations as for + apply.

5.3 Multiplicative operators

The multiplicative operators *, /, and % group left or right. For all multiplicative operators, all operand need to be explicitly conversed. Implicitly conversion is not permitted. EasKey offers functions to help type conversion, as stated before.

*5.3.1 expression * expression*

The binary * operator indicates multiplication. If both operands are int or char, the result is int; if one is int or char and one float, the former is converted to float, and the result is float; if both are float, the result is float. No other combinations are allowed.

5.3.2 expression / expression

The binary / operator indicate division. The same type considerations as for multiplication apply.

5.3.3 expression % expression

The binary % operator obtains the remainder from the division of the first expression by the second. Both operands must be int or char, and the result is int.

5. Relational operators

The relational operators group left or right. Relational operators cannot be used sequentially. For example, ‘‘a<b<c’’ is not permitted by EasKey.

expression < expression;

expression > *expression;*

expression <= *expression;*

expression >= *expression*

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) return boolean type. Operand conversion is exactly the same as for the +operator.

6. Equality operators

EasKey do not have pointer, the equality operators only operates the contents.

6.1 expression == ex*pression; expression* != *expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence.

*6.2 expres*sion && e*xpression*

The &&operator returns true if both its operands are true, false otherwise. &&guarantees left or right evaluation; moreover the second operand is not evaluated if the first operand is false.

6.3 expression || *expression*

The || operator returns true if either of its operands is true, false otherwise. || guarantees left or right evaluation; moreover, the second operand is not evaluated if the value of the first operand is true. The operands need not have the same type.

7. Assignment operators

There are a number of assignment operators, all of which group right to left. All require an lvalue as their left operand, and the type of an assignment expression is that of its left operand. The value is the value stored in the left operand after the assignment has taken place.

7.1 lvalue = *expression*

The value of the expression replaces that of the object referred to by the lvalue. The operands must have the same type. The value of the expression is simply stored into the object referred to by the lvalue.

7.2 lvalue += *expression;*

lvalue –= expression;

*lvalue *= expression;*

lvalue /= *expression;*

lvalue %= *expression*

The behavior of an expression of the form ‘‘E1 op= E2’’ may be inferred by taking it as equivalent to

‘‘E1 = E1 op E2’’; however, E1 is evaluated only once.

8. Declarations

Declarations are used within function definitions to specify the interpretation of each identifier. Declarations have the form

declaration:

decl-specifiers declarators**;**

The declaratory is the identifier being declared. Unlike C or Java or other languages, EasKey does not have storage class specifier; it only consists of type-specifier and it only consists of at most one type-specifier.

8.1 The type-specifiers are:

type-specifier:

int

char

float

string

boolean

key

point

color

Examples:

int a (Uninitialized Integer identifier will be set to 0 as default value)

char b (Uninitialized char identifier will be set to ‘\0’ as default value)

string c (Uninitialized string identifier will be set to “\0” as default value)

float d (Uninitialized float identifier will be set to 0.0 as default value)

color e (Uninitialized color identifier will be set to white, which represents a RGB form #ffffff#, as default value)

key f (Uninitialized key identifier will be set to $null$, as default value)

point g (Uninitialized point identifier will be set to tuple {0,0}, including two integers within “{“ and”}”. Point represents the coordinate of point in 2D space)

int h=10 (will initialize the value of h to 0)

key i=A (It means i is initialized to A, which is the key A in the keyboard)

8.2 Declarators

The specifier in the declaration indicate the type of the objects to which the declarator refer.

Declarators have the syntax:

```
declarator:
    identifier
    declarator ( )
    declarator [ constantexpressionopt ]
```

8.3 Meaning of declarators

Each declarator is taken to be an assertion that when a construction of the same form as the declarator appears in an expression, it yields an object of the indicated type. Each declarator contains exactly one identifier; it is this identifier that is declared.

If an unadorned identifier appears as a declarator, then it has the type indicated by the specifier heading the declaration.

For example,

```
int a;
```

means that the identifier a, which is to be declared, has the type of int.

If a declarator has the form

    D( )

then the contained identifier has the type "function returning …", where "…" is the type which the identifier would have had if the declarator had been simply D.

A declarator may have the form

    D[constant-expression]

or

    D[ ]

In the first case the constant expression is an expression whose value is determinable at compile time, and whose type is int. In the second the constant 0 is used. Such a declarator makes the contained identifier have type "array". If the unadorned declarator D would specify a non-array of type "…", then the declarator "D[i]" yields a one-dimensional array with rank i of objects of type "…".EasKey does not support multi-dimensional array.

An array may be constructed from one of the basic types, such as int, float, string, etc.

9. Constant expressions

In several cases it requires expressions which evaluate to a constant: after case or as array bounds. In these two cases, the expression can involve only integer constants, and character constants, possibly connected by the binary operators

    +   –   *   /

or by the unary operators

    –

10. Built-in functions

The built-in functions are classified into three categories: (1) Mouse operations; (2) Key operations; (3) Other functions.

10.1 Mouse operations

10.1.1 *left_click (point, time)*

Click the left mouse button on a designated point for a specific time.

10.1.2 *right_click (point, time)*

Click the right mouse button on a designated point for a specific time.

10.1.3 *move (point)*

Move the mouse from current point to a designated point.

10.1.4 *double_click (point)*

Double click the mouse on a designated point.

10.1.5 *left_down ()*

Press down the left mouse button.

10.1.6*right_down ()*

Press down the right mouse button.

10.1.7 *left_up()*

Let go the left mouse button.

10.1.8 *right_up()*

Let go the right mouse button.

10.2 Keyboard operations

10.2.1 *key_down(key)*

Press down a designated key on the keyboard.

10.2.2 *key_up(key)*

Let go the key.

10.2.3 *key_stroke(key)*

Stroke a designated key on the keyboard.

10.2.4 *combo(key1,key2,key3,key4)*

This function supports at most 4 different keys as arguments at a time by the means of overload. It presses the first three keys by calling function *key_down(key)*, then strokes the fourth key by calling function *key_stroke(key)* to implement the whole process.

10.3 Other functions

10.3.1 *getcolor(point)*

Get the color of a designated point.

10.3.2 *getareacolor(point, int)*

Get the color of a specific circle area with a designated center point and a default radius. The color is the average of the color of several sample points within this area.

10.3.3 *print(string)*

Print out designated content.

11. Statements

Except as indicated, statements will be executed in sequence within certain Blocks, and the execution of Statements in different Block will also follow the sequence of block appearance.

11.1. Expression Statement

Most statements are expression statements, which have the form

    expression ;

Usually expression statements are assignments or function calls.

Expression statement is a construct consisting of variables, operators, certain real value and even functions.

Examples:

```
int a=32

int a=b+32+c;

point p={2,4}+g;
```

11.2. Conditional Statement

There are 4 forms of conditional statement:

```
if (expression) then (statement) end if
if (expression) then (statement) else (statement) end if
if (expression) then (statement) elseif(expression) then (statement) end if
if (expression) then (statement) elseif (expression) then (statement) else (statement) end if
```

Each keyword if must end with up with a keyword end if. The elseif and the else after the keyword if are optional. With these forms, the body of a conditional statement are covered in a pair of if and end if, with no exception. Besides, each keyword end if will be paired with the last encountered if.

If the outcome of first expression is Boolean true then the first statement will be executed, else the following statement with Boolean true expression outcome will be executed.

Example:

```
1. if (a==b) then a=0 end if;
2. if (a==b) then a=0 else a=1 end if;
3. if (a==b) then a=0 elseif (a<b) then a=1 end if;
4. if (a==b) then a=0 elseif (a<b) then a=0 else a=1 end if;
5.  if (a==b) then
 if (a>c) then a=0
           end if
   end if;
```

11.3. While Statement

The while statement has the form:

```
while ( expression )
do
statement
end while
```

The substatement will be executed repeatedly so long as the value of the expression remains true.

Example:

```
while (a>b)
    do
     c=a;
     a=b;
     b=c;
    end while
```

The statement means a single statement or a sequence of statements, which are placed in the block between do and end while, will be executed consecutively if the condition in expression is true.

11.4. For Statement

The for statement has the form

```
for ( expression1; expression2; expression3)
do
statement
end for
```

It is equivalent to

```
expression1;
while ( expression2)
do
statement
expression3;
end while
```

The first expression specifies initialization for the loop. The second expression specifies a conditional judgment, which is tested before each iteration. The loop exits when the second expression is false; the third expression specifies an incrementation or decrementation which is performed after each iteration.

Example:

```
for(int i=0;i<100;i = i+1)
    do
      a=a+i;
end for
```

11.5. Switch Statement

The switch statement causes control to be transferred to one of several statements depending on the value of an expression. It has the form:

```
switch ( expression )
do
statement
end switch
```

The expression outcome can be any type other than float. The statement is typically a list of statements and each statement within the statement may follow a case prefixes as follows:

```
case constant-expression:
```

where the constant expression can be any type other than float. No two of the case constants in a switch may have the same value.

There may also be at most one statement prefix of the form

```
default :
```

When the switch statement is executed, its expression is evaluated and compared with each case constant in an undefined order. If one of the case constants is equal to the value of the expression, control is passed to the statement following the matched case prefix. If no case constant matches the expression, and if there is a default prefix, control passes to the prefixed statement. In the absence of a default prefix none of the statements in the switch is executed.

11.6. Break Statement

The statement

break ;

causes termination of the smallest enclosing while, do, for, or switch statement; Or in other words, it will jump out of the inner loop.

11.7. Return Statement

A function returns to its caller according to the return statement, which has the following form:

return expression ;

Example:

```
 return 0;

 return a+b+c;

return a==b;
```

12. Examples

12.1. Basic function implementation

In this function, a dividend calculation function is implemented. Two doubles are passed into the function as parameters. Then the function calculates the dividend of the two doubles and returns the dividend which is a double value. If the second parameter is zero, the function will return an error message and return -1.0.

```
%calculate i divided by j

function float dividend(float i, float j)

if j==0 then print("error");return -1.0;

else

return i/j;

end if;

end function
```

12.2. Basic array operation

In this example, an integer array is passed to the function to calculate the sum of the elements inside the array. This function takes two arguments, the first one is the array and the second one is the size of the array. At the end of the function, the sum of the array is returned.

```
function sum(int s[], int num)
int i=0;
int sum=0;
for(;i<num;i=i+1)
sum=sum+s[i];
end for
end function
```

12.3. Area average color

In this function, basic point and color operations are shown. This function takes two point type variables as parameters and returns the average color value in the rectangle area with the two points as corner points. This function basically traverses every point in the region covered by the rectangle area and adds the color value together. In the end returns the average color value in the region.

```
function color avg(point i, point j)
color total=0;
int points=(j.y-i.y)*(j.x-i.x);
if
while(i.x<j.x)
do
while(i.y<j.y)
do
point nc=(i.x,i.y);
color=color+getcolor(nc);
end while
i.x=i.x+1;
```

```
end while
return total/points;
end function
```

12.4. Drag window to a different position

In this function, two point type parameters are passed to the function to implement the action that move a dialog window from point A to point B. This function left presses the mouse and keeps the mouse down to drag the window to another position.

```
function void dragwin(point origin, point desti)
left_down(origin);
move(desti);
left_up;
end function
```

12.5. Enter a string into a textbox

In this function, a point textb and a string msg are passed to the function to implement the function that enters the string msg into the textbox pointed by the textb point. The function left click the region inside the textbox to focus on the textbox and then enter the string msg into the box by traversing each character in the string.

```
function void enter(point textb, string msg)
%point textb is the point within the area of the textbox
left_click(point);
while msg.i!=eof do
key_stroke(msg.i);
delay(100);
end while
end function
```

12.6. Color dependent mouse click

In this function, a color type variable and a point pointing to the button are passed to the function to implement the action that if the color at or is the same as the red variable then click the button, else do not press the button.

```
function void press(color red, point or, point but)
if(getColor(or)==red)
then
left_click(but);
end if
end function
```