Concise Animation Language (CAL) Reference Manual

Authors

Tianliang Sun, ts2825

Xinan Xu, xx2153

Jingyi Guo, jg3421

Summary

The reference manual for Concise Animation Language (CAL).

Revisions:

Last updated: 10/28/2013 8:47:00 PM

Table of Contents

1. Introduction

CAL is a compact programming language used to facilitate the process of making animation. By providing minimal language support for users to create animation, CAL gives users the ability to easily and precisely create simple animations.

The syntax of CAL is C-alike, but it removes a lot of redundancies in C which will not be used for developing simple animations. CAL includes primitive data types such as int, double, and string. It also includes built-in data structures such as Point, Shape, and Image. With the basic control flow primitives like for-loop and if-else statements, users can program almost all graphics related algorithms in this language.

2. Lexical Convention

A CAL program consists of a number of categories of tokens, including Comments, Identifiers, Reserved Keywords, Constants, Strings, Expression Operators, and other separators like some special punctuation. A whitespace of any form (tab, newline, single or multiple spaces) is meaningless but to separate tokens.

When parsing a CAL program, if the input stream has been parsed up to a given character c, the next token will be the longest string of characters starting from c that could be matched as a legitimate token.

2.1 Line Comment

A line of comment starts with // and ends with the \n character. All the text between // and the end of line will be ignored, including block comment tokens. The escape character does not have any effect in the comment line and will not make the line comment splitting into multiple lines.

2.2 Block Comment

A comment block starts with one /* and ends with the next following */. A comment block could be either a single-lined or a multiple-lined one. However, CAL does not support nested comments. With that being said, a /* will be paired with the first */ in the following input. So the following example will not be a correct comment:

```
/* This is a comment. /* Nested Comment? */ Not a comment anymore! */
```

Only “/* This is a comment. /* Nested Comment? */” will be considered as a comment. The rest part will possibly cause a parsing error. The escape character does not have any effect in the comment block.

2.3 Identifiers

An identifier is to represent a data in the program. An valid Identifier has the form of [’a-z’,’A-Z’][‘0-9’,’a-z’,’A-Z’]{0,19}, which means it consists of a string of alphabetic letters and digits of length 1 to 20. It must starts with an alphabetic letter. It is also case-sensitive.

2.4 Reserved Keywords

CAL inherits the minimum set of reserved keywords from C to support graphics programming, which includes the keywords for primitive data types:

> int
> double
> string
> list
> point
> shape
> image
> struct

and keywords for control flow:

> break
> return
> if
> else
> for

Note that we use string instead of char and char* in CAL. All those keywords are reserved and no other tokens should be identical to them.

2.5 Constants

There are several kinds of constants of follows:

2.5.1 Integer Constants

An integer constant is a continuous sequence of numerical digits [‘0’-’9’]+. All numbers in CSL are in decimal representation.

2.5.2 String Constants

An string constant is a sequence of characters enclosed by a pair of “ and ”. The characters in a string constant could be anything (e.g. letters, digits, punctuations) but certain characters have to be escaped by a preceding \. Those characters are \, “, and ”. When parsing a string and a \ is encountered, it will be used to escape the next following character. If the escaped character is \, “, or ”, then that character will appear in the string. If the next character is n, t, or r (i.e. a \n, \t, \r in the string), it will be considered as a newline, tab, or return, respectively.

2.5.3 Double Constants.

Those are floating point values. A double consists of an signed integer part, a decimal point, a fraction part, and an optional e and signed exponent. The first three parts must present. A regular expression for a double constant is ‘-’{0,1}[‘0’-‘9’]+‘.’[‘0’-‘9’]+(‘e’ ‘-’{0,1}[‘0’-‘9’]+){0,1}.

3. Syntax Notation

In syntax notations throughout this manual, syntax categories are distinguished by *Italic* type. Each alternative expansion is listed in a separate line. All the optional symbols are labeled with a subscript “$_{\text{opt}}$”.

4. What’s in a Name

Being different from C, an identifier has the form identifier_type identifier_name.

The type determines the meaning of the values found in the identifier’s storage. CAL does not have storage classes. All variables are visible to their containing functions and anything within the scopes of containing functions, and all variables are allocated on the stack.
There are three primitive types in CAL: int, double, string.
From the primitive types, one can construct more complex types like:

list: a list of variable length that contains variables of a same type
function: which will return a value of the given type
struct: a container for variables of different types

CAL supports recursive construction of list and struct, but not function.

5. Objects and lvalues

An object is a manipulative region of memory. A lvalue is an expression referring to an object. “lvalue” is shorthand for left hand side values. In an assignment expression “E1 = E2”, E1 must be a lvalue expression. Later in section “Expressions”, for every operator we discuss whether it expects operands of lvalue type, and whether it yields a result of lvalue type.

6. Conversions

CAL does not provide automatic conversions between types. The operands of an operator or the arguments to a function must match the expected types. However, functions that can do primitive type conversions (e.g. double int_to_double(int i)) will be provided.

7. Expressions

Following the convention of traditional programming languages, expression operators are categorized into different subsections, each associated with a precedence level. Within each subsection, the operators have the same precedence. Left- or right-associativity is specified in each subsection for the operators discussed therein.

Precedence	Operator	Description	Associativity
1	++ --	Suffix/postfix increment and decrement	Left-to-right
	()	Function call	
	[]	Array subscripting	
	.	Element selection by reference	
2	++ --	Prefix increment and decrement	Right-to-left
	−	Unary minus	
	!	Logical NOT	
3	* / %	Multiplication, division, and remainder	Left-to-right
4	+ −	Addition and subtraction	
5	< <=	For relational operators < and ≤ respectively	
	> >=	For relational operators > and ≥ respectively	
6	== !=	For relational = and ≠ respectively	
7	&&	Logical AND	
8	||	Logical OR	
9	=	Direct assignment	Right-to-left
	+= −=	Assignment by sum and difference	
	*= /= %=	Assignment by product, quotient, and remainder	
10	,	Comma	Left-to-right

7.1 Primary expressions

7.1.1 *identifier*

The identifier, whose regular expression has been defined in the section 2.2, is a primary expression representing an object. The type of this object has to be declared explicitly.

7.1.2 *constant*

An integer, string, double with a fixed value is a primary expression. The regular expression of those three types has been defined in the previous sections.

7.1.3 *(expression)*

A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The presence of parentheses does not affect whether the expression is an lvalue.

7.1.4 *primary-expression [expression]*

List dereferencing: a primary expression followed by an expression in square brackets is a primary expression. The meaning of the brackets is to dereference a list to the object offset by the expression within the square brackets, from the start point of the list.

7.1.5 *primary-expression (expression-list)*

Function call: 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 type of the expression list should exactly match the types one will pass into. If not, explicit type cast has to be done to successfully make the function call.

7.1.6 *lvalue . identifier*

An lvalue expression followed by a dot followed by the name of a member of a structure is a primary expression. The object referred to by the lvalue is assumed to have the same form as the structure containing the structure member. The result of the expression is an lvalue appropriately offset from the origin of the given lvalue whose type is that of the named structure member.

7.2 Unary operations

7.2.1 − *expression*

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

7.2.2 *! expression*

The result of the logical negation operator ! is 1 if the value of the expression is 0, and 0 otherwise. The type of the expression must be int, and the return type will be int.

7.2.3 *++ lvalue-expression*

The object referred to by the lvalue expression is incremented. The value is the new value of the lvalue expression and the type is the type of the lvalue. The type of the expression is int, and it increments by 1.

7.2.4 -- *lvalue-expression*

The object referred to by the lvalue expression is decreased. The value is the new value of the lvalue expression and the type is the type of the lvalue. The type of the expression is int, and it decreases by 1.

7.2.5 *lvalue-expression ++*

The result is the value of the object referred to by the lvalue expression. After the result is noted, the object referred to by the lvalue is incremented in the same manner as for the prefix ++ operator.

7.2.6 *lvalue-expression* --

The result is the value of the object referred to by the lvalue expression. After the result is noted, the object referred to by the lvalue is incremented in the same manner as for the prefix -- operator.

7.3 Multiplicative operators

The multiplicative operators *, /, and % group left-to-right.

7.3.1 *expression * expression*

The binary * operator indicates multiplication. If both operands are int, the result is int; if either one is double, the result will be double.

7.3.2 *expression / expression*

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

7.3.3 *expression % expression*

The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be int and the result will be int. The remainder has the same sign as the dividend.

7.4 Additive operators

7.4.1 *expression + expression*

The result is the sum of the expressions. If both operands are int, the result is int. If either is double, the result is double. If both operands are string, the result, which returns another copy, is the concatenation of both strings.

7.4.2 *expression - expression*

The binary - indicates minus. If both operands are int, the result is int. If either is double, the result is double.

7.5 Relational operators

7.5.1 *expression < expression*

7.5.2 *expression > expression*

7.5.3 *expression <= expression*

7.5.4 *expression >= expression*

The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true. If one side involves a double type and another side is int, it will be converted into double for comparison. string has to be compared to string only. Belowed boolean operators also follow this rule.

7.6 Equality operators

7.6.1 *expression == expression*

7.6.2 *expression != expression*

The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators.

7.7 expression && expression

The && operator returns 1 if both operands are non-zero, and returns 0 otherwise. Moreover, the second operand is not evaluated if the first operand is 0. Either side accepts int only.

7.8 expression || expression

The || operator returns 1 if either of its operands is non-zero, and 0 otherwise. Moreover, the second operand is not evaluated if the value of the first operand is non-zero. Either side accepts int only.

7.9 Assignment operators

7.9.1 *lvalue = expression*

The value of the expression replaces that of the object referred to by the lvalue. The operands need to have the same type.

7.9.2 *lvalue += expression*

7.9.3 *lvalue -= expression*

7.9.4 *lvalue *= expression*

7.9.5 *lvalue /= expression*

7.9.6 *lvalue %= expression*

The behavior of an expression of the form ‘E1 op = E2’ is equivalent to ‘E1 = E1 op E2’

7.10 expression , expression

A pair of expressions separated by a comma is evaluated left-to-right and the value of the left expression is discarded. The type and value of the result are the type and value of the right operand.

8. Declarations

Declarations in CAL appear at the beginning of a function body and they are to tell C how it should allocate and possibly initialize the identifiers associated with them. The declaration discussed here are only data declarations. Function declarations are specified later in this document.

Declarations have the form:

declaration-list:
- *declaration*
- *declaration declaration-list*

declaration:
- *type-specifier declarator-list*$_{opt}$ *;*

8.1 Type specifiers

Type specifiers in CAL have the form:

type-specifier:
- *int*
- *double*
- *string*
- *list<type-specifier>*
- *point*
- *shape*
- *image*
- *struct identifier*

struct identifier { declaration-list }

8.2 Declarators

Slightly different from the syntax in raw C, CAL allows declaring and initializing an identifier at the same time, and it is strongly suggested to initialize a variable upon declaration.

```
    declarator-list:
        declarator
        declarator, declarator-list

    declarator:
        identifier initializer_opt

    initializer:
        = constant
        = { constant-expression-list }
```

8.3 Structure Declarations

Structure declarations have two forms as noted previously:

struct identifier { type-decl-list }
struct identifier

When declaring a structure for the first time, always use the first form to define the data layout in this structure. Then the subsequent declarations of this structure must use the second form.

9. Statements

9.1 Expression Statement

The basic form of a statement is an expression followed by a semicolon:

statement:

expression ;

Assignments and function calls are most commonly used expression statement.

9.2 Statement group

A group of statements can appear wherever a single one is allowed and it has the form:

```
statement-group:
        { statement-list }

statement-list:
        statement
        statement statement-list
```

9.3 Conditional statement

The two forms of conditional statement are:

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

First the expression is evaluated, if it is non-zero, the first statement is executed. In the second situation, if the expression is zero, then the statement after else is executed. In addition, the “else” ambiguity is resolved by connecting an “else” with the nearest elseless “if”.

9.4 For statement

For statement has the form:

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

The first expression initializes the for loop; it's executed once when the loop begins; the second expression makes a test before each iteration and the loop is exited when the expression evaluates to zero; the last expression is invoked after each iteration through the loop.

9.5 Break statement

The statement

> *break ;*

terminates the current for loop and control passes to the statement following the terminated statement.

9.6 Return statement

A function returns to its caller by means of return statement as follows:

> *return expression ;*

When it is used, it will return to its caller with one or multiple values which could have different types.

10. Function Definitions

Function definitions must take the form:

> *function-definition:*
>
> > *type-specifier function-declarator function-body*

However, unlike C which does not allow functions to return structures, CAL does allow functions returning structures.

The function declarator takes the form:

> *function-declarator:*

identifier(param-list$_{opt}$)

parameter-list:

function-parameter

function-parameter, parameter-list

function-parameter:

type-specifier identifier

(CAL does not allow passing functions as parameters to a function)
And the body of a function takes the form:

function-body:

{ declaration-list$_{opt}$ statement-list }

11. Scope Rules

11.1 Variables

A variable is visible and accessible within the body of the function it is declared in (a.k.a the containing function), and only accessible by the code after its declaration.

11.2 Functions

Functions must be defined at the outermost scope in a program, and therefore nested function declaration is not supported in CAL. Functions can be accessed globally within the program it is declared in, but only after its declaration.

11.3 Structures

Structures must be defined at the outermost scope in a program. Although a structure may recursively contains a structure, but the definition of the contained structure must be again at the outermost scope in a program.

12. Types Revisited

This section summarizes the operations that can be done on a given type.

12.1 Structures

Use . to select a member in the structure. Structures can also be passed as function parameters. Structures cannot be directly assigned since it is not a lvalue type.

12.2 Point, Shape, Image

Those three types are actually predefined structures in CAL. So the only difference between these three and other structures is that there is no need to use the first form of structure declaration

struct identifier { declaration-list }

when declaring them, because their data layout are specified already. Even the data members in those structures are initialized already, but those initial values are usually useless. Use the predefined initialization function to adjust those values.

12.3 Lists

List is also similar to a vector in C. One operation that can be performed on a list is to use *[expression]* to access the element at the offset specified by *expression*.

12.4 Functions

The only operation that is available to a function is to call it. A function can neither be passed as a parameter, nor directly assigned.

Appendix: Syntax Summary

1. Expressions

expression:

primary

− *expression*

! expression

++ lvalue-expression

-- *lvalue-expression*

lvalue-expression ++

lvalue-expression --

expression binop expression

lvalue asgnop expression

expression , expression

primary:

identifier

constant

(expression)

primary-expression [expression]

primary-expression (expression-list)

lvalue . identifier

binop:

*

/

%

+

-

<

>

<=

>=

==

!=

&&

||

asgnop:

=

+=

-=

**=*

/=

/=

2. Declarations

declaration-list:

declaration

declaration declaration-list

declaration:

type-specifier declarator-list$_{opt}$ *;*

type-specifier:

int

double

string

list<type-specifier>

point

shape

image

struct identifier { declaration-list }

struct identifier

declarator-list:

declarator

declarator, declarator-list

declarator:

identifier initializer$_{opt}$

initializer:

= constant

= { constant-expression-list }

3. Statements

statement:

expression ;

statement-group:

{ statement-list }

statement-list:

statement

statement statement-list

if (expression) statement

if (expression) statement else statement

for(expression1; expression2; expression3) statement

break ;

return expression ;

4. Function Definitions

function-definition:

type-specifier function-declarator function-body

function-declarator:

identifier(param-list$_{opt}$*)*

parameter-list:

function-parameter

function-parameter, parameter-list

function-parameter:

type-specifier identifier

function-body:

{ declaration-list$_{opt}$ *statement-list }*