PLT Fall 2018

C Something? Language Reference Manual

Tong Liu (tl2871)
Zhuohao Li (zl2630)
Zixiong Liu (zl2683)
Madeline Zechar (mez2113)

October 31st, 2018

1 Lexical Elements

C Something? considers six kinds of tokens: identifiers, keywords, constants, strings, expression operators, and other separators.

Comments

The characters /* begin a comment and */ end a comment. Comments are ignored by the compiler.

Identifiers

Identifiers begin with a letter and are directly followed by a sequence of digits and letters. C Something? distinguishes identifiers that begin with a lowercase letter from identifiers that begin with a uppercase letter. The underscore is considered a lowercase letter.

Keywords

The following identifiers are reserved as keywords and cannot be used otherwise:

int	break	mat	continue	if	float
else	while	for	case	return	

Operators

An operator is a symbol that performs a certain mathematical or logical manipulation. Full coverage of C Something? can be found in part 3.

Constants

C Something has both integer and real number constants. An integer constant is a sequence of digits. We only support decimal base. A real number constant represents a fractional number value. A real number constant involves a sequence of digits representing the integer, followed by a decimal point, which is then followed by a sequence of digits representing the fraction.

Separators

Separators are used to separate one token from another (keyword from other keyword, keyword from identifier, identifier from other identifier, etc). C Something? separators are the following:

(	)	[	]	{	}	;	,	.	:

White Space

White space refers to spaces, tabs, newlines, vertical tab character,the form-feed character. White space is a separator but not a token. In this way, the compiler ignores white space except when it used to separate tokens. At least one whitespace is required to separate adjacent adjacent identifiers, constants, and specific operator pairs. Whitespace is not required between operators and operands or separators and the tokens they separate.

2 Types and Data

Float

A variable of type float is defined as an IEEE 754 32 bit float point number. Example: float pi = 3.1415.

Double

A variable of type double is defined as an IEEE 754 64 bit float point number. Example: float pi = 3.1415.

Int

A variable of type int is defined to be a signed 64 bit integer. A literal integer is interpreted as having the type int. Example: int a = 32.

Array

An array of type T of length n has the type signature T[n]. It is stored consecutively in memory. Array literals have the syntax {x1, x2, x3, …}. Example: float[5] float_array = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0}.

Matrix

A matrix of type T of dimension m*n has the type signature T[m, n]. It is stored in column major order in memory. Matrix literals have the syntax such as {{1,2,3}, {4,5,6}, {7,8,9}}. Example: float[3,3] float_mat = {{0.0, 1.0, 2.0}, {3.0, 4.0, 5.0}, {6.0, 7.0, 8.0}}.

3 Expressions and Operators

Expressions

In C something, an expression is any legal combination of symbols that represents a value.

Arithmetic Operators

add	minus	multiply	divide	mod
+	-	*	/	%

Assignment Operators

assign
=

Matrix Operators

add	minus	multiply	determinant	inverse
+	-	*	det()	inv()

Comparison Operators

Equal to	Not Equal to	Greater than	Less than
==	!=	>	<

Greater than or equal to	Less than or equal to

>=	<=

Logical Operators

Logical AND	Logical OR
&&	||

Operator Precedence

OPERATOR	DESCRIPTION	ASSOCIATIVITY
( )	Parentheses	Left-to-right
[ ]	Brackets	
+ –	Unary plus/minus	Right-to-left
Sizeof Det Inv	Size-of Determinant of a matrix Inverse of a matrix	
* / %	Multiplication, division, and remainder	Left-to-right
+ -	Addition and subtraction	
< <= > >=	For relational operators < and ≤ respectively For relational operators > and ≥ respectively	
== !=	For relational = and ≠ respectively	
&&	Logical AND	
||	Logical OR	
=	Assignment	Right-to-left
,	Comma	Left-to-right

Statements

Declaration Statement

A declaration statement is a declaration ended with a semicolon.

Expression Statement

An expression statement is an expression ended with a semicolon.

Block Statement

A block statement is any number of statements surrounded by braces. It does not return a value. It doesn't and shouldn't end with a semicolon.

Conditional Statement

A conditional statement has the syntax "if (expression) statement". It can be optionally followed by "else statement". The statement in the if branch is executed if expression is true, and otherwise the statement in the else branch is executed (if there is one).

While Statement

A while statement has the syntax "while (expression) statement". It evaluates expression each time before executing statement, and it continues if and only if expression evaluates to true.

For Statement

A for statement has the syntax "for (declaration-or-expression-1; expression-2; expression-3) statement". It is equivalent to:

```
{
      Declaration-or-expression-1;
      while (expression-2) {
            statement
            expression-3;
      }
}
```

Switch Statement

A switch statement has syntax:

```
switch(expression) {
      case x1: statement
```

```
    case x2: statement
    …
    default: statement
}
```
The default statement is mandatory. The individual cases do fall through, so a break statement will be needed after each case.

Break Statement
A break statement has syntax “break;”. It redirects execution to the first statement after the innermost loop or switch statement.

Continue statement
A continue statement has syntax “continue;” It redirects execution to the next iteration of the innermost loop statement.

Return statement
A return statement has syntax “return;” or “return x;”. It returns control to the caller of the current function.

Functions
Functions separate a program into distinct subroutines. A function definition is required in order to write a function. Every program must have the main function, which is where program execution begins.

Function Declarations
A function declaration specifies a function name, input parameters, and return type. The return type describes the data type of the value returned by the function. A function that does not return a data type has a `void` return type. A function declaration has the form:
`return-type function-name(input-parameters);`

The function name is a valid identifier. The input parameters consists of zero or more parameters separated by commas. A single parameter consists of a data type and a valid identifier.

For example:
```
mat multiply_mat(mat a, mat b);
```

Function Definitions

A function definition indicates what the function does. A function definitions consists of the return type, function name, input parameters, and the body of the function. Function definitions have the form:

```
return-type function-name (parameter-list)
{
     function-body
}
```

Function Parameters

C Something functions exchange information by means of parameters and arguments. The term parameter refers to any declaration within the parentheses following the function name in a function declaration or definition; the term argument refers to any expression within the parentheses of a function call.

The following rules apply to parameters and arguments of C Something functions:

1. The number of arguments in a function call must be the same as the number of parameters in the function definition. This number can be zero.

2. Arguments are separated by commas. However, the comma is not an operator in this context, and the arguments can be evaluated by the compiler in any order. There is, however, a sequence point before the actual call.

3. Arguments are passed by reference; that is, when a function is called, the parameter receives a reference of the argument's value. This rule applies to all values.

Main Function

Every program begins executing in the main function. As such, the main function must be included in every program. The main function must be defined

though it does not need a function declaration. The return type for main is always int and accepts no input parameters.
For example:

```
int main () {
	printf ("Hello World!");
	return 0;
}
```