COLOGO

A Graph Language

Reference Manual

Advisor: Stephen A. Edwards

Shen Wang (sw2613@columbia.edu)

Lixing Dong (ld2505@columbia.edu)

Siyuan Lu (sl3352@columbia.edu)

Chao Song (cs2994@columbia.edu)

Zhou Ma (zm2167@columbia.edu)

1. Introduction:

COLOGO language is an effective programming language for drawing 2D graphics. The COLOGO language is designed in spirit of low threshold, which enables easy entry by novices and yet meet the needs of high-powered users. We can use COLOGO for education as it contains basic computer concepts appropriate for beginners. We can also draw interesting pictures and design complicated logos with COLOGO so that the language could be widely used for entertainment or commercial area.

2. Lexical Conventions:

The first step to compile our language is lexical analysis. In this step, the imported files are added in, and the program is recognized as a sequence of tokens.

2.1 Character Set:

COLOGO supports ASCII character set.

2.2 Identifier:

An identifier is a sequence of letters and digits. There are several rules for our identifier. For instance, the first character of the identifier must be a letter. The underscore_ is also viewed as a letter. The upper and lower case letters are different in the identifier.

Identifiers may have different length, and at least the first 31 characters are significant for the internal identifiers while for some implementations more characters are significant. Internal identifiers include preprocessor marco names and all other names without external linkage. Identifiers with external linkage are more restricted

2.3 Comments:

Comments are introduced by (: and ended by :).
If you see the character -_- in one line, everything behind -_- in this line are comments. Comments are not allowed to be nested. When a comment starts with a (: , the comment will be ended by the next occurrence of :). In the line with a -_-, if another -_- appears

behind the first -_-, it would be omitted.

2.4 Keyword:

Keywords identify statement constructs and specify basic types. Keywords cannot be used as identifiers. The keywords are listed in Table 1.

Table 1: Keywords

FD	RT	LF	RESET	GLS
BK	RAND	PU	PD	PF
RGB	AND	OR	NOT	INT
REAL	OBJ	BOOL	BRK	CONDITION
FX	GOON	LOOP	RETURN	ELSE
ENDC	ENDL	STATIC	EXTERN	FUNC
TRUE	FALSE			

In general, keywords are separated into three categories:

1. Drawing functional
2. Logical operator
3. Variable type indication
4. Part of statement

2.5 Operators

COLOGO has 7 categories of operators. They are unary, cast, additive, multiplicative,

relational, logical and object reference operator, respectively

Unary

-	!

Cast

(INT)	(REAL)

Additive

+	-

Multiplicative

*	/	%

Relational

==	!=	<	<=	>	>=

Logical

AND	OR	NOT

2.6 Separators

COLOGO recognizes three types of separators of tokens. They are space, tab, new line. The compile considers no difference among them.

2.7 Syntax group

{}: Braces are delimiter of compound statement, used in the cases of statements block and constant array initialization

[]: Brackets are used for array index dereference

(): Parenthesis are for expression grouping and argument expression list

2.8 Sequential punctuator

, : used to separate arguments for function calls or array assignments.

; : used to separate statements.

3. Lvalue

Lvalue is an expression that refers to a region of storage. It is required by certain operators. Refer to the operator part to see which operators expect an lvalue.

4. Expression and Operators

4.1 Primary Expressions

Primary expressions are the identifiers, constants or expressions in parentheses.

primary-expression:

->identifier

->constant

->(primary-expression)

If an identiifier has been suitably declared as the following parts shown below, it would be viewed as a primary expression. The type of the identifier is specified by its declaration. And if an identifier's type is arithmetic or object this identifier would be an lvalue.Lvalue is an expression that refers to a region of storage. It is required by certain operators. Refer to the operator part to see which operators expect an lvalue.

A constant is a primary expression. Its type depends on its form as described before
A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression. The precedence of parentheses does not affect whether the expression is an lvalue.

4.2 Postfix Expressions

The operations in postfix expressions group left to right.

postfix-expression:

->primary-expression

->postfix-expression.identifier

->postfix-expression(argument-expression-list)

->assignment-expression assignment-expression-list, assignment-expression

4.2.1 Array Reference

A postfix expression followed by an expression in square brackets is a postfix expression denoting the index of the wanted element inside array, e.g. *array[expression]*

4.2.2 Function Calls

A function call is a kind of postfix expression, which we may name it the function designator,. Usually the function call is followed by parentheses containing a possibly empty, comma-separated list of assignment expressions which help to constitute the arguments to the function.

We use the term argument for an expression passed by a function call and use the term parameter for an input identifier received by function definition or described in a function declaration. In preparing for the call to a function, a copy is made of each argument; all argument-passing is strictly by value.

4.2.3 Object Reference

A postfix expression followed by a dot followed by an identifier is a postfix expression. The first operand expression must be an object, and the identifier must name a member of the object. The value is the named member of the object, and its type is the type of that member. Detailed information is discussed in later chapters.

5. Declarations:

Declarations create variables with several attributes: variable name, type, scope, variable value(optional).

5.1 Scope Specifier

There are three different scopes of variables.

Local: The variable can only be seen within the statement block.

Static: The variable can be seen within a file

External: The variable can be seen by other files.

5.2 Type Specifier

5.2.1 Primitive Types

There are three primitive types in COLOGO. They are declared as below:

INT id = value;

REAL id = value;

BOOL id = value;

Where id is the name of variable and value is an expression or a primitive value.

5.2.2 Array Type

For each primitive type, COLOGO has a corresponding array container. They are:

INT id[length];

REAL id[length];

BOOL id[length];

Where id is the name of variable and length is the number of elements contained in the array. The above form will initialize the array as zero for INT and REAL, and false for BOOL type. Other than this, the array can also be initialized with a constant array of the same type. For example:

INT id[2] = {1,2};

REAL id[5] = {1.2, 3,5, 10.2, 5.6, 7.8};

BOOL id[3] = {TRUE, FALSE, TRUE};

5.2.3 Object Type

COLOGO allows the user to integrate multiple primitive type and form a complex object type such that all the primitive type variables can be passed and referred to together. The

declaration are as follows:

OBJ id

{

primitive-declaration-list

}

Where id stands for the name of variable and the primitive-declaration-list stands for a list of primitive declaration in the form of primitive- declaration-1; primitive-declaration-2; etc.

6. Statements

In COLOGO, statements are executed in sequence. They fall into several groups.

Statement:

expression-statement

compound-statement

selection-statement

iteration-statement

jump-statement

6.1 Expression Statement

Most statements in COLOGO are expression statements, which have the form

expression-statement:

expression-opt;

Most expression statements are assignments or function calls. All side effects from the expression are completed before the next statement is executed. If the expression is missing, the construction is called a null statement; it is often used to supply an empty body to an iteration statement to place a label.

6.2 Compound Statement

So that several statements can be used where one is expected, the compound statement (also called “block”) is provided. The body of a function definition is a compound

statement.

compound-statement:

{ declaration-list-opt statement-list-opt }

declaration-list:

declaration

declaration-list

declaration

statement-list:

statement

statement-list

statement

If an identifier in the declaration-list was in scope outside the block, the outer declaration is suspended within the block, after which it resumes its force. An identifier may be declared only once in the same block. These rules apply to identifiers in the same name space; identifiers in different name spaces are treated as distinct.

Initialization of automatic objects is performed each time the block is entered at the top and proceeds in the order of the declarators. If a jump into the block is executed, these initializations are not performed. Initialization of static objects is performed only once, before the program begins execution.

6.3 Selection Statements

Selection statements choose one of several flows of control.

selection-statement:

CONDITION (expression) statement ENDC

CONDITION (expression) statement ELSE statement ENDC

In both forms of the CONDITION statement, the expression, which must have arithmetic or pointer type, is evaluated, including all side effects, and if it compares unequal to 0, the first substatement is executed. In the second form, the second substatement is executed if the expression is 0. The else ambiguity is resolved by introducing keyword ENDC.

6.4 Iteration Statements

Iteration statements specify looping.

iteration-statement:

LOOP (expression(opt)) statement

In the LOOP statement, the parameter expression must have BOOL type; it is evaluated before each iteration, and if it becomes equal to FALSE, the LOOP is terminated. Side-effects from each expression are completed immediately after its evaluation.

6.5 Jump Statements

A GOON statement may appear only within an iteration statement. It causes control to pass to the loop-continuation portion of the smallest enclosing such statement. More precisely, within each of the statements

LOOP (...) { …; GOON; }

A BRK statement may appear only in an iteration statement or, and terminates execution of the smallest enclosing such statement; control passes to the statement following the terminated statement.

A function returns to its caller by the RETURN statement. When RETURN is followed by an expression, the value is returned to the caller of the function. The expression is converted, as by assignment, to the type returned by the function in which it appears. Running to the end of a function is equivalent to a return with no expression. In either case, the returned value is undefined.

6.6 Draw Statements

Draw statements include operations for the turtle, which provide the drawing functionality. We have 9 kinds of drawing statements.

FD expression; : Turtle move forward expression distance.

BK expression; : Turtle move backward expression distance.

LF expression; : Turtle turn left expression radian.

RT expression; : Turtle turn right expression radian.

RESET; :Reset the turtle to original position.

CLS; : Clear the screen.

PF; : Pen flip.

PD; : Pen down.

PU; : Pen up

7. Scope

A program need not all be compiled at one time: the source text may be kept in several files containing translation units. Communication among the functions of a program may be carried out both through calls and through manipulation of external data.

In our language the only one scope to consider is the lexical scope of an identifier which is the region of the program text within which the identifier's characteristics are understood;

Identifiers fall into several name spaces that do not interfere with one another; the same identifier may be used for different purposes, even in the same scope, if the uses are in different name spaces.

The scope of a parameter of a function definition begins at the start of the block defining the function and persists through the function; the scope of a parameter in a function declaration ends at the end of the declarator. The scope of an identifier declared at the head of a block begins at the end of its declarator, and persists to the end of the block.

The scope of a structure, union, or enumeration tag, or an enumeration constant, begins at its appearance in a type specifier, and persists to the end of a translation unit (for declarations at the external level) or to the end of the block (for declarations within a function).

If an identifier is explicitly declared at the head of a block, including the block constituting a function, any declaration of the identifier outside the block is suspended until the end of the block.

8. Grammer

declaration:

declaration-specifiers init-declarator-list(opt);

declaration-list:
declaration
declaration-list declaration

declaration-specifiers:
storage-class-specifier declaration-specifiers(opt)
type-specifier declaration-specifiers(opt)
type-qualifier declaration-specifiers(opt)

storage-class specifier: one of
STATIC EXTERN

type specifier: one of
INT REAL
object-specifier

type-qualifier:

CONST

object-specifier:
object identifier { *object-declaration-list* }
object identifier

object:
OBJ

object-declaration-list:
object declaration
object-declaration-list object declaration

init-declarator-list:
init-declarator
init-declarator-list, init-declarator

init-declarator:
declarator
declarator = initializer

object-declaration:
specifier-qualifier-list object-declarator-list;

specifier-qualifier-list:
type-specifier specifier-qualifier-list(opt)

type-qualifier specifier-qualifier-list(opt)

object-declarator-list:
object-declarator
object-declarator-list , *objectdeclarator*

object-declarator:
declarator
declarator(opt) : *constant-expression*

declarator:
direct-declarator

direct-declarator:
identifier
(declarator)
direct-declarator [*constant-expression(opt)*]

direct-declarator (*parameter-type-list*)

type-qualifier-list:
type-qualifier

parameter-type-list:
parameter-list

parameter-list:
parameter-declaration
parameter-list , *parameter-declaration*

parameter-declaration:
declaration-specifiers declarator \

identifier-list:
identifier
identifier-list , *identifier*

initializer:
assignment-expression
{ *initializer-list* }

initializer-list:
initializer
initializer-list , *initializer*

type-name:
specifier-qualifier-list

statement:
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
draw-statement

draw-statement:
FD expression;
BK expression;
LF expression;
RT expression;
RESET;
CLS;
PF;
PD;
PU;

expression-statement:
expression(opt);

compound-statement:
{ *declaration-list(opt) statement-list(opt)* }

statement-list:
statement
statement-list statement

selection-statement:

CONDITION (expression) *statement ENDC*
CONDITION (expression) *statement* ELSE *statement ENDC*
iteration-statement:
LOOP(expression(opt)*) statement*

jump-statement:
GOON;
BRK;
RETURN *expression(opt);*

expression:
assignment-expression
expression , *assignment-expression*

assignment-expression:
conditional-expression
unary-expression assignment-operator assignment-expression

assignment-operator:
=

conditional-expression:
logical-OR-expression

constant-expression:
conditional-expression

logical-OR-expression:
logical-AND-expression
logical-OR-expression OR *logical-AND-expression*

logical-AND-expression:
inclusive-OR-expression
logical-AND-expression AND *inclusive-OR-expression*

equality-expression:
relational-expression
equality-expression == *relational-expression*
equality-expression != *relational-expression*

relational-expression:
shift-expression
relational-expression < *shift-expression*
relational-expression > *shift-expression*
relational-expression <= *shift-expression*
relational-expression >= *shift-expression*

shift-expression:
additive-expression

additive-expression:
multiplicative-expression
additive-expression + *multiplicative-expression*
additive-expression - *multiplicative-expression*

multiplicative-expression:
multiplicative-expression * *cast-expression*
multiplicative-expression / *cast-expression*
multiplicative-expression % *cast-expression*

cast-expression:
unary expression

(type-name) *cast-expression*

unary-expression:
postfix expression
unary-operator cast-expression

unary operator:
- !

postfix-expression:
primary-expression
postfix-expression[expression] postfix-expression(argument-expression-list(opt))

postfix-expression.identifier

primary-expression:
identifier
constant
(expression)

argument-expression-list:
assignment-expression
assignment-expression-list , *assignment-expression*

constant:
integer-constant
real-constant