CompArt Reference Manual

Aaron Priven, Julia Reichel, Asher Willner, Evan Zauderer

1) Introduction

   a) CompArt is a computer language that allows users to create beautiful digital canvases in an easy and efficient manner. The language is built on top of the Simple DirectMedia Layer (SDL2) Library and uses SDL's features in order to create a more friendly and comprehensible language.

2) Lexical Conventions

   a) There are five kinds of tokens described below: identifiers, keywords, constants, expression operators, and other separators. In general blanks, tabs, newlines, and comments are meant to separate tokens from one another.

   b) Comments

      (i) Inline comments are delineated by // and multi-line comments are introduced with the sequence of characters $/*$ and terminated with the characters $*/$.

   c) Identifiers

      (i) An identifier is a sequence of letters, digits, and the underscore symbol. The first character of an identifier must be alphabetic and all letters and digits must be drawn from the Unicode character set. Upper and lower case letters are considered different and two identifiers are the same only if they have the same Unicode character for each letter or digit.

   d) Keywords

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

         (1) int
         (2) struct
         (3) function
         (4) while
         (5) for
         (6) return
         (7) if
         (8) else
         (9) new

3) Data Types

a) CompArt supports four fundamental types of objects: integers, colors, canvas, and arrays.

b) Integers

   (i) An integer is a sequence of digits.

c) Color - built-in options

   (i) Black //Default Color for all Objects
   (ii) White
   (iii) Red
   (iv) Lime
   (v) Blue
   (vi) Yellow
   (vii) Aqua
   (viii) Magenta
   (ix) Silver
   (x) Gray
   (xi) Maroon
   (xii) Olive
   (xiii) Green
   (xiv) Purple
   (xv) Teal
   (xvi) Navy

d) Canvas

   (i) Surface that all shapes are drawn to.

e) Array

   (i) List of elements that can be of any data type.

4) Expressions

a) The precedence of expression operators is that of highest precedence first. Within each subsection, the operators have the same precedence. Whether an operator has left or right associativity is specified in each subsection. If unspecified, the order of evaluation of expressions is undefined. This means that the compiler can freely choose to compute subexpressions in whatever order it believes is most efficient and fitting.

b) Primary Expressions. All primary expressions group left to right.

   (i) Identifier

      (1) An identifier is a primary expression whose type is specified by its declaration.

   (ii) (expression)

      (1) 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 the primary expression.

   (iii) primary-expression (expression-list)

      (1) A function call is a primary expression followed by parentheses containing a possibly empty or comma-separated list of expressions that form the actual arguments to the function. The primary expression must be of type "function returning . . .", and the result of the function call is that same type. Recursive calls to any function are permissible.

   (iv) primary-value.member-of-structure

      (1) A value 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 value is assumed to have the same form as the structure containing the structure member.

   (v) primary-value[expression]

      (1) A subscripting expression contains a primary-value followed by a bracketed expression, which indicates the index of the desired element in the identified array.

c) Unary operators

   (i) Expressions with unary operators group right-to-left.

   (ii) − expression

      (1) The result is the negative of the expression, and has the same type. The type of the expression must be int.

   (iii) ! expression

      (1) The result of the logical negation operator ! is 1 if the value of the expression is 0, 0 if the value of the expression is non-zero. The type of the result is int. This operator is applicable to ints and expressions.

   (iv) Multiplicative operators

      (1) The multiplicative operators ∗ , / group left-to-right.

   (v) expression ∗ expression

    (1) The binary $*$ operator indicates multiplication. If both operands are int, the result is int. No other combinations are allowed.

  (vi) expression / expression

    (1) The binary / operator indicates integer division. The same type considerations as for multiplication apply.

  (vii) Additive operators

    (1) The additive operators $+$ and $-$ group left-to-right.

  (viii) expression $+$ expression

    (1) The result is the sum of the expressions. If both operands are int, the result is int. No other type combinations are allowed.

  (ix) expression $-$ expression

    (1) The result is the difference of the operands. If both operands are int, the same type considerations as for $+$ apply.

d) Relational operators

  (i) The relational operators group left-to-right, but this fact is not very useful; "a<b<c" does not mean what it seems to.

  (ii) expression $<$ expression

    (1) The operators $<$ (less than) yields 0 if the specified relation is false and 1 if it is true. Operand type requirements are exactly the same as for the $+$ operator.

  (iii) expression $==$ expression

    (1) The $==$ (equal to) operator is exactly analogous to the relational operators except for their lower precedence. (Thus "a<b == c<d" is 1 whenever a<b and c<d have the same truth-value).

  (iv) expression & & expression

    (1) The & & operator returns 1 if both its operands are non-zero, 0 otherwise. & & guarantees left-to-right evaluation; moreover the second operand is not evaluated if the first operand is 0.

  (v) expression | | expression

    (1) The | | operator returns 1 if either of its operands is non-zero, and 0 otherwise. | | guarantees left-to-right evaluation; moreover, the second operand is not evaluated if the value of the first operand is non-zero.

e) Assignment operators

  (i) identifier $=$ expression

(1) The assignment operator groups right-to-left. The value of the expression replaces that of the object referred to by the identifier. The types of the identifier and expression must be compatible.

(2) The value of the expression is simply stored into the object referred to by the identifier.

5) Statements

a) Except as indicated, statements are executed in sequence.

b) Expression statement

(i) Most statements are expression statements, which have the form:
expression ;

(ii) Usually expression statements are assignments or function calls.

c) Block statement

(i) Lists of statements can be defined in a specific block, delineated by braces { } .

d) Conditional statement

(i) The two forms of the conditional statement are

(1) if (expression) { statement }

(2) if (expression) { statement } else { statement }

(ii) In both cases the expression is evaluated and if it is non-zero, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual, the "else" ambiguity is resolved by connecting an else with the last encountered elseless if.

e) While statement

(i) The while statement has the form
while (expression) { statement }

(ii) The substatement is executed repeatedly so long as the value of the expression remains non-zero. The test takes place before each execution of the statement.

f) For statement

(i) The for statement has the form
for (expression-1opt ; expression-2opt ; expression-3opt) { statement }

(ii) This statement is equivalent to
expression-1;
while (expression-2) { statement expression-3 ;}

(iii) Thus the first expression specifies initialization for the loop; the second specifies a test, made before each iteration, such that the loop is exited when the expression becomes 0; the third expression typically specifies an incrementation which is performed after each iteration.

(iv) Any or all of the expressions may be dropped. A missing expression-2 makes the implied while clause equivalent to "while(1)"; other missing expressions are simply dropped from the expansion above.

6) Scope Rules

a) Any variable defined at the highest level of the program will be accessible throughout the program. If a variable is declared inside a function or a statement (i.e. a for-loop), it will only be accessible inside that function or statement.

7) Structs and Functions

a) Struct

(i) A struct is user defined type with certain input arguments. Users can define their own attributes for the struct.

(ii) Example:

```
struct Hoop (int x, int y) {
int hoopCenterx = x;
int hoopCentery = y;
int radius = 6;
color basketColor = Black;
}
```

b) Function

(i) A user can define a function with the following syntax:

```
function identifier ( { argument list} ) {
{ statement list }
        return expression;  // optional
}
```

8) Using Structs, Functions, and Arrays

a) Functions

(i) There is only one thing that can be done with a function: call it.

b) Arrays and subscripting

   (i) To initialize an array, one uses the notation:
   arr identifier = size;
   to instantiate an array of that size. The length of the array is the maximum number of expressions in the list. The first element of the array is considered to be at position 0. An array can contain elements of any type. To access a specific element in the array, one can subscript into the array using square brackets and listing the element's position, as such identifier[0].

   (ii) To add elements to the array, one uses the method call:
   identifier.add(element);
   and to remove an element at a specific index, one uses the method call:
   identifier.remove(index);

c) Structs

   (i) The two things you can do with a struct is instantiate a new object and pick out one of its members by means of the dot operator.

   (ii) Example:
   Hoop hoop = new Hoop(80, 10);
   hoop.radius = 7;

9) Standard Library

   a) The standard library will automatically be included so the user does not need to manually include it. It will include functions like:

      (i) drawEllipse(Canvas, int xcenter, int ycenter, int width, int height, [Color color])
      (ii) drawCircle(Canvas, int xcenter, int ycenter, int radius, [Color color])
      (iii) drawLine(Canvas, int x1, int y1, int x2, int y2, [Color color])
      (iv) drawArc(Canvas, int xcenter, int ycenter, int radius, int startDegree, int endDegree, [Color color])
      (v) drawPoint(Canvas, int x, int y, [Color color])
      (vi) drawRect(Canvas, int xcenter, int ycenter, int width, int height, [Color color])
      (vii) createWindow ([int width], [int height])
      (viii) render(Canvas)

10) Example

   a) This example is intended to illustrate some typical Compart constructions as well as a serviceable style of writing Compart programs.

```
// Example Code: Shooting a basketball into a hoop.

createWindow();

struct Ball (int x, int y) {
        int ballCenterx = x;
        int ballCentery = y;
        int radius = 5;
        color ballColor = Orange;
int middleLineX1 = x;
int middleLineX2 = x;
int middleLineY1 = y+radius;
int middleLineY2 = y-radius;
}

struct Hoop (int x, int y) {
        int hoopCenterx = x;
        int hoopCentery = y;
        int radius = 6;
        color basketColor = Black;
}

function pathBallToHoop(int startx, int starty, int goalx, int goaly) {
        arr pathx = 1000;
        arr pathy = 1000;
        // Straight line (ish) path
        int slope = (starty-goaly) / (startx-goalx);
        int b = goaly - (slope* goalx);
        int x = startx;
        int y = starty;
        while (x < goalx) {
                pathx.add(x);
                pathy.add(y);
                x = x+1;
                y = slope* x+b;
        }
        pathx.add(goalx);
        pathy.add(goaly);
        arr paths = 2;
        paths.add(pathx);
        paths.add(pathy);
        return paths;
}

Ball ball = new Ball(10, 70);
Hoop hoop = new Hoop(80, 10);

int[][] paths = pathBallToHoop(ball.ballCenterx, ball.ballCentery,
                                hoop.hoopCenterx, hoop.hoopCentery);

int[] xpath = paths[0];
int[] ypath = paths[1];

// Animate the ball going to the hoop
for (int i=0; i<xpath.length; i=i+1) {
        Canvas court = Canvas(height=100, width=100, background=Yellow);
        drawCircle(court, xpath[i], ypath[i], ball.radius, ball.ballColor);
        drawCircle(court, hoop.hoopCenterx, hoop.hoopCentery, hoop.radius, hoop.hoopColor);
        render(court);
}
```