c.def

(pronounced SEE-def)

Macromedia® Flash™ animation language

Language Reference Manual

Dennis Rakhamimov (dr524@columbia.edu), Group Leader
Eric Poirier (edp29@columbia.edu)
Charles Catanach (cnc26@columbia.edu)
Tecuan Flores (tf180@columbia.edu)

Table of Contents

1 Language Overview

The goal of **c.def**^{TM} is to create a well-organized, concise, and easy-to-use language that streamlines the creation of compound scenes and animations in Macromedia$^{®}$ Flash^{TM}.

A **c.def** program defines one or more drawing templates, called *Glyph*s, and one or more *Path*s, and then proceeds by rendering the created templates. Each of these elements can be transformed using scaling, rotation, or translation. The language supports control flow logic such as loops and conditions, and is capable of evaluating expressions using basic mathematic operators. The program author can thus conveniently use the program to create several drawing templates, and place them throughout the animation or have them move along a motion path, all with relatively little amount of code. **c.def** interpreter handles the task of creating Flash symbols, layers, and frames corresponding to the program code.

Since **c.def** enables creation of Flash animation without using the proprietary Flash GUI, it thus offers an easy and free solution to creating effective graphics. A program written in **c.def** will be interpreted to Java code, which in turn will leverage Flagstone Software's *Transform SWF* package to create a Flash SWF file. *Transform SWF* is a set of Java libraries that provide programmatic access to elements of the open SWF file format, thus creating an intermediary between low-level implementation details and the actual Flash components. The Java file can then be run to generate the actual SWF file containing the animation.

The resulting SWF file could be viewed in a Macromedia Flash viewer, or embedded into a webpage using HTML code. In the latter case, users with the Flash plug-in would be able to view the Flash animation in their favorite web browser.

2 A Tutorial Introduction

Each **c.def** program first defines a block called Document. For example,

```
Document myFlash [ &(400, 300), #Blue ]
{
   statements
}
```

The above statement had defined a Flash movie with a width of 400 pixels, height of 300 pixels, and blue as the background color.

The `Document` block can then contain declarations for `Glyph` objects, `Path`s, and control flow and iteration statements such as `for` loops.

Each `Glyph` object specifies a template consisting of drawing primitives. It can then be rendered into the Flash animation using the `Render` command, or transformed and altered using commands such as `Rotate` or `SetColor`.

Following is an example of a `Glyph` declaration that creates a template with a circle superimposed on top of a square.

```
Glyph g [ ]
{
   color[#Red]; // set the stroke color
   rect[&(-10, -10), &(20, 20) ];
   circle[&(0, 0), 50 ];
}
```

The Glyph can now be rendered into the Flash movie. It can be placed either on a particular frame, or animated over a motion guide defined by a Path. Let’s create a simple linear path.

```
Path p [ ]
{
   line[&(0, 0), &(100, 100) ];
   point[&(0, 0)];
   int[0];
}
```

The last two statements specify that the origin of the motion is the point (0, 0), and that we should start at the beginning of the path (0 percent). As an example, if we wanted to start in the middle of the path, we would use int[50].

Now we can Render the Glyph.

```
Render [ g, ->(1, 30), p ];
```

This command will then create 30 frames in the Flash movie to translate the glyph over the path.

Voila! The **cdef** source file can now be interpreted and thus converted to an SWF animation using the steps outlined in **Section 7**.

3 Lexical Conventions

3.1 Keywords

The **c.def** language consists of the following keywords. These names are reserved and thus cannot be used for identifiers.

```
break        for        Rect
circle       Glyph      Render
color        if         Rotate
continue     int        SetColor
Document     line       SetFillColor
ellipse      Path       Translate
else         point
fillcolor    polygon
```

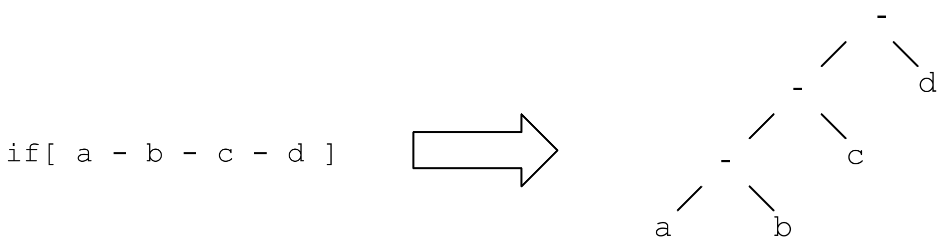
3.2 Arithmetic and Comparison Operators

c.def supports all arithmetic and comparison operators, similar to Java or C. They can be applied to the values of type `int`. Following is the table of operators, listed from highest precedence to lowest.

Operator
()
*, /, %
+, -
==, !=, >, <, >=, <=
!
&&, ||

If two operators have the same precedence are used adjacently, the operator that is parsed first has higher precedence. This means that the all of these operators are *left-associative*.

For example, the following code would internally produce the following syntax tree:



If an operation evaluates a non-zero value, it is considered to be logically true. Otherwise, it evaluates to `0` and is logically false. Thus, an arithmetic expression such as `x` or `n * (z - 5)` can be used as a logical condition.

3.3 Block Operators

Code in **c.def** is delimited by block operators. They are as follows:

```
{} /* defines the bounds of a scope */
() /* encloses the parameters for an expression */
[] /* encloses the arguments for a function call
      or an object declaration */
;  /* terminates a statement */
```

3.4 Color Identifiers

The pound (#) operator is used to identify colors. There are two ways for defining a color, which are described in **Section 4.4**.

Examples:

```
#colorname
#(r, g, b)
#Blue
#Green
#(100, 0, 0)
```

3.5 Coordinate Identifiers

The ampersand (&) operator is used to identify coordinates. Coordinates are defined by an x- and y- value, indicating their Cartesian position on the Document relative to the center of the containing object. The center thus represents the (0, 0) coordinate.

Examples:

```
&(x, y)
&(30, 70)
&(50,90)
```

3.6 Range Identifiers

The arrow (->) operator is used to identify ranges of integers. The *i* and *j* values indicate the start- and end-position of the range, respectively. The optional *k* value represents the increment that should be used when iterating over the range.

Note that if *i* < *k*, *j* should be a positive value (or can be omitted if desired). If *i* > *k*, *j* should be a negative value.

Examples:

```
->(from, to)
->(from, to, step)
->(1, 10)
->(1, 360, 30)
```

For uses of range identifiers, please see sections on `for` loops and `Render` statements.

3.7 Case Sensitivity

c.def™ is a case-sensitive language. For example, `Glyph` and `glyph` represent different identifiers.

3.8 Scoping

The language uses lexical scoping, with each level of scope being defined by curly braces. An identifier defined inside a block will go out of scope and thus be discarded when the block ends. If an identifier inside a block has the same name as an identifier in a parent block, this new identifier will temporarily hide the other one until the block ends.

3.9 Comments

c.def uses Java-style comments. There are two formats for comments:

a. Block Comments

```
/*
   This section of code is commented out.
   This section of code is commented out.
   This section of code is commented out.
*/
```

b. Single-line Comments

```
// This single line of code is commented out.
```

4 Data Types and Attributes

4.1 Fundamental Objects

Each **c.def**™ document is comprised of a single `Document` object, and a collection of `Glyph` and `Path` objects.

a. Document

The `Document` object serves as the container for `Glyph` and `Path` objects and control flow commands such as `if` and `for`.

Example:

```
Document d[&(width, height), #color]
{
   statements
   ...
}
```

Where *width* and *height* specify the size of the Flash movie, and *color* specifies the background color.

b. Glyph

A `Glyph` object is composed of a collection of primitive objects and can be rendered and moved about a `Path` object as a single entity. Since a `Glyph` can be rendered multiple times along a `Path` on a `Document`, it can be considered a form of a template.

Example:

```
Glyph g[ ]
{
   statements
   ...
}
```

The statements specified become part of the template, with the more recent statements specifying graphics that will superimpose the previous graphics. Control flow statements also can be used within `Glyph` definitions. Note that the declaration for `Glyph` currently takes no arguments.

c. Path

A Path object is the motion layer on which Glyph objects can traverse. The Path object is composed of multiple elements. The first element is an object primitive, on which the Glyph object traverses the perimeter (in the case that the object has more than 1-dimension) or outline. The second element is a reference point, in the form &(*x*, *y*), at which a Glyph and this Path object intersect. The third element is an `int` statement specifying the percentage of the perimeter that the Glyph should start at on its traversal of the Path. The object will start the path at the given percentage relative to the specified reference point, and continue until the end of path is reached (for non-looping paths) or it has reached the starting point (for looping paths).

Example:

```
Path p[ ]
{
   statements
   ...
}
```

Note that the declaration for `Path` currently takes no arguments.

4.2 Identifiers

Fundamental objects in the language are referenced by unique identifiers. Identifiers must start with a character and can contain characters, numbers, or underscores.

Examples:

```
Glyph c1 ...
Glyph myCircle ...
Path line ...
Path another_line ...
Document hello123 ...
```

4.3 Primitive Objects

Primitive objects can be used inside Glyph and Path objects to construct the particular shapes. Their coordinates are defined relative to the containing object, with (0, 0) coordinate being the center of that container. Note that the order of primitive objects matters, as the more recent objects will be rendered above the older ones.

Stroke color and fill color can be set by using the `color` and `fillcolor` commands.

a. point

A point primitive constructs a one-pixel point at the coordinate specified by the `&(x, y)` arguments.

Examples:
```
point[&(x, y)];
```

```
point[&(10, 30)];
```

b. line

A line primitive constructs a line that spans between the points that are specified by the `&(`x_1`, `y_1`)` and `&(`x_2`, `y_2`)` arguments.

Examples:

```
line[&(x1, y1), &(x2, y2)];
line[&(10, 20), &(40, 50)];
```

c. circle

A circle primitive constructs a circle with radius `rad` and is centered at the point specified by coordinate `&(x, y)`.

Examples:

```
circle[&(x, y), rad];
circle[&(0, 10), 120];
```

d. rect

The `rect` primitive constructs a rectangle with the lower-left corner at (x_1, y_1) and upper-right corner at (x_2, y_2).

Examples:

```
rect[&(x1, y1), &(x2, y2)];
rect[&(10, 10), &(100, 130)];
```

e. ellipse

The `ellipse` primitive constructs an ellipse with width `w` and height `h`, and is centered at the point specified by coordinate `&(x, y)`.

```
ellipse[&(x, y), w, h];
```

f. polygon

The `polygon` primitive constructs a polygon that is composed of n-points, with each $_i$ point specified by coordinates `&(`x_i`, `y_i`)`, and i spans from 1 to n.

```
polygon[&(x1, y1), ... , &(xn, yn)];
```

g. int

The `int` primitive represents an integer value between -32,768 and +32,767. It is currently used to specify a positional parameter in Path objects. It is also automatically defined within a `for` statement, having the name specified in the declaration of the loop.

4.4 color and fillcolor

The outline `color` and `fillcolor` can be set explicitly, and apply to all objects that are defined in the same block. If the color is changed in the middle of the block, then subsequent objects will inherit the new color. In the below example, the first rectangle and line that are defined are color #COLOR$_1$, whereas the third rectangle is of color #COLOR$_2$:

```
fillcolor[#COLOR1]
rect [&(0, 0), &(50, 20)];
line [&(0, 10), &(25, 30)];

fillcolor[#COLOR2]
rect [&(15, 20), &(100, 120)];
```

There are two ways to define colors. We have included sixteen basic colors for the programmer's convenience, but colors can also be defined using the RGB value.

a. Basic Colors

The following table identifies the 16 basic colors that can be specified as #*colorname*. The RGB column identifies the red, green, and blue composition that corresponds to each of the basic colors.

Color	Corresponding RGB Value
Aqua	(0,255,255)
Gray	(128,128,128)
Navy	(0,0,128)
Black	(0,0,0)
Green	(0,128,0)
Teal	(0,128,128)
Olive	(128,128,0)
Blue	(0,0,255)
Lime	(0,255,0)
White	(255,255,255)
Purple	(128,0,128)
Fuchsia	(255,0,255)
Maroon	(128,0,0)
Yellow	(255,255,0)
Silver	(192,192,192)
Red	(255,0,0)

Additionally, we defined a `#None` constant that is interpreted by Flash as an object with no fill.

Examples:

```
color[#Blue];
fillcolor[#Red];
fillcolor[#None];
```

b. RGB Colors

Advanced colors can be defined by their RGB value. To use a color that is not pre-defined, use the following syntax. Note that *r*, *g*, and *b* are integers between 0 and 255, specifying the RGB color composition.

```
fillcolor[#(r, g, b)];
```

Examples:

```
fillcolor[#(0, 0, 100)];
fillcolor[#(255, 0, 37)];
```

c. Color Precedence

A Glyph is composed from one or many primitive objects, each of which has a color that is specified in the definition of the Glyph. These colors can be overridden after the Glyph is created using the `SetColor` and `SetFillColor` actions. These methods override the color of each embedded primitive.

5 Action Type

5.1 Render

The Render action only applies to the Glyph foundational object. It draws the Glyph on the Document, on a specified frame. The first construct illustrates this action, which simply draws Glyph *g* onto Frame *frameNum*. The Render action also has the flexibility to draw the object on multiple frames, using the second construct below. The second construct draws Glyph *g* from *startFrame* to *endFrame*, placing glyph along Path object *p*. The range parameter can optionally be specified with a step value if it is desired to render the object with a lesser frequency.

Note that more recent Render statement will render on top of the frames rendered by previous statements. Thus, that the order in which Glyphs are rendered does matter in the resulting SWF movie.

Examples:

```
Render [g, frameNum];
Render [g, ->(startFrame, endFrame), p];
Render [g, ->(startFrame, endFrame, step), p];
Render [ferrisCar, ->(1, 360), circularPath];
```

5.2 Rotate

The Rotate action applies to both the Glyph and Path objects. The first argument for this action specifies the identifier for the Glyph or Path object to be rotated. The `degrees` argument specifies the integer number of degrees by which to rotate the foundational object. The `&(x,y)` argument specifies the x- and y- coordinates of the point around which to rotate the object.

Example:

```
Rotate [g, degrees, &(x, y)];
Rotate [p, degrees, &(x, y)];
Rotate [mySquare, 45, &(10, 10)];
```

5.3 Translate

The Translate action applies to both the Glyph and Path objects. The first argument for this action specifies the identifier for the Glyph or Path object to be rotated. The second argument specifies the x- and y- values by which to shift the object from its current position.

Example:

```
Translate [g, &(x, y)];
Translate [p, &(x, y)];
```

5.4 SetColor

The SetColor action only applies to the Glyph object. It allows for the outline colors of the Glyph (which includes all primitive objects by which the glyph is constructed) to be changed to the color specified. The first argument for this action specifies the identifier for the Glyph object to be colored. The `#color` argument specifies the color that should be inherited by all primitive objects contained within the Glyph.

Example:

```
SetColor [glyph, #color];
```

5.5 SetFillColor

The SetFillColor action only applies to the Glyph object. It allows for the fill colors of the Glyph (which includes all primitive objects by which the glyph is constructed) to be changed to the color specified. The first argument for this action specifies the identifier for the Glyph object to be filled. The `#color` argument specifies the color that should fill all primitive objects contained within the Glyph.

Example:

```
SetFillColor [glyph, #color];
```

6 Control Flow

6.1 Conditional

Logical expressions can be given using comparison operators such as == and/or specified as arithmetic expressions. An expression which evaluates to a non-zero value will be considered *true*, and one evaluating to zero will be considered *false*. The operators that are supported in a logical statement are included in **Section 3.2**. Unlike Java or C, curly braces are *required* to denote the actions to be performed.

Example:

```
if[(x + y) % z == 0)] { ... }
```

An optional else clause may be specified to denote actions that should occur if the given condition is false.

Example:

```
if[ condition ]
{
   ...
}
else
{
   ...
}
```

6.2 Iterative

An iterative statement loops over a range of integers. The `break` keyword may be used to exit a loop if a specific condition occurs.

Additionally, the `continue` keyword may be used to step out of a given iteration of the loop, and begin with the next iteration.

The specified *var* is automatically declared to be an `int` primitive and loses scope when the block ends. The second parameter specifies the range of iteration, with an optional step value. For more information on specifying a range, see **Section 3.6.**
Note that curly braces are required to denote the actions to be performed.

Example:

```
for[ var : -> ( fromValue, toValue) ]
{
   statements
   ...
}
```

7 Compilation and Execution

Programs written in **c.def** are interpreted to Java code. Executing this code produces an SWF file. The process can be performed using the following steps:

1. Write a **c.def** program.

2. Run the **c.def** interpreter to convert this program to Java code, by executing the following command:

   ```
   java CDEF filename.cdef
   ```

3. Now you will have a file called *filename*_CDEF*.java* in the current working directory. Run this file to produce the resulting SWF.

   ```
   java filename_CDEF
   ```

4. The file *filename.SWF* will now be in the current working directory. It can be viewed with the Macromedia Flash viewer or embedded in a webpage as desired.

8 Sample Code

The following sample code produces an animation of a colorful Ferris wheel with six cars.

```
/*
   c.def Language Example
   Flash animation of a Ferris Wheel
   Author: Dennis Rakhamimov
   Date:   10/24/2003
 */

/* Create a Flash animation with width=400, height=400 and white
   as the background color */
Document d[&(400, 400), #White]
{
   /* Ferris wheel base */
   Glyph base[]
   {
      line[&(0, 0), &(-100, -100)];
      line[&(0, 0), &(100, -100)];
      line[&(-100, -100), &(100, -100)];
   }

   Render [base, ->( 1, 180 )];

   /* A wheel's spike */
   Glyph spike[]
   {
      line [&(0, 0), &(0, 120)];
   }

   /* Define the wheel */
   Glyph wheel[]
   {
      for[ i: ->( 1, 6 ) ]
      {
         //Alternate colors
         if[ i + 2 == 0 ]
         {
            SetColor [spike, #Blue];
         }
         else
         {
            SetColor [spike, #Red];
         }

         Rotate [spike, 60, &(0, 0)];

         spike [];
      }
```

```
        color[#Black];
        circle[&(0, 0), 120];

        fillcolor[#Red];
        circle[&(0, 0), 20];

        fillcolor[#None];
    }

    /* Render the rotation of the wheel over 180 frames */
    for[ i : -> ( 1, 180) ]
    {
        Render [wheel, i]; /* Place wheel on frame i */
        Rotate [wheel, 2, &(0, 0)]; /* Rotate wheel by
                                        2 degrees */
    }

    /* Define a Ferris wheel car */
    Glyph ferrisCar[]
    {
        fillcolor[#Yellow];
        rect [&(0, 0), &(50, 20)];

        line [&(0, 20), &(25, 30)];
        line [&(50, 20), &(25, 30)];

        fillcolor[#Red];
        circle[&(25, 30), 3];

        fillcolor[#None];
    }

    for[ i : ->(1, 6, 1) ]
    {
        /* This is a path that ferris wheel cars will follow */
        Path circularPath[]
        {
            circle[&(0, 0), 120];
            point[&(0, 120)];
            int[i * 60];   /* start percentage */
        }

        /* Draw out the animation for this car wheel */
        Render [ferrisCar, ->(1, 180), circularPath];
    }

}
```