# Context Free in a Nutshell

## Basic Directives

***startshape***

```
startshape core
```

Indicates which rule or path is used as the starting point for the generated image. If there are multiple startshape directives then the first one is used and the rest are ignored.

***import***

```
import@font alfreebet.cfdg
import "lib/alien eyes.cfdg"
```

Causes the contents of a specified file to be inserted after the import directive. Quotes are optional for the file name; unless the file name is in a different directory, contains white space, or does not end in '.cfdg'. The contents can also be inserted into a separate namespace.

***CF::Background***

```
CF::Background = [a -1]  // transparent
```

Causes the color of the background to be other than opaque white. If there is more than one background directive then the first one is used and the rest are ignored.

***CF::Tile***

```
CF::Tile = [s 30]
CF::Tile = [s 20 30]       # rectangular
```

Enables tiled rendering and sets the size of the tiling lattice. If there is more than one tile directive then the first one is used and the rest are ignored. The tiling lattice can be square, rectangular, skewed, or rotated as long as one lattice axis is either perfectly vertical or horizontal.

```
CF::Tile = [s 30 x 15 y 10]
```

x or y shifts move the content with respect to the tiling lattice. This allows the user to center part of the design in the tile.

***CF::Size***

```
CF::Size = [s 30]
CF::Size = [s 20 30]       # rectangular
```

Explicitly sets the canvas dimensions instead of letting dynamic sizing occur. x or y shifts can be used to move the canvas around.

## Shapes

```
shape BoxedCircle {
  SQUARE []
  CIRCLE [b 1]
}
```

A rule is a list of shape replacements that describes how a particular shape can be made from other shapes.

When a rule is executed the current shape is replaced by the shapes in the list. If one of these shapes is a primitive shape (SQUARE, CIRCLE, TRIANGLE, or FILL) or a path then it is drawn immediately. All other shapes are put in a list to be drawn later.

***Randomness***

```
shape foo
rule { SQUARE[] }
rule { CIRCLE[] }
```

A shape can have more than one rule. If a shape has more than one rule then each time the shape is invoked one of the rules is chosen at random and used to draw the shape.

***Rule weights***

```
shape foo
rule { SQUARE[] }     # 1/11.01=9.1%
rule 10 { CIRCLE[] } # 10/11.01=90.8%
rule 0.01 {}          # 0.01/11.01=0.09%
```

A rule can have a rule weight following the **rule** keyword that indicates the relative probability for that rule to draw the shape.

```
shape foo
rule { SQUARE[] }      # 9.9% leftover
rule 90% { CIRCLE[] } # 90%
rule 0.1% {}           # 0.1%
```

Rule probability can be directly specified in %. Left-over probability is distributed to rules w/o %weight.

## Shape Adjustments

A shape passes on its state (geometry, color, life-time, and z position) to the shapes that replace it when a rule for the shape is executed. Each replacement can have shape adjustments that modify the state in the replacement shape:

| Adjustment | Meaning |
|---|---|
| **x** *num* | translate *num* along the x-axis |
| **x** *x y* | translate *x, y* along the x-axis & y-axis |
| **x** *x y z* | translate *x, y* along the x-axis & y-axis, and *z* along the z-axis |
| **y** *num* | translate *num* along the y-axis |
| **z** *num* | translate *num* along the z-axis† |
| **size** *num*<br>**s** *num* | scale in x and y the same amount |
| **size** *x y*<br>**s** *x y* | scale in x and y independently |
| **size** *x y z*<br>**s** *x y z* | scale in x, y, and z independently |
| **rotate** *num*<br>**r** *num* | rotate *num* degrees |
| **flip** *num*<br>**f** *num* | reflect across a line through the origin at *num* degrees |
| **skew** *y x* | shear *y* degrees from the y-axis and *x* degrees from the x-axis |
| **time** *b d* | translate birth time by *b* and death time by *d* |
| **timescale** *t* | scale temporal changes by *t* |

† The z-axis position determines which shape is on top when they overlap.

***Basic & Ordered Adjustments***

```
// move (2,5), rotate 30°, scale (2,1)
SQUARE [s 2 1 r 30 x 2 y 5]
// move (0.5,0), scale 0.95, move (0.5,0)
SQUARE [[x 0.5 s 0.95 x 0.5]]
```

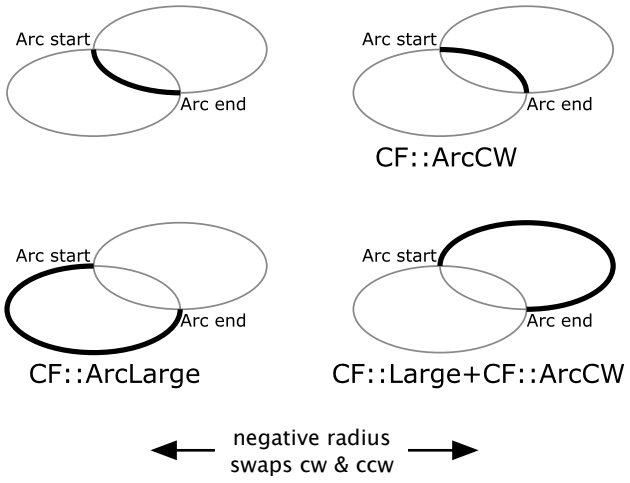
If the geometric adjustments are in double square brackets, **[[ ]]**, then they are applied in order. If the adjustments are in single square brackets, **[ ]**, then they are re-ordered to x, y, rotate, size, skew, and flip. Duplicates are dropped.

# Color Adjustments

**hue** *num* **h** *num*
add num to the hue value, wrapping from 360° to 0°
**saturation** *num* **sat** *num*
**brightness** *num* **b** *num*
**alpha** *num* **a** *num*
Range [-1,1]. If num < 0 then change the drawing saturation, brightness, or alpha num% toward 0. If num > 0 then change it num% toward 1.

***Target Color***

**hue** *num target* **h** *num target*
Range [-1,1]. If num < 0 then change the drawing hue num% toward the target hue moving clockwise around the color wheel. If num > 0 then change the drawing hue in the counterclockwise direction.
**saturation** *num target* **sat** *num target*
**brightness** *num target* **b** *num target*
**alpha** *num target* **a** *num target*
Range [-1,1]. If num < 0 then change the drawing saturation, brightness, or alpha num% toward 0 or the target value, whichever is closer. If num > 0 then change it num% toward 1 or the target value, whichever is closer.



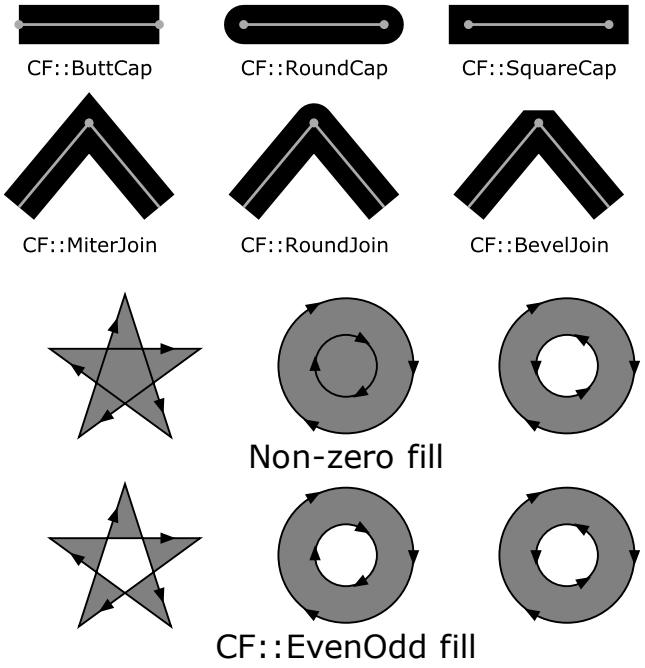

# Paths

***Path operations***

**MOVETO(***xnum*, *ynum* **)**
**LINETO(***xnum*, *ynum* **)**
**ARCTO(***xnum*, *ynum, xradius, yradius, angle{, arc flags}* **)**
**ARCTO(***xnum*, *ynum, radius{, arc flags}* **)**
**CURVETO(***xnum*, *ynum,* CF::Continuous**)**
**CURVETO(***xnum*, *ynum, xctrl1, yctrl1* **)**
**CURVETO(***xnum*, *ynum, xctrl2, yctrl2,* CF::Continuous**)**
**CURVETO(***xnum*, *ynum, xctrl1, yctrl1, xctrl2, yctrl2* )
**CLOSEPOLY() / CLOSEPOLY(CF::Align)**
All of the ***xxx*TO** path operations have ***xxx*REL** variants where the end point and any control points are relative to the beginning point.

***Path commands***

**STROKE [***adjustments* **]**
**STROKE(***stroke flags***) [***adjustments* **]**
CF::IsoWidth, CF::ButtCap, etc.
**FILL [***adjustments* ]
**FILL(***fill flags***) [***adjustments* ]
CF::EvenOdd



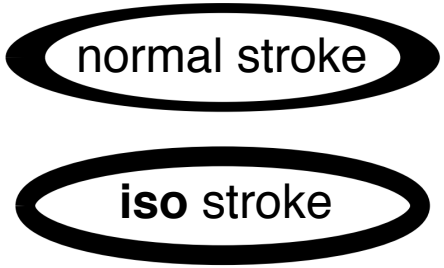

# Expressions

***Operators***

| | |
|---|---|
| **( )** | parentheses |
| **^** | exponentiation |
| **-, !** | negation, boolean not |
| ***, /** | multiplication and division |
| **+, -** | addition and subtraction |
| -- | proper subtraction |
| +-, ±, .., … | random ranges |
| <, >, <=, ≤, >=, ≥, <>, ≠ | comparison |
| &&, ll, ^^ | boolean |

***Functions***

**sin**, **cos**, **tan**, **cot**, all in degrees
**asin**, **acos**, **atan**, **acot**, returning degrees
**sinh**, **cosh**, **tanh**, **asinh**, **acosh**, **atanh**
**log**, **log10**, **sqrt**, **exp**, **abs, floor,**
**atan2(y, x)**, **mod(x, y), divides(x, y), div(x, y),**
**infinity, factorial, sg,**
**rand_static(), rand_static(x), rand_static(x, y),**
**rand(), rand(x), rand(x, y),**
**randint(), randint(x), randint(x, y),**
**select(index, choice0, choice1, …),**
**if (condition, true_exp, false_exp),**
**min(exp0, exp1, exp2, …),**
**max(exp0, exp1, exp2, …),**
**ftime, frame**



# Loop Statement

***Anonymous form***

**loop** *expression* **[***adjustments***]** *loop-body*

Where both *adjustments* can either be basic or ordered ([] or [[ ]]).

***Named form***

**loop** *var* **=** *expression* **[***adjustments***]** *loop-body*

The loop index is bound to the variable *var* inside *loop-body* (and *finally-body*).

***Finally variants***

**loop** *expression* **[***adjustments***]** *loop-body*
**finally** *finally-body*
*or*
**loop** *var* **=** *expression* **[***adjustments***]** *loop-body*
**finally** *finally-body*

Loop *expression* has one of three forms:

*count-expression*

loop counts from zero until index is ≥*count-expression*, incrementing by 1

*start-expression*, *count-expression*

loop counts from *start-expression* until index is ≥*count-expression*, incrementing by 1

*start-expression*, *count-expression, step-expression*

loop counts from *start-expression* until index is ≥*count-expression*, incrementing by *step-expression (or ≤count-expression if step-expression < 0)*

# If Statement

***if-then form***

**if (***expression***)** *then-body*

Evaluate expression and execute *then-body* if it is true (≠0).

***if-then-else form***

**if (***expression***)** *then-body* **else** *else-body*

Evaluate expression and execute *then-body* if it is true (≠0), otherwise execute *else-body*.

# Transform/Clone Statement

See online documentation

# Switch Statement

**switch (***expression***) {**
**case** *const-case-expression1***:** *case-body1*
**case** *const-case-expression2***:** *case-body2*
**case** *const-case-expression3***:** *case-body3*
**case** *const-case-expression4***:** *case-body4*
…
**else:** *else-body*
**}**

Evaluate the switch expression and compare the result to each of the constant case expressions. If there is a match then execute the corresponding *case-body*. Otherwise execute the *else-body* (if there is an *else-body*). All comparisons are done in the integer domain: all values are rounded down.

# Simple vs. Compound Body

Any body element in a loop, if, transform, or switch statement can either be a simple body or a compound body. Rule and path bodies must be compound.

***simple body***

A simple body is just a single element, where an element is a shape replacement, a path operation or command, or a loop/if/transform/switch statement.

```
loop i = 10 [x 1]
  loop i+1 [y 1]
    CIRCLE []
  finally
    SQUARE []
```

The inner loop body and finally body are simple. The outer loop body is also simple.

***compound body***

A compound body is a list of arbitrarily many elements contained in a pair of curly brackets.

```
loop i = 10 [x 1]
  loop i+1 [y 1] {
    SQUARE []
    CIRCLE [b 1]
  }
```

The inner loop body is compound and the outer loop body is simple.

# Variable Definition

A variable definition has the form

*var-name* **=** *expression*

where *expression* can a numeric expression, a shape adjustment, or a shape specification. There are global variables and local variables, depending on where the definition lies.

| Variable | Global | Local |
|---|---|---|
| definition | Between rules and paths | Inside a rule or path body or a compound body. |
| scope | To end of cfdg file | To end of enclosing body |
| evaluation | Once, at the start | Each time body is executed |

Numeric variables can be inserted into any expression wherever a number is permitted. Shape adjustment variables can be inserted into a shape adjustment using a **transform** adjustment:

```
shape tendril {
      SQUARE []
      bump = [[y 0.5 size 0.98 y 0.5 r 3]]
      tendril [transform bump]
}
```

A shape specification variable can be used in a replacement as if it were the name of a shape:

```
shape randShape {
      pickShape = select(randint(3),
            SQUARE, CIRCLE, TRIANGLE)
      pickShape []
}
```

A variable's value cannot change after it is defined. A variable with the same name can be defined if it has a smaller scope, but the original variable will be hidden, not changed. After the new variable goes out of scope the old variable will be unhidden with its original value.

```
foo = 1    // global scope
shape bar {
  foo = foo + 1  // 2nd foo, always equal to 2
  loop 5 [x 1] {
    foo = foo + 1  // 3rd foo, equal to 3
                   // for all loop iterations
    CIRCLE [y foo]
  }
  // 3rd foo out of scope, 2nd foo visible,
  // equal to 2
}
// 2nd foo out of scope, global foo visible,
// equal to 1
```

# Expressions

**Random range operators:**

`x .. y` or `x … y` returns a random number within the semi-closed range [x,y). The range operator is either formed from two ASCII period characters or one Unicode ellipsis.

`x +- y` or `x ± y` returns a random number in the semi-closed range of [x-y,x+y). The 2nd random range operator is either formed from the ASCII pair +- or by a single Unicode plus-minus symbol.

**Other Unicode symbols:**

≤ less than or equal to, equivalent to ASCII **<=**
≥ greater than or equal to, equivalent to ASCII **>=**
≠ not equal to, equivalent to ASCII **<>**
∞ infinity
π 3.1415926535

**Integer functions:**

**floor(*x*)** - rounds x to the next lower integer
**factorial(*x*)** - x!
**sg(*x*)** - returns 0 if x=0 or 1 if x≠0
**isNatural(*x*)** - returns 1 if x is a legal natural number, and integer in the interval [0,**CF::MaxNatural**]
**div(*x, y*)** - *x* ÷ *y* in the integer domain
**divides(*x, y*)** - return 1 if *y* divides *x* or 0 otherwise.

**Binary functions:**

**bitnot(*x*)** - binary inverse of x
**bitor(*x, y*)** - binary OR of x and y
**bitand(*x, y*)** - binary AND of x and y
**bitxor(*x, y*)** - binary exclusive-OR of x and y
**bitleft(*x, y*)** - binary left shift of x by y bits
**bitright(*x, y*)** - binary right shift of x by y bits

The binary functions convert their operands to 52-bit, unsigned binary numbers before performing the binary operation. Results are masked by 0xfffffffffffff

**Miscellaneous functions:**

**infinity()** - ∞
**infinity(*x*)** - ∞ if x≥0 or -∞ if x<0
**min(*x0, x1, x2, …*)** - evaluates arguments and returns the smallest one
**max(*x0, x1, x2, …*)** - evaluates arguments and returns the largest one

**Random functions:**

**rand_static()**: Static random number in the interval [0,1), uniform distribution
**rand_static(*x*)**: Static random number in the interval [0,x) (if x > 0) or [x,0) (if x < 0), uniform distribution
**rand_static(*x, y*)**: Static random number in the interval [x,y) (if y > x) or [y,x) (if x > y), uniform distribution
**rand()/rand(*x*)/rand(*x,y*)**: Random number, same intervals as rand_static(). Returns a different random number on each invocation.
**rand::normal(*mean, stddev*)**: Generates random numbers according to the Normal (or Gaussian) random number distribution.
**rand::lognormal(*mean, stddev*)**: Generates random numbers according to the Log-Normal random number distribution.
**rand::exponential(*rate*)**: Generates random non-negative floating-point values x, distributed according to and exponential probability density function.
**rand::gamma(*alpha_shape, beta_scale*)**: Generates random numbers according to the gamma random number distribution.
**rand::weibull(*alpha_shape, beta_scale*)**: Generates random numbers according to the weibull random number distribution.
**rand::extremeV(*location, scale*)**: Generates random numbers according to the extreme value random number distribution.
**rand::chisquared(*degree_freedom*)**: Generates random numbers according to the chi squared random number distribution.
**rand::cauchy(*location, scale*)**: Generates random numbers according to the Cauchy random number distribution.
**rand::fisherF(*m_degree_freedom, n_degree_freedom*)**: Generates random numbers according to Fisher's F-distribution.
**rand::studentT(*degree_freedom*)**: Generates random numbers according to Student's T-distribution
**randint()/randint(*x*)/randint(*x, y*)**: Random integer same intervals as rand_static(). Returns a different random integer on each invocation.
**randint::bernoulli(*probability*)**: Generates random boolean value with a specified probability of being true.
**randint::binomial(*trials, probability*)**: Generates random non-negative integer values i, distributed according to a binomial distribution.

**Random functions (continued):**

**randint::negbinomial(*trial_failures, probability*)**: Generates random non-negative integer values i, distributed according to a negative binomial distribution.
**randint::geometric(*probability*)**: Generates random non-negative integer values i, that represents the number of yes/no trials which are necessary to obtain a single success.
**randint::poisson(*mean*)**: Generates random non-negative integer values i, distributed according to a Poisson distribution.
**randint::discrete(*weight_0, weight_1, … , weight_n*)**: Generates random integer in the range [0,n], where each value *i* appears with a probability according to *weight_i*.

The rand_static() functions are converted into a random number when the cfdg file is compiled. So the value is constant for the entire run, but it is different for each variation and for each instance of rand_static() in the cfdg file. The other random functions return a different random value each time they are called.

**Special function:**

**select(*n, expr0, expr1, expr2, expr3*, …)** - evaluates *n* and then evaluates and returns *expr0* if *n* < 1, *expr1* if 1 ≤ *n* < 2, expr2 if 2 ≤ *n* < 3, etc. Must have at least two arguments. The *expr0, expr1, expr2,* etc. need not be numeric expressions, they can all be vectors, naturals, shape adjustments, or shape specifications. They must all be the same type.
**if(*cond, expr_true, expr_false*)** - evaluates *cond* and then evaluates and returns *expr_true* if *cond* ≠ 0 or *expr_false* if *cond* = 0. This is just syntactic sugar] for the **select()** function.
**let(*var1=expr1; var2=expr2; … ; expression*)** - evaluates each of the argument expressions, *expr1*, *expr2*, etc., and binds them to the variables *var1*, *var2*, etc. Then *expression* is evaluated n the context of the bound variables and returns this value. NB: the variable bindings are separated by semicolons, not commas. The scope of each variable is the variable bindings that follow it as well as the last expression (*expr2* can reference *var1*, *expr3* can reference *var1* and *var2*, etc.)