Fracturtles

William Bricken
May 1988, updated March 1989, resurrected August 2005, completed August 2006

Introduction

Algebra of Drawing

Fracturtles provides an *algebra of drawings* that can be used for succinct description of complex scenes, for compression of drawings, and for construction of scene templates. This implementation is a prototype with limited expressability, only two-dimensional line drawings can be constructed. Path composability, however, permits construction of extremely complex drawings from very simple descriptions.

Paths can be composed *additively* simply by joining two or more path specifications "in the same space". This results in a spatial concatenation of drawings. Paths can also be composed *multiplicatively*, every unit line of one form is expanded to the entire shape of a second form. This results in recursively complex drawings, and can be used to construct generalized template abstractions of drawings. A fractal is a path that is recursively composed with itself an infinite number of times.

Programming

I wrote this *Mathematica* program to learn *Mathematica* coding and to illustrate its integrated programming/ pattern-matching/graphics capability. It's the first large *Mathematica* program that I wrote, the code is *not* exemplary.

The code includes exploration of several programming styles -- procedural (not philosophically correct), functional/recursive (similar to LISP style), pattern-matching (a *Mathematica* strength), mathematical (use of pure functions, mapping functions, etc.), and graphics (mainly built-in to *Mathematica*).

I found *Mathematica* programming to be quite treacherous, its reliance on commas, semicolons and spaces to delineate function and argument boundaries makes debugging the syntax difficult. It's hard to use indentation and formatting as a guide to program structure. The freedom to incorporate different programming styles requires skill to use effectively. Mathematical programming is clearly the way to go, although it reqires a deep familiarity with built-in *Mathematica* functions and with a new programming style not taught in Computer Science courses. The feeling of power in the language is great, encouraging attempts to write very clean code. And there do seem to be a few unnatural quirks, especially in management of returning values from individual functions. In general, the language exposes some of the personality of its author, requiring considerable mental skill to use comfortably, and even brilliance to use correctly. With so much built-in algorithmic power, knowledge of the application of mathematical approaches to modeling is paramount. What has lagged behind, quite frankly, is the ability of the Computer Science community to understand the fundamental unification embedded in *Mathematica*.

Amazing!

The Fracturtles program was written in 1988 using a beta pre-release version of *Mathematica*. It ran immediately in *Mathematica* 5.1 in 2005, and required one trivial modification to achieve full functionality. Wolfram's 1988 prototype computational core, already extremely functional for higher mathematics, was so correct in almost every detail that it is has been necessary to change nothing at the user interface. Yes, most of the algorithms have been made more efficienct and both the functioanlity and the interface have been significantly extended, but the underlying model, that all of mathmatics can be implemented by string rewrite rules (substitution of equals for equals), has rigorously stood the test of time. Equally amazing is that language decisions made in beta required little to no revision, while the capabilities and the computational scope of *Mathematica* increased very substantively over the seventeen or so years of its refinement.

References

Abelson and diSessa,Turtle Geometry,Chapter 1

Mandelbrot,The Fractal Geometry of Nature,Section II
Peitgen and Saupe,The Science of Fractal Images,Appendix C

Fracturtles Documentation

FRACTURTLES is a version of the LOGO language extended with recursive composition (fractal in the limit) of Pen (aka turtle) paths. The Fracturtle specification language provides a simple instruction set for composing line drawings as paths,using an intrinsic coordinate system. Recursive composition of paths provides an unexplored universe of complex drawings with simple specifications.

CAUTION: There are several input formats (languages) available for the path specification code. This program is NOT input format friendly -- it will exhibit strange behavior (or crash) if a path specification does not match the specified syntax of its language. Details are included below.

Using Fracturtles

The *Mathematica* command `setUpFracturtles` (or its abbreviation, `suf`) establishes a user-interface that permits easy entry of path specifications. The system first opens an input window that requests the user to configure the output MODE and the LANGUAGE for the session. Modes and languages can be changed at any time by typing one of the key words at the Fracturtle input prompt "`InputPath: `". A user can also enter non-case-sensitive abbreviations for modes and languages as described below.

The four display *modes* are:

LINE (abbreviated as L) draws paths as lines
POINT (P) draws end-points of lines as points
DLINE (DL) draws paths (optionally input from a file) as lines, and saves output to a file
DPOINT (DP) draws paths (optionally input from a file) as end-points of lines, and saves output to a file

The two input *languages* are:

SHORT (S) allows single letter specification of Pen steps
LONG (L) requires a parameter for *every* step, including distance forward and turn angle.

There are two composition *models* used in Fracturtles, SIMPLE and MATRIX, that do not need to be specified. The only difference between the two is how path multiplication is handled.

The MATRIX algorithm is more efficient; it uses matrix composition of paths. The SIMPLE model uses a naive iterative path composition method. Specifications without a multiply command use the SIMPLE model, while specifications containing Branch or Fill commands *must* use the SIMPLE algorithm, since a branching component may be present at the finest level of detail of the composed path. The Branch command can be used to achieve interesting and complex effects including random variation.

A *path* is specified in a Command Language. Paths may be named; a path name within a command list is replaced by the path specification that is named. It is often convenient to create a separate file of *named* path specifications. Then, entering the path name at the prompt is sufficient to generate the path. Many examples of paths are included at the end of this notebook.

COMMAND options include:

forward	basic change position
rotate	basic change heading
iterate	repeat command sequence

pen up/down	activate pen
stack	begin independent command sequence
multiply	recursively embed paths
branch	choose one of two options
fill	color interior of closed polygon

ERRORS in type-in,naming,and specifications will usually generate a blank path. *Mathematica* diagnostics may or may not accompany a failure.The most common source of error is specification lists with the wrong number of nested brackets.

Typing EXIT (E) or QUIT (Q) at the prompt will exit Fracturtles and return to the *Mathematica* top-level prompt.

Pen Command Language

A COMMAND LIST is a list of Pen drawing commands and other command lists:

```
command-list::={command, command-list,...}
```

A PEN COMMAND may be any of the following:

`f[n]::=` FORWARD `n` units; negative `n` moves backward

`r[n]::=` ROTATE counterclockwise `n` degrees; negative `n` rotates clockwise

`i[n, command-list]::=` ITERATE, or repeat, the command list `n` times

`p[u|d]::=` active PEN command
  `p[u]::=` PenUp
  `p[d]::=` PenDown
    The Pen starts at coordinate `{0,0}`, facing East.
    The initial heading vector is `{1,0}`.

`s[b|e] ::=` STACK command. A stack is a separate command list that interrupts an ongoing path.
  `s[b]::=` Begin a Stack
  `s[e]::=` End a Stack

`m[command-list1, command-list2, ...]::=` MULTIPLY command, compose paths recursively

`m[list1, {list2, list3}, ...]::=` MULTIPLY command, first ADD the commands inside braces

`m[n, command-list]::=` MULTIPLY the command-list `n` times

`b[test,command-list1,command-list2]::=` BRANCH command
  If TEST is a real number between `0` and `1`, it is interpreted as a random probability, `p`.
    When a normalized random number is greater than `p`,
      do `command-list1`,
      otherwise do `command-list2`.
  If TEST is a boolean function,it is evaluated.
    When TEST is `True`,
      do `command-list1`,
      otherwise do `command-list2`.

`c[b|e] ::=` FILL (color) command. A filled polygon has a solid color (Black) filling its inside.
  `c[b]::=` Begin a filled polygon

`c[e]::=` End a filled polygon

Some of these commands (Forward, Rotate, PenUp/Down, Stack) have a short, single letter form used in a non-parametric short-form input language (examples: `f`, `r`, `F`). Step lengths and turn angles are fixed in the short-form languages. The Iterate, Branch, Multiply, and Fill commands can be used only in a parametric language that explicitly includes the Forward length and the Rotation angle for each step.

The full power of the *Mathematica* language is available within a path specification. For example,the angle of rotation can be specified by a function. This is a valid path specification within an environment that binds `theta:`

```
{f[1], r[fn[theta]], f[1]}
```

Path Composition

Paths are composed in two ways, by *addition* in space and by *multiplication* as recursive embedding.

Commands and command lists enclosed within a list are added, or concatenated together, in space:

```
step/path addition  ::= { command1, command2, command3,...}
```

Commands and command lists enclosed by a the MULTIPLY command are recursively embedded within each other:

```
path multiplication ::= { m[command-list1, command-list2,...] }
```

In path multiplication,the first path on the composition list is substituted for every unit line `f[1]` in the second path on the list, and recursively until the end of the list. Thus a global shape, such as a `square`, would appear as the last item on the composition list. The first item is always a simple unit straight line. If this base is omitted, the program will provide it automatically. The specification

```
m[path1, path2, path3]
```

means: insert `path1` in place of every forward step in `path2`, then insert the composition of `path1` and `path2` for every forward step in `path3`. If `path1` is a square for example, containing four forwards steps, and `path2` contains six forward steps, then the composition will contain 24 (4x6) forward steps, scaled to be the same size as `path1`. `Path3` is composed with the combination of `path1` and `path2`. Should `path3` contain nine forward steps, then will be 216 (4x6x9) forward steps in the composition of the three paths. Thus, multiplying several paths quickly increases the complexity of a drawing.

All arguments to MULTIPLY *must* be lists or named paths. To *add* paths within a MULTIPLY, they can be placed within another list.. In the specification below, `path1` and `path2` are added together, and then `path3` is multiplied by the sum:

`m[{path1, path2}, path3]`   ADD then MULTIPLY example

Paths are not distributive, nor are they commutative; however, joining paths in space is associative.

Recursive composition of the *same* path can be abbreviated using an integer as the first argument to the MULTIPLY command:

`m[n, command-list]::=` MULTIPLY the command-list `n` times

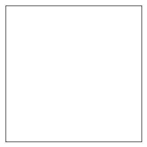
Here the MULTIPLY command contains an integer as the first argument. The command-list that follows can be complex, consisting itself of several paths. The integer is taken to mean the successive multiplication of the path(s) that follow, so:

```
m[3, path] = m[path, path, path]
```

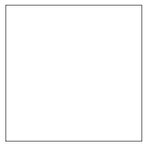
The drawing that results when `n` is taken to infinity is a FRACTAL. In practice, the deepest nesting is limited by the size of a pixel on the display screen.

Some example simple paths:

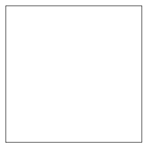
```
    square1 = { f[1], r[90], f[1], r[90], f[1], r[90], f[1], r[90] }
```



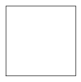
```
    square2 = { i[4, {f[1], r[90]}] }
```



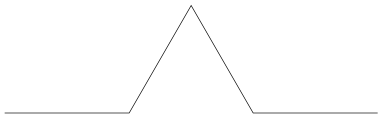
```
    square3 = { i[4, {f[1], r[-90]}] }
```



```
    square[size_] = { i[4, {f[size], r[90]}] }
```



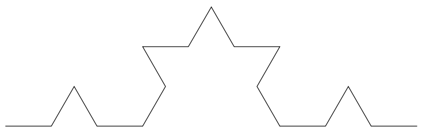
```
    koch = {f[1], r[60], f[1], r[-120], f[1], r[60], f[1]}
```



`Square1` explicitly moves the pen forward, then rotates it 90 degrees counterclockwise, then moves and rotates again, completing the Square in four repetitions. `Square2` uses the Iterate command to abbreviate the explicit movement list. The Fracturtles engine internally converts the `Square2` specification into the explicit `Square1` specification. `Square3` is similar to `Square2`, but the pen draws by travelling in a clockwise direction. `Square4` uses the parameter `size` to construct a Square with sides of any integer length.

Some example composed paths:

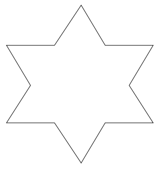
```
        { m[2, koch] }
```



draws a simple Koch path, and then embeds that path into each unit line in a copy of itself. The label `koch` is expanded in place.

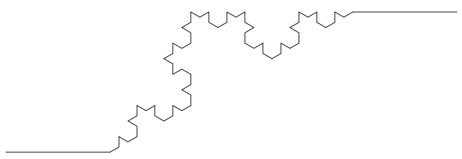
The following command draws an Equilateral Triangle with simple Koch paths as sides:

```
{ m[koch, equilateral] }
```



This command draws one forward line ten units long, rotates 30 degrees counterclockwise, draws a Koch path with three levels of recursion, rotates 30 degrees clockwise, and then draws another line ten units long:

```
{ f[10], r[30], m[3, koch], r[-30], f[10] }
```



Path Specification Languages

Fracturtles uses two different path specification languages. Each language incorporates a selection of different types of Pen COMMANDS. The reasons for different language types are largely developmental, reflecting the incremental addition of new drawing capabilities.

The interface offers two choices for languages: SHORT and LONG.

SHORT provides the capabilities of StackGraftals.

LONG provides the functionality of the Parameterize, Iterate, Multiply, Branch and Fill.
The Matrix Engine is used for Multiply commands, unless the command also contains Branch or Fill.

Each language can be arranged in a hierarchy of complexity:

Language	Included Command Types
SHORT-FORMS	
ForwardTurn	`{f, t}`
ForwardLeftRight	`{f, t, l, r, a}`
ForwardPenUpDown	`{f, t, l, r, a, F}`
StackGraftals	`{f, t, l, r, a, F, p, q, b, d}`
LONG-FORMS	
Parameterize	`{f[n], r[n], p[u|d], s[b|e]}`
Iterate	`{i[n, path]}`
Multiply	`{m[path1, path2, ...]}`
Branch	`{b[n, path1, path2]}`
Fill	`{c[b|e]}`

The first four SHORT-FORM sub-languages assume a standardized step size and turn angle. The latter five LONG-FORM sub-languages are parameterized; the step size and turn angle are specified separately for each step and eah turn.

There are two types of long-form specifications: Matrix and Simple. The difference is only internal to the Fracturtles engine MULTIPLY command. The Matrix Engine composes paths using previously composed specification sets when possible. Thus if a path is recursively embedded it itself four times, the Matrix Engine will create one path, compose it with itself once, and then compose that entire composite path specification with itself, to achieve four multiplications of the original path. The Matrix Engine is more efficient than the Simple Engine, which uses no prior computed paths while expanding a path composition.

The Simple Engine *must* be used for branching paths, because the Branch command requires a capability to probabilistically omit a small portion of a simple path at any level of nesting. Similarly, the Fill capability requires the Simple Engine, since compositions of filled polygons are generally not possible.

Short-form Languages

FORWARD TURN

ForwardTurn is the minimal Fracturtle specification language. It includes only two commands

`f`	forward one unit step
`t`	turn counterclockwise one unit angle

The turning angle is specified independently, at the beginning of a path specification, and remains the same for the entire path specification. TURN is counterclockwise to conform to the conventions of Cartesian coordinate systems. The default forward stepping distance is one unit, which is scaled by the *Mathematica* drawing package to the size of the display.

This example ForwardTurn specification draws a Square:

`{ ftftftft }` angle = 90 degrees

The Pen begins heading East. It moves one step forward and then turns 90 degrees to face North. Steping and turning continue for four steps. The final `t` command turns the Pen to face East, but does not cause a line to be drawn.

FORWARD LEFT RIGHT

ForwardLeftRight is identical to the ForwardTurn language, but with an abbreviation. The turn angle is specificed for a Left, or counterclockwise, turn as `l`. The same turn angle is also specified for a Right, or clockwise, turn as `r`.

`f`	forward one unit step
`t|l`	turn counterclockwise one unit angle
`r`	turn clockwise one unit angle
`a`	turn 180 degrees, to face in the opposite direction

The *abbreviation* is that the Right turn command, `r`, can be accomplished by a number, `n`, of consecutive Left turns, The number `n` is determined by how many times the specificed turn angle divides into 360 degrees. Similarly, the about face turn, `a`, can be achieved by a number, `n/2`, of consecutive turns.

This example ForwardLeftRight specification draws a Square:

`{ flflflfl }` angle = 90 degrees

The Square can also be drawn by turning Right, tracing the Square out in a clockwise direction:

`{ frfrfrfr }` angle = 90 degrees

A Square that is identical to the one above, with clockwise turns, can be drawn using only the `l` command, since for 90 degree turns, three Left turns has the same effect as one Right turn.

```
{ flllflllflllflll }            angle = 90 degrees
```

Compared to the ForwardTurn language for the example 90 degree turning angle, the abbreviations (also called syntactic sugar) are:

```
r = lll = t                     angle = 90 degrees

a = rr = ll = tt
```

Each of the following combinations of turns results in no change in the heading of the Pen.:

```
rrrr = llll = aa
```

FORWARD PEN UP DOWN

This language is the same as the ForwardLeftRight language with one additional capability -- the Pen can be in one of two states, Up or Down. Pen Up results in no path being drawn, but the location of the next Pen Down step will be changed:

`f`	forward one unit step with the Pen Down
`t|l`	turn counterclockwise one unit angle
`r`	turn clockwise one unit angle
`a`	turn 180 degrees, to face the opposite direction
`F`	forward one step with the Pen Up

The Pen Up capability permits specification of discontinuous paths. A Square with no top and bottom sides can be specified as:

```
{ FrfrFrfr }                    angle = 90 degrees
```

STACK GRAFTALS

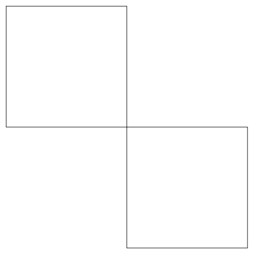
StackGraftals is the default short-form language since all other short-form languages are subsets.

StackGraftals is the same as the ForwardPenUpDown language, with one more additional capability -- separate paths can be attached to the original path using a stack. When a `p` "Begin Stack" command is encountered, the current position and heading of the Pen is stored, and a new Pen path, which is specified between the command `p` and the "End Stack" command `q` is drawn. When a `q` command is encountered, the Pen will jump to its stored location and heading to continue the original path.

`f`	forward one unit step with the Pen Down
`t|l`	turn counterclockwise one unit angle
`r`	turn clockwise one unit angle
`a`	turn 180 degrees, to face the opposite direction
`F`	forward one step with the Pen Up
`p`	begin stack (separate path) specifications
`q`	end stack (separate path) specifications
`b`	begin polygon fill (color interior)
`d`	end polygon fill (color interior)

A Square with another Square attached to its lower right corner would be specified as:

`{ frfrpafrfrfrfqfrfr }` angle = 90 degrees



In this specification, note that the stack begins after the second Right turn, "`frfrp`"; to face the Pen East it is turned 180 degrees using `a`. After the stack is completed, it is not necessary to reorient the heading of the Pen, since the specification jumps to where it left off prior to the `p` Begin Stack command.

Stacks permit a path to be inserted at any point within another path, without having to modify the specification of the second path. The `F` Pen Up command could be used to jump the Pen to any point in the drawing to insert the stack. Stacks can be composed recursively (i.e., stacks can contain other stacks).

The Fill commands is also included as an additional capability of Stackgraftals. A closed polygon can be filled with a color (black only in this implementation). The `b` command indicates that the following enclosed polygon is to be filled/colored. The `d` command indicates the end of the polygon specification.

Long-form Languages

PARAMETERIZE

The Parameterize sub-language is essentially the same language as StackGraftals, with the additional capability of specifying different step lengths and different turn angles for each command. Every simple command in a Parameterize specification is a function taking one argument.

`f[n]`	Forward `n` unit steps
`r[n]`	Rotate clockwise `n` degrees
`p[u|d]`	`p[u]` is Pen Up; `p[d]` is Pen Down
`s[b|e]`	`s[b]` begins a Stack; `s[e]`ends a Stack

In the Parametrize language, the Left command, `l`, the Right command, `r`, and the TurnAround command, `a`, are not needed, since each can be achieved as an argument to the Rotate command, `r[n]`, which turns `n` degrees counterclockwise:

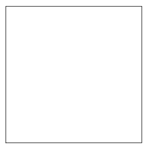
```
l = r[ang]          r = r[-ang]          a = r[180]
```

The Pen Up/Down command, `p[u|d]`, and the Stack Begin/End command, `s[b|e]`, each take one of two choices, any other argument is an error:

`p[u]`	Pen Up
`p[d]`	Pen Down
`s[b]`	Begin Stack
`s[e]`	End Stack

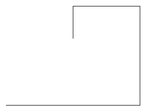
This example Parameterize specification draws a Square with side of 2 units:

```
{ f[2], r[90], f[2], r[90], f[2], r[90], f[2], r[90] }
```



This example draws a non-closed "square" with its sides spiraling inward:

```
{ f[4], r[90], f[3], r[90], f[2], r[90], f[1], r[90] }
```



ITERATE

The Iterate command adds one *abbreviation* to the Parameterize language. The *iteration* command, `i[n, path]`, allows any number of copies of a path specification to be joined together.

`f[n]`	Forward `n` unit steps
`r[n]`	Rotate clockwise `n` degrees
`p[u|d]`	`p[u]` is Pen Up; `p[d]` is Pen Down
`s[b|e]`	`s[b]` begins a Stack; `s[e]`ends a Stack
`i[n, path]`	incorporate `n` copies of the path specification, `path`

The iterate command, `i[n, path]`, is an abbreviation for `n` explicit copies of a specification, `path`. Iteration is *additive*.

This example Iterate specification draws a Square with unit sides:

```
{ i[4, {f[1], r[90]}] }
```

MULTIPLY

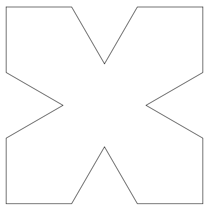
The Multiply command completes the default long-form language that uses the Matrix path-composition engine. It adds the capability of multiplying paths together via recursive embedding.

`f[n]`	Forward `n` unit steps
`r[n]`	Rotate clockwise `n` degrees
`p[u|d]`	`p[u]` is Pen Up; `p[d]` is Pen Down
`s[b|e]`	`s[b]` begins a Stack; `s[e]`ends a Stack
`i[n, path]`	incorporate `n` copies of the path specification, `path`
`m[path1, path2, ...]`	substitute `path2` for every forward step in `path1;` continue for additional specs
`m[n, path]`	recursively compose the specification with itself `n` times
`m[path1, {path2, path3}]`	add `path2` and `path3`, then substitute `path1` for each forward step

Path multiplication is the rapid way to construct complex drawings from simple specifications. The first specification forms the smallest drawn path in a composition; the last specification provides the largest, global structure of the composed path. The Multiply command can have an integer, `n` , as its first argument; in such cases, all following specifications are composed, and then the result is composed with itself `n` times. Paths can be added within a multiply specification by enclosing them within a list.

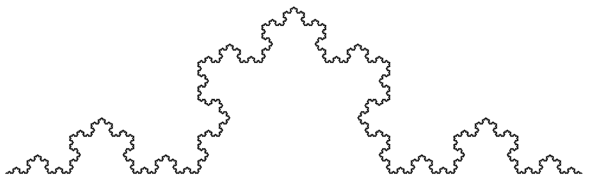
This example Multiply specification draws a Square with unit side; each side is composed of an inward facing Koch curve rather than a straight line:

```
{ m[ koch, i[4, {f[1], r[90]}] ] }
```



This example draws a complex Koch curve that approaches being a fractal. The number of actual recursive compositions is determined by the display screen resolution.

```
{ m[6, koch] }
```



The specification `{koch}` consists of 4 lines, while `{m[2, koch]}` has 16 (4x4) lines. The above specification, `{m[6, koch]}` , has 4096 (4^6) explicit lines.

BRANCH

The Branch command adds one additional, significantly different capability. Two portions of the specificed path can be given a probability between 0 and 1 of being drawn by the Pen.

`f[n]`	Forward `n` unit steps
`r[n]`	Rotate clockwise `n` degrees
`p[u|d]`	`p[u]` is Pen Up; `p[d]` is Pen Down
`s[b|e]`	`s[b]` begins a Stack; `s[e]`ends a Stack
`i[n, path]`	incorporate `n` copies of the path specification, `path`
`m[path1, path2, ...]`	substitute `path2` for every forward step in `path1;` continue for additional specs
`m[n, path]`	recursively compose the specification with itself `n` times
`b[n, path1, path2]`	with random probability, `R`: `R>n,` use `path1`; `R<n`, use `path2`
`b[v, path1, path2]`	when Boolean `v` is True`,` use `path1`; when it is False, use `path2`

When a Branch command, `b[n, path1, path2]`, is encountered, the engine selects a random probability between `0` and `1`. If the selected probability is GREATER than the specificed probability `n`, then the `spec1` path is drawn. If the selected probability is not greater than `n`, then the `path2` path is drawn.

The specified probability can be a Boolean value, `True` or `False`. This would be the same as using `.5` as the specified numerical probability. If a `Null` specification is probabilistically selected, the engine omits drawning any path for the Branch command.

The Branch command can be used to make some paths more naturalistic by including "errors" and missing components. This example Branch specification draws a unit square with a small chance (`.15`) that a given side will be omitted:

```
{ i[4, b[.15, {f[2]}, {p[u], f[2], p[d]}], r[90]}]] }
```

A unit line is drawn when the random probability `R >.15`. When `R <.15`, then second spec follows the path of the same line, but with Pen Up. The iterated Rigth Turn command, `r[90]`, is outside the Branch command, so it always occurs.

FILL

The Fill command adds an additional graphics capability -- closed polygons can be filled with a solid color. Black is the default color, since Fracturtles currently has no capabilities to specify which color. Fill is the default long-form language that uses the Simple path-composition engine.

`f[n]`	Forward `n` unit steps
`r[n]`	Rotate clockwise `n` degrees
`p[u|d]`	`p[u]` is Pen Up; `p[d]` is Pen Down
`s[b|e]`	`s[b]` begins a Stack; `s[e]`ends a Stack
`i[n, path]`	incorporate `n` copies of the path specification, `path`
`m[path1, path2, ...]`	substitute `path2` for every forward step in `path1;` continue for additional specs
`m[n, path]`	recursively compose the specification with itself `n` times
`b[n, path1, path2]`	with random probability, `R`: `R>n,` use `path1`; `R<n`, use `path2`
`b[v, path1, path2]`	when Boolean `v` is True`,` use `path1`; when it is False, use `path2`
`c[b|e]`	`c[b]` begins the specification of a closed polygon; `c[e]` ends the closed polygon path

Fill requires the specification of a closed polygon between the Color Begin command `c[b]` and the Color End command, `c[e]`. The final heading of the Pen when the polygon is closed can be any direction. Unclosed paths will not be colored. Fill requires the use of the Simple Engine, and is thus not as efficient as the Iterated language.

This example Fill specification draws a unit square and fills it with a color:

```
{ c[b], i[4, {f[1], r[90]}], c[e] }
```

Other Specification Options

NAMED PATHS

Any specification can be named, and that name can be used within another specification to place the entire named path inside the second specification.

The iterated specification for a square is:

```
{ i[4, {f[2], r[90]}] }
```

This example draws the same square:

```
path1 = { f[2], r[90] }

{ i[4, path1] }
```

FUNCTIONS AS COMMANDS

ANY *Mathematica* code can be evaluated within a path specification, so long as the result is of the right type. This example draws a square while calling a separate function to supply a specified branching probability dynamically, using the Pen heading that is in effect when the Branch is called. Each iteration generates a new, different probability:

```
{ i[4, b[getProb[getHeading[]], {f[1]}, {p[u], f[1], p[d]}], r[90]}]] }
```

When the Branch command is encountered, a function `getProb` is called, with the argument of the *current* Pen heading, `getHeading[]`. The accessor function `getHeading[]` and the probability assignment function `getProb[...]` are assumed to be written elsewhere.

Paths themselves can be functions that are evaluated when encountered by the engine. The following example names a path `pSquare`, using the name as a function that takes a direction, `dir` (counterclockwise=True, clockwise=False) as an argument. The argument is passed into the specification and used to control the Branch probability:

```
pSquare[dir_] = { b[dir, {f[2], r[90]}, {f[2], r[-90]}] }
```

The named path function can be included in any other path specification, as a substitute for the explicit named command list. From the above example, the function/path `pSquare[dir_]` , is used here as a stack that is built perpendicular (`r[90]`) to the two forward steps:

```
{ f[1], s[b], r[90], pSquare[dir_], s[e], f[1] }
```

TURN DIRECTION

Whether the Pen is travelling in a clockwise or a counterclockwise direction makes a significant difference when paths are multiplied together. Assume that the Pen is heading East and travelling counterclockwise, for example, as it draws an Equilateral Triangle.

```
equilateral1 = { f[1], r[120], f[1], r[120], f[1], r[120] }
```

The first Forward Pen step will generate a unit line heading East; the first Rotation will head the Pen up, in a Northwest direction. The specification continues until the Triangle is fully drawn.

Now consider drawing the Equilateral triangle using a clockwise rotation:

```
equilateral2 = { f[1], r[-120], f[1], r[-120], f[1], r[-120] }
```

In `equilateral2`, the first Pen step is the same as in `equilateral1`. However, the first Rotation will head the Pen *down*, in a Southwest direction. The resulting figure will be upside down compared to the first figure.

As another example, if a Stack is added somewhere along a convex polygon, a counterclockwise rotation may draw the Stack on the outside of the polygon, while a clockwise rotation will draw the Stack on the *inside* of the polygon.

Path compositions accentuate the effect of Pen rotation; multiple compositions can result in substantially different drawings depending upon rotational direction.

It is equally important to manage the final direction of the Pen heading when paths are either added or multiplied. Should the above equilateral Triangle not make the final rotational step that returns the Pen to its original heading, as in:

```
    equilateral3 = { f[1], r[120], f[1], r[120], f[1] }
```

then the next path added to the Triangle may head off in an unexpected direction (here, Southeast). When the Equilateral triangle is self-multiplied, the resulting composition will differ substantively.

NESTING COMMANDS

Three commands -- Pen, Stack, and Fill -- have Begin and End delimiters as separate commands. Other Pen, Stack, and Fill commands can occur within the scope of an initial command. Thus, the following specification is valid:

```
    stackNest = { f[1], s[b], r[90], f[1], s[b], r[60], f[1], s[e], s[e], f[1] }
```

Three commands -- Iterate, Branch, and Multiply -- have functional bracket delimiters as part of a single command. Other Iterate, Branch, and Multiply commands can be nested to any depth within these three, so that the following command is valid:

```
    iterateNest = { i[5, {f[2], r[30], i[6, {f[6], r[60]}]}] }
```

Iterate commands within Multiply commands are interpreted as path ADDITION, just as they would be outside of Multiply. For example:

```
  koch3ieq  = { m[ i[3, koch], equilateral2 ] }
```

is the same as

```
  koch3lieq = { m[ {koch, koch, koch}, equilateral2 ] }
```

The shorthand for "iterated" Multiply is a nested Multiply command. Thus,

```
  koch3meq = { m[ m[3, koch], equilateral2 ] }
```

is the same as

```
  koch3lmeq = { m[ koch, koch, koch, equilateral2 ] }
```

Nested Multiply commands accumulate complexity very quickly. The following specifications all generate the same figure that has 4096 pen strokes:

```
  koch6   = { m[koch, koch, koch, koch, koch, koch] }

  koch6s  = { m[6, koch] }

  koch3x2 = { m[2, m[3, koch]] }

  koch2x3 = { m[3, m[2, koch]] }

  koch141 = { m[koch, m[4, koch], koch] }

  koch222 = { m[m[2, m[2, koch]], m[2, koch]] }
```

2005 Extensions to the 1988 Implementation

The 1988 implemenrtaion has been extended in several ways. The most significant improvement is the ability to construct arbitrarily nested specifications.

INTERFACE

-- the interface is simpler; the program determines many parameter settings from the input itself
-- inputs are more tolerant of errors and parameter changes

PROCESSING

-- iteration and branching are expanded as a preprocess, rather than during construction of graphics
-- steerage of specifications to the two engines is smarter and more efficient

FUNCTIONALITY

-- specifications can be embedded recursively to any degree
-- additive specificatons can be included in the multiplicative space
-- all specification types can be freely intermixed

Fracturtles Code

```
(*********************  LANGUAGE DOCUMENTATION        *********************)


(*
basicLanguage { f, r, l, a, F, p, q } is the minimal subset of commands,
    all basic commands can be parameterized.
        f   --  f[1]   --  forward one unit step
        l   --  r[30]  --  turn counterclockwise one unit angle, eg 30 degrees
        t   --  r[30]  --  short language only, same as t
        r   --  r[-30] --  turn clockwise one unit angle, eg 30 degrees
        a   --  r[180] --  turn 180 degrees, to face the opposite direction
        F   --  {p[u], f[1]. p[d]}  --  forward one step with the Pen Up
        p   --  s[b]   --  begin stack (separate path) specifications
        q   --  s[e]   --  end stack (separate path) specifications
        b   --  c[b]   --  begin fill polygon
        d   --  c[e]   --  end fill polygon


Possible unexpanded string input specifications are (as examples):
        ftfftf              --  string of one-token commands
        {ftfftf}            --  list of one-token commands
        f[1]                --  single parametric command
        {f[1], r[5]}        --  list of parametric commands
        koch                --  single specname, expands to list of commands
        {koch, dash}        --  list of specnames (addition in space)
        {f[1], koch}        --  mixed list of commands and specs
        any keyword         --  restarts or aborts before processing

Parametric features add to the basic language, and include:
        stack (parametric)  --  s[b], s[e]
        iterate             --  i[_Integer, __Command]
        random branch       --  b[_Probability, path1_, path2_]
        generic branch      --  b[_BooleanTest, action1_, action2_]

        add (join in space) --  List[a, b, c]
        multiply            --  m[path1, path2, path3]
        multiply (fractal)  --  m[n, path]
        multiply (and add)  --  m[path1, {path2, path3}]
                                m[n, path1, {path2, path3}]

        fill (for art)      --  c[b] <<ClosedPolygonPath>> c[e]
*)


(********************  INTERFACE AND INITIALIZATION  ********************)


$RecursionLimit = 1024

suf := setUpFracturtles

setUpFracturtles :=
 "Initiates Fracturtles functionality.
  Opens input dialog; queries for output mode and input language."
    Block[{ filecounter = 0,    (* global *)
            mode, lang, turnang,
            tranmode, tranmodel, tranlang, tranang = 0 ,
            langs = {"S","s","short","Short","L","l","long","Long"},
            modes = {"L","l", line,"Line","P","p","point","Point",
                     "SL","sl","Sl","saveline","Saveline","SaveLine",
                     "SP","sp","Sp","savepoint","Savepoint","SavePoint"} },
        mode = InputString[
                "Output (Line, Point, SaveLine, SavePoint): "];
        exitFunction[mode];  (* aborts *)
        If[MemberQ[modes, mode],
              tranmode = determineMode[mode];
            Print["  Current Output is ", modeswitch[tranmode]],
              Print["  Output mode not recognized.  Try again."];
              setUpFracturtles;
              Abort[]];
        lang = InputString["Select Language (Short, Long): "];
        exitFunction[lang];
        If[MemberQ[langs, lang],
              tranlang = determineLanguage[lang];
            Print["  Current Language is ", langswitch[tranlang]],
              Print["  Input language not recognized.  Try again."];
              setUpFracturtles;
              Abort[]];
        If[tranlang == short,
              turnang = InputString["  Specify turning angle in degrees:  "];
              tranang = ToExpression[turnang];
              exitFunction[turnang];
              If[NumberQ[tranang],
                   Print["  Turning angle is ", tranang, " degrees."],
                     Print["  Turning angle not recognized.  Try again."];
                     setUpFracturtles;
                     Abort[]] ];
        inputLoop[tranmode, tranlang, tranang];
        Print[];
        Print["Returning to Mathematica top-level."] ]
```

```
determineLanguage[lang_] :=
    Block[{ langangs = {"S","s","short","Short"},
            langpars = {"L","l","long","Long"}    },
        Which[
            MemberQ[langangs, lang],    short,
            MemberQ[langpars, lang],    long] ]

determineMode[mode_] :=
    Block[{ modeLines = {"L","l","line","Line"},
            modePoints = {"P","p","point","Point"},
            modeDLines =
                {"SL","sl","Sl","saveline","Saveline","SaveLine"},
            modeDPoints =
                {"SP","sp","Sp","savepoint","Savepoint","SavePoint"} },
        Which[
            MemberQ[modeLines, mode],   line,
            MemberQ[modePoints, mode],  point,
            MemberQ[modeDLines, mode],  dline,
            MemberQ[modeDPoints, mode], dpoint] ]

langswitch[lang_] :=
    Switch[ lang,
        short,      "Short -- StackGraftals",
        long,       "Long -- Parameterized"]

modeswitch[mode_] :=
    Switch[ mode,
        line,       "Draw Lines",
        point,      "Draw Points",
        dline,      "Draw Lines and SaveToFile",
        dpoint,     "Draw Points and SaveToFile"]

exitFunction[in_] :=
    Block[{ exits = {"e","E","q","Q","exit","quit","Exit","Quit"} },
        If[MemberQ[exits, in],
            Print[];
            Print["Returning to Mathematica top-level."];
            Abort[] ]]

changeFunction[in_] :=
    Block[{ changes = {"S","s","short","Short","L","l","long","Long",
                       "L","l","line","Line","P","p","point","Point",
                       "DL","dl","Dl","dline","Dline","DLine",
                       "DP","dp","Dp","dpoint","Dpoint","DPoint"} },
    If[MemberQ[changes, in],
        Print["  Configuration change, restart"];
        setUpFracturtles;
        Abort[]]]

configFunction[lang_, mode_] :=
    Block[{ availmodes = {line, point, dline, dpoint},
            availlanguages = {short, long} },
        If[ Not[MemberQ[availlanguages, lang]]
            ||  Not[MemberQ[availmodes, mode]],
            Print[ "Input configuration error.  Please start again."];
            setUpFracturtles;
            Abort[] ]]
```

```
(*********************  INPUT LOOP AND PROCESS        *********************)


inputLoop[mode_, lang_, angle_] :=
 "Main Fracturtles input loop."
    Block[{ currentpathstr },
        currentpathstr = InputString["InputPath:  "];
        exitFunction[currentpathstr];
        changeFunction[currentpathstr];
        configFunction[lang, mode];
        processAndShowInput[currentpathstr, lang, mode, angle];
        inputLoop[mode, lang, angle];
        Abort[]]

(* standardize everything, all paths in parametric form *)
processAndShowInput[pathstr_, lang_, mode_, angle_] :=
 "Main Fracturtles processing loop.
  Preprocesses input specification, builds path, calls display."
    Block[{ path = ToExpression[pathstr],   (* expanded here *)
            prepath,
            points },
        If[lang === short,  (* m[] not used *)
            path = expandShort[path, angle] ];
        prepath = preprocessLong[path];
        If[FreeQ[prepath, {___, mh[___], ___}],
            points = simplePath[prepath][[1]],
            (* matrix composition for multiply only *)
            points = multiplyDispatch[prepath] ];
        showGraphics[pathstr, points, mode ] ]



(*********************  PATH PREPROCESSING            *********************)


expandShort[path_, ang_] :=
 "Changes short single letter command into parameterized command."
    Block[{ deleteBrace,
            stringcommands,
            result },
        deleteBrace = If[Head[path] === List, path[[1]], path];
        stringcommands = Characters[ToString[deleteBrace]];
        result = stringcommands /.
                 {"f" → f[1],
                  "t" → r[-ang],
                  "r" → r[-ang],
                  "l" → r[ang],
                  "a" → r[180],
                  "F" → {p[u] ,f[1], p[d]},
                  "p" → s[b],
                  "q" → s[e],
                  "b" → c[b],
                  "d" → c[e] };
        If[Length[result] > 0,
            Flatten[result],
            Print["expandShort error", result];
            Interrupt[] ]]
```

```
(* m[coms, coms] or m[n, coms, coms] as MULTIPLY *)
(* comLists in same space is ADD --> Flatten *)
(* out is {{...}, mh[...], {...}}   *)
preprocessLong[path_] :=
 "Top level for preprocessing long commands."
    Block[{ pathList,
            heads,     (* f, r, p, s, c -- i, b, m *)
            res },
        Switch[ Head[path],
            Symbol, pathList = List[path],     (* f[1] --> {f[1]} *)
            List,   pathList = Flatten[path],  (* {{a, b}, c} --> {a, b, c} *)
            _,      Print["Parsing error:  ", path];
                        Interrupt[] ];
        res = preprocessLongRecur[pathList];
            If[FreeQ[res, {___, mh[___], ___}],
                penWrap[res],                          (* {f,r,...} *)
                Map[penWrap,
                    If[Length[res] == 1,
                        res,
                        bundlize[res, { }, { }] ]]]] (* {f,r,mh[lists],...} *)


preprocessLongRecur[path_] :=
 "Recurs preprocessing over Iterate, Branch, and Multiply commands."
  Block[{ pathheads,
         res = { } },
    pathheads = Map[Head, path];
    Which[
        MemberQ[pathheads, i], res = preprocessIterate[path, { }],
        MemberQ[pathheads, b], res = preprocessBranch[path, { }],
        MemberQ[pathheads, m],
            res = removeNestedMHRecur[ bundleRecur[path, { }, { }], { }],
            (* all short language and simple recur paths *)
        Length[path] > 0,      res = path,
        True,                  Print["Parsing error in preprocessLong."];
                                  Interrupt[] ];
        res ]

(* path has m[...] *)
bundleRecur[path_, tmacc_, acc_] :=
    Which[
        path === { }  &&  tmacc === { }, acc,
        path === { }  &&  acc === { },  {tmacc},
        path === { }, Append[acc, tmacc],
        True,
            Block[{ fst = First[path],
                    rst = Rest[path] },
                If[ fst[[0]] === m,
                     fst = Apply[mh, expandM[fst]];    (* mh[] NEW HERE *)
                    bundleRecur[rst, { },
                        Join[ Join[acc, If[tmacc === { }, tmacc, {tmacc}]],
                                   {fst}]],
                    bundleRecur[rst, Join[tmacc, {fst}], acc] ]]]

(* f[1] is the default base for which paths are substituted *)
(* input is m[...] *)
(* returns {{f[1]},com, com} for m[2, com], with com expanded *)
expandM[commandM_] :=             (*  mh[] -> m[] after debug  *)
    Block[{ possint = commandM[[1]],
            recurMH, comMH, res },
        If[ IntegerQ[possint],
              recurMH = expandinner[Rest[commandM]];
            comMH = Flatten[ Table[ Apply[List, recurMH], {possint}], 1],
              recurMH = expandinner[commandM];
            comMH = recurMH];
        res = Insert[ comMH, {f[1]}, 1];
        res ]

(* inside m[] can have lists, all need to be added/flat *)
(* RECUR here, egs koch56, koch3eq *)
expandinner[coms_] :=
    Block[{ flrec, mrecur, res },
        flrec = Map[lflatten, coms];
        mrecur = preprocessLongRecur[Apply[List, flrec]];
        If[ First[mrecur][[0]] === List,
            res = Apply[mh, mrecur],
            res = Apply[mh, {mrecur}]];
        res ]

lflatten[inner_] :=
    preprocessLongRecur[ If[ inner[[0]] === List,
                            Flatten[inner],
                            {inner}]]

(* top-level mh[] has no outer List, all internals are listed.
   Recur says only one level of internal *)
(* path is mh[lists], lists may contain {mh[lists]} *)
removeNestedMHRecur[path_, acc_] :=
    If[ path === { },
        acc,
        Block[ {fst = First[path],
                fhead = First[path][[0]] },
            Which[
                fhead === mh,
                    removeNestedMHRecur[Rest[path],
                                        Join[acc, flattenMHRecur[fst, { }]]],
                fhead === List,
                    removeNestedMHRecur[Rest[path], Join[acc, fst]],
                True,
                    Print["Parsing error in removeNestedMHRecur"];
                    Interrupt[] ]]]

(* mhform is mh[lists] *)
flattenMHRecur[mhform_, mhacc_]  :=
    If[ mhform === mh[],
        List[ Apply[mh, mhacc]],
        Block[ {fst = First[mhform],
                shead = First[First[mhform]][[0]] },
            If[ fst[[0]] =!= List,
                    Print["Parsing error 1 in flattenMH"];
                       Interrupt[],
                If[ shead === mh,
                    flattenMHRecur[Rest[mhform],
                        Join[mhacc, Apply[List, Rest[First[fst]]]]],
                    flattenMHRecur[Rest[mhform], Append[mhacc, fst]]  ]]]]

penWrap[lst_] :=
    Switch[ lst[[0]],
        mh,    Map[penWrap, lst],
        List,  If[ Length[lst] == 0 || First[lst][[0]] === p,
                 lst,
                 Join[{p[d]}, lst, {p[u]}]],
        _,     Print["Parsing error in penWrap"];
                    Interrupt[] ]

bundlize[pth_, tmacc_, acc_] :=
    Which[
        pth === { }  &&  tmacc === { }, acc,
        pth === { }  &&  acc === { },  {tmacc},
        pth === { }, Append[acc, tmacc],
        True,
            Block[{ fst = First[pth],
                    rst = Rest[pth] },
                If[ fst[[0]] === mh,
                    bundlize[rst, { },
                        Join[ Join[acc, If[tmacc === { }, tmacc, {tmacc}]],
                                   {fst}]],
                    bundlize[rst, Join[tmacc, {fst}], acc] ]]]


preprocessIterate[ipath_, res_] :=
    If[ MatchQ[ipath, { }],
        preprocessLongRecur[res],
        Block[{ hd = ipath[[1, 0]],
                fst = ipath[[1]],
                rst = Rest[ipath],
                itemp, icoms  },
            If[ hd =!= i,
                preprocessIterate[rst, Append[res, fst] ],
                  itemp = fst /.  { i[n_, {co___}] -> {co},
                                    i[n_,  co___ ] -> {co} };
                  icoms = Table[itemp, {Round[fst[[1]]]}];
               preprocessIterate[rst, Join[res, Flatten[icoms]]] ]]]

preprocessBranch[bpath_, res_] :=
    If[ MatchQ[bpath, { }],
        preprocessLongRecur[res],
        Block[{ hd = bpath[[1, 0]],
                tst = bpath[[1, 1]],
                fst = bpath[[1]],
                rst = Rest[bpath],
                btemp, bcoms  },
            If[ hd =!= b,
                preprocessBranch[rst, Append[res, fst] ],
                  btemp =
                    fst /. { b[_, co1_List, co2_List] -> { co1,   co2 },
                             b[_, co1_List, co2_ ]    -> { co1,  {co2}},
                             b[_, co1_,     co2_List] -> {{co1},  co2 },
                             b[_, co1_,     co2_ ]    -> {{co1}, {co2}}  };
                  bcoms = If[If[ NumberQ[tst],
                                 Greater[Random[], tst],
                                 TrueQ[tst] ],
                             bcoms = btemp[[1]],
                             bcoms = btemp[[2]] ];
                preprocessBranch[rst, Join[res, Flatten[bcoms]]] ]]]
```

```
(********************  MATRIX PATH RECURSION        ********************)


(* needs an optimizing compiler, flatten/remove eg: {..., p[u], p[d],...} *)
(* comlsts are {{...}, mh[...], {...} ...}   *)
(* has at least one multiply, mh[] *)
(* output is {{pt, pt,...},...} *)
multiplyDispatch[comlsts_] :=
 "Multiplies paths by converting to homogeneous coordinates."
    Block[{ ptbundles },
        ptbundles = multDispLoop[comlsts, { }];
        addBundles[ptbundles] ]

(* list of points is add, mh of points is multiply *)
multDispLoop[comlsts_, res_] :=
    Block[{ tprores,  prores },
        If[Length[comlsts] == 0,
            Chop[res],   (* exit *)
              If[ comlsts[[1,0]] === mh,  (* strip mh[] *)
                prores = multiplyCompose[ Apply[List, comlsts[[1]]]],
                prores = simplePath[comlsts[[1]]]  ];
            multDispLoop[Rest[comlsts], Append[res, prores]] ]]

(* at least one bundle *)
addBundles[bunds_] :=
    Block[{ fstpts = bunds[[1,1]],
            fsth = bunds[[1,2]]  },
        If[ Length[bunds] == 1,
            fstpts,
            addBundsRecur[fsth, Rest[bunds], fstpts] ]]

(* >one bundle:  first      rest  *)
addBundsRecur[hd_, bunds_, res_] :=
    If[ bunds === { },
        res,
        Block[{ endpt = res[[-1,-1]],
                pts = bunds[[1,1]],
                newhd = bunds[[1,2]],
                rotationM,  rotated, rothd, shifted  },
            rotationM = {{hd[[1]], hd[[2]]},
                         {-hd[[2]], hd[[1]]}};
            rotated = pts . rotationM;
            rothd = newhd . rotationM;
            shifted = Map[Map[Plus[#, endpt]&, #]&, rotated];
            addBundsRecur[rothd, Rest[bunds], Join[res, shifted]] ]]



(********************  MULTIPLY PATH GENERATOR       ********************)


multiplyCompose[multLst_] :=
 "Converts command list to graphics point list, while multiplying paths."
    Block[{ baselength = multLst[[1, 2, 1]],   (* is {p[d],f[N],p[u]} *)
            generator, generatorbundle,
            carryheading = {1,0},   (* init is East *)
            hopath, thepath, ppath },
      If[ Length[multLst] == 1,
        simplePath[multLst[[1]] ],
          generatorbundle = simplePath[Rest[multLst][[1]]];
          carryheading = generatorbundle[[2]];
          generator = generatorbundle[[1]] / baselength;
          If[ Length[multLst] == 2,
            generatorbundle,
              hopath = Map[ Map[ Insert[#, 1, 3]&, # ]&, generator ];
              thepath = recurtle[baselength, hopath, Rest[Rest[multLst]] ];
            (* return to graphic point specifications *)
            ppath = Map[Map[Take[#, 2]&, #]&, thepath];
            mh[Chop[ppath], Chop[carryheading]]   ]]]

(* process path commands.
   In contrast to SIMPLEPATH which processes commands in sequence,
   RECURTLE applies matrix transformations to compose entire paths. *)
(* DO NOT USE matrix FOR BRANCHING or for POLYGON FILL *)
recurtle[dist_, template_, specList_]:=
    If[Length[specList] == 0,
        template,
        Block[{ commands = specList[[1]],
                pattern = template,
                penflag,
                active,
                statestack = {template},
                track = { },
                sM, tM, rM },
            sM = scaleM[1/dist, pattern[[1,1]]];
            pattern = Map[Dot[#, sM]&, pattern];
            While[commands =!= {},
                active = First[commands];
                commands = Rest[commands];
                Switch[active[[0]],
                    f,  Do[Switch[penflag,
                              pendown, track = Join[track, pattern],
                              penup, Null];
                        tM = translateM[pattern[[-1,-1]] - pattern[[1,1]]];
                        pattern = Map[Dot[#, tM]&, pattern], {active[[1]]}],

                    r,  rM = rotateM[active[[1]], pattern[[1,1]]];
                        carryheading = carryheading . headM[ active[[1]] ];
                        pattern = Map[Dot[#, rM]&, pattern],

                    p,  Switch[active[[1]],
                            u,  penflag = penup,
                            d,  penflag = pendown],

                    s,  Switch[active[[1]],
                            p,  PrependTo[statestack, pattern],
                            q,  pattern = statestack[[1]];
                                statestack = Rest[statestack]]]];
            recurtle[dist, track, Rest[specList]]]]


(*********************  SINGLE PATH GENERATOR         ********************)


(* must use this for fillpolygons *)
(* {p[d], f[1], p[u]}  not list-of-lists, no strings *)
simplePath[coms_] :=
 "Converts sequence of commands into a graphic path specification."
    Block[ { commands = coms,
             heading = {1,0} ,         (* initial direction is facing EAST *)
             track =  { {0,0} } ,      (* initial position is the origin *)
             fulltrack = { } ,
             penflag = penup,
             fillflag = fillup,
             statestack = { } ,
             fillpoly = { } ,
             uplocation = {0,0},
             active },
        While[ commands != { },
            active = First[commands] ;
            commands = Rest[commands] ;
            Switch[ active[[0]],

               f,  Switch[ penflag,
                       pendown,
                          Switch[ fillflag,
                             fillup,
                               AppendTo[track,
                                    track[[-1]] + active[[1]] heading],
                             filldown,
                               AppendTo[fillpoly,
                                    fillpoly[[-1]] + active[[1]] heading]],
                       penup,  uplocation = uplocation + active[[1]] heading ],

               r,  heading =  heading . headM[ active[[1]] ],

               p, Switch[ active[[1]],
                     u, If[ (penflag == pendown) && (Length[track] > 1),
                            AppendTo[fulltrack, track]] ;
                        penflag = penup ;
                        uplocation = track[[-1]] ,
                     d, If[ penflag == penup,
                            track = { uplocation } ] ;
                        penflag = pendown ] ,

               s,  Switch[ active[[1]],
                     b, PrependTo[statestack, { track[[-1]], heading }] ,
                     e, AppendTo[fulltrack, track] ;
                        track = { statestack[[1,1]] } ;
                        heading = statestack[[1,2]] ;
                        statestack = Rest[statestack] ],

               c,  Switch[ active[[1]],
                     b, AppendTo[fillpoly, track[[-1]] ];
                        fillflag = filldown ,
                     e, AppendTo[fulltrack, Polygon[fillpoly]] ;
                        fillpoly = { } ;
                        fillflag = fillup ]  ]] ;
      mh[Chop[fulltrack], heading]  ]    (* for readability *)



(********************  TOP LEVEL DISPLAY FUNCTIONS   ********************)


(* provide graphics specifications *)
```

```
showGraphics[pointpath_] := showGraphics["NoName", pointpath, line]

showGraphics[label_, pointpath_, mode_] :=
 "Displays graphics list in user-set mode."
    Block[{steps, gp},
        steps = Count[pointpath,
                  {(x_ /; NumberQ[x]), (y_ /; NumberQ[y])}, 2]
                  - Count[pointpath, {__}, 1];
        Print["This figure has ", steps, " pen strokes."];
        gp = graphicsPath[label, pointpath, mode];
        Show[ gp, AspectRatio → Automatic];
        Print[];
        Print["      ", label] ]

graphicsPath[label_, pointpath_, mode_] :=
    Block[{ linewidth = .0015,
            pointsize = .02,
            filename = ""},
        Switch[mode,
            line ,  Graphics[ Join[ {Thickness[linewidth]},
                                    {linemode[pointpath]} ],
                              AspectRatio -> Automatic ],
            point,  Graphics[ Join[ {PointSize[pointsize]},
                                    {pointmode[pointpath]} ],
                              AspectRatio -> Automatic ],
            dline,  filename = InputString["  Filename:  "];
                    Print[];
                    If[filename == "",
                        filename = makefilename[filecounter++]];
                    Display[ToString[filename],
                        Graphics[ Join[{Thickness[linewidth]},
                                        linemode[pointpath]],
                            AspectRatio → Automatic]],
            dpoint,  filename = InputString["  Filename:  "];
                     Print[];
                     If[filename == "",
                        filename = makefilename[filecounter++]];
                     Display[ToString[filename],
                            Graphics[pointmode[pointpath],
                            AspectRatio → Automatic]]]]

makefilename[numid_] := StringForm["out``.g", numid]

linemode[pointpath_]  := Flatten[ Map[fillterLine,  pointpath]]

pointmode[pointpath_] := Flatten[ Map[fillterPoint, pointpath, {2}] ]

fillterLine[path_] := If[MatchQ[path, _Polygon], path, Line[path]]

fillterPoint[path_] := If[MatchQ[path, _Polygon], path, Point[path]]


(*********************  MATRIX UTILITIES             *********************)


translateM[distance_] :=
    {{1,0,0},
     {0,1,0},
     {distance[[1]], distance[[2]], 1}}

rotateM[angle_, fpt_] :=
    Block[{cosa, sina},
            cosa = N[ Cos[ Mod[angle Degree, 2 Pi]]];
            sina = N[ Sin[ Mod[angle Degree, 2 Pi]]];
        {{cosa, sina, 0},
         {-sina, cosa, 0},
         {(1-cosa) fpt[[1]] + sina fpt[[2]],
          (1-cosa) fpt[[2]] - sina fpt[[1]], 1}}]

scaleM[size_, fpt_] :=
    {{size, 0, 0},
     {0, size, 0},
     {(1 - size) fpt[[1]], (1 - size) fpt[[2]], 1}}

headM[angle_] := headM[angle] =
    Block[{cosa, sina},
            cosa = N[ Cos[ Mod[angle Degree, 2 Pi]]];
            sina = N[ Sin[ Mod[angle Degree, 2 Pi]]];
      {{cosa, sina},
       {-sina, cosa}}]
```

Example Command Lists and Path Specifications

```
(*********************  EXAMPLE PATH SPECIFICATIONS   *********************)

square1 =
    { f[1], r[90], f[1], r[90], f[1], r[90], f[1], r[90] }

square2 =
    { i[4, {f[1], r[90]}] }

square3 =
    { i[4, {f[1], r[-90]}] }

square[size_] =
    { i[4, {f[size], r[90]}] }

equilateral =
    { f[1], r[-120], f[1], r[-120], f[1] }

equilateral1 =
    { i[3, {f[1], r[120]}] }

equilateral2 =
    { i[3, {f[1], r[-120]}] }

koch =
    {f[1], r[60], f[1], r[-120], f[1], r[60], f[1]}

peano =
    {f[1], r[90], f[1], r[-90], f[1], r[-90], f[1], r[-90],
     f[1], r[90], f[1], r[90], f[1], r[90], f[1], r[-90], f[1]}

cog =
    {f[1], r[90], f[1], r[-90], f[1], r[-90], f[1], r[90], f[1]}

bay =
    {f[1], r[90], f[1], r[-90], f[1], r[-90], f[1], f[1], r[90],
     f[1], r[90], f[1], r[-90], f[1]}

cloud =
    {f[2], r[90], f[1], r[-90], f[1], r[-90], f[1], r[180],
     f[1], r[90], f[1], r[-90], f[2], r[-90]}

thing =
    {f[4], r[-90], f[4], r[-90], f[2], r[-90], f[2], r[-90],
     f[4], r[-90], f[1], r[-90], f[1], r[-90], f[2], r[-90]}

thingcircle =
    {i[12, { thing, r[80], f[2] }]}

gon1[angle_] =
    { i[360/angle, {f[1], r[angle]}] }

gon2[sides_] =
    { i[sides, {f[1], r[360/sides]}] }

gon3[sides_] =
    { i[sides, {f[1], r[-360/sides]}] }

truncpoly[sides_, angle_] =
    { i[sides, {f[1], r[angle], f[1], r[2 angle]}]}

itgon[sides_] =
    { i[sides, {gon3[sides], f[1], r[360/sides]}]}

dash =
    {f[1], p[u], f[1], p[d], f[1]}

dash2 =
    {f[1], p[u], f[1], p[d], f[1], p[u], f[1], p[d]}

dash7 =
    { i[6, {f[1], p[u], f[1], p[d]}], f[1] }

island =
    {f[1], p[u], r[90], f[1], p[d], f[1], r[-90], f[1], r[-90],
     f[1], r[-90], f[1], r[90], p[u], f[1], p[d], r[90], f[2]}

box =
    {f[3], r[90], f[3], r[90], f[3], r[90], f[3], r[90], p[u],
     f[1], r[90], f[1], p[d], f[1], r[-90], f[1], r[-90],
     f[1], r[-90], f[1], p[u], r[90], f[1], r[90], p[d], f[2]}

vert =
    { f[1], s[b], r[90], f[1], s[e], f[1] }

star =
    {i[5, {f[1], r[144]}]}

star2 =
    { i[5, {f[1], r[144]}], f[1] }

ccurve1 =
    {f[1], r[90], f[1], r[-90]}

ccurve2 =
    {r[45], f[1], r[-90], f[1], r[45]}

ccurve3 =
    {r[-45], f[1], r[90], f[1], r[-45] }

gosper1 =
    {r[-60], f[1], r[60], f[1], f[1], r[120], f[1],
     r[60], f[1], r[-120], f[1], r[-60], f[1] }

gosper2 =
    {r[-60], f[1], r[60], f[1], r[120], f[1], r[-60], f[1],
     r[-120], f[1], f[1], r[-60], f[1] }

gosperwave =
    { gosper2, r[120], gosper1, r[120],
      gosper1, gosper2, gosper2, r[120], gosper2, gosper1 }

arcl[deg_] =
    { i[deg, {f[1], r[1]}] }

arcr[deg_] =
    { i[deg, {f[1], r[-1]}] }

ray[wavydeg_, bumps_] =
    { i[bumps, { arcl[wavydeg], arcr[wavydeg] }] }

carcl[deg_, grain_] =
    { i[deg/grain, {f[1], r[grain]}] }

carcr[deg_, grain_] =
    { i[deg/grain, {f[1], r[-grain]}] }

cray[wavydeg_, bumps_, grain_] =
    { i[bumps, { carcl[wavydeg, grain], carcr[wavydeg, grain] }] }

i1 =
    { f[1], r[45], i[4, f[1]], r[45], f[1] }

i2 =
    { f[1], r[45], i[4, {f[1]}], r[45], f[1] }

i3 =
    { f[1], r[45], i[4, f[1]], r[45], f[1] }

i4 =
    { i[2, f[1]], r[45], i[4, f[1]], r[45], i[6, f[1]] }

b1 =
    { b[.5, koch, bay] }

b2 =
    { b[.5, f[1], bay] }

b3 =
    { b[.5, koch, f[1]] }

b4 =
    { f[1], r[45], b[.5, f[1], f[5]], r[45], f[1] }

b5 =
    { f[1], r[45], b[.5, {f[1]}, f[5]], r[45], f[1] }

b6 =
    { f[1], r[45], b[.5, f[1], {f[5]}], r[45], f[1] }

b7 =
    { f[1], r[45], b[.5, {f[1]}, {f[5]}], r[45], f[1] }

b8 =
    { b[.5, f[1], f[5]], r[45], b[.5, f[1], f[5]], r[45], b[.5, f[1], f[5]] }

cwave[shift_, reps_] =
    Flatten[ If[ reps == 0, { },
                { b[ EvenQ[reps], ccurve2, ccurve3 ],
                  r[shift],
                  cwave[shift, reps - 1] } ] ]

alt[curve2_, curve3_, shift_, reps_] =
    Flatten[ If[ reps == 0, { },
                { b[ EvenQ[reps], curve2, curve3 ],
                  r[shift], alt[curve2, curve3, shift, reps - 1] } ] ]

polyrep[angle_, reps_] =
    Flatten[ If[ reps == 0, { },
                { f[1], r[angle], polyrep[angle, reps - 1] } ]]

polyspi[side_, angle_, reps_] =
    Flatten[ If[ reps == 0, { },
                { f[side], r[angle],
                  polyspi[side + 1, angle, reps - 1]}]]

polygen[side_, angle_, sideinc_, angleinc_, reps_] =
    Flatten[ If[ reps == 0, { },
                { f[side], r[angle],
                  polygen[side + sideinc,
                          angle + angleinc,
                          sideinc, angleinc, reps - 1]} ]]

spiro[side_, angle_, max_, count_, reps_] =
    Flatten[ If[ reps == 0, { },
                { f[side count], r[angle],
                  spiro[side, angle, max,
                        If[count > max, 1, count + 1], reps - 1] } ]]

rand1[reps_] =
    Flatten[ If[ reps == 0, { },
                { r[Random[Integer, {0, 360}]],
                  f[Random[Integer, {1, 10}]], rand1[reps - 1] } ]]

rand2[reps_] =
    { i[reps, {r[Random[Integer, {0, 360}]],
      f[Random[Integer, {1, 10}]] }] }

randgen[reps_, fl_, fu_, rl_, ru_] =
    Flatten[ If[ reps == 0, { },
                { r[Random[Integer, {rl, ru}]],
                  f[Random[Integer, {fl, fu}]],
                  randgen[reps - 1, fl, fu, rl, ru] } ] ]

randomkoch =
    {f[1], b[.5, {r[60], f[1], r[-120], f[1], r[60]}, {f[1]} ], f[1]}

randomdash =
    {f[1], b[.5, {f[1]}, {p[u], f[1], p[d]}], f[1]}
```

```
p1a =
    { s[b], r[22.5], f[1], s[e] }

p1b =
    { s[b], r[-22.5], f[1], s[e] }

p1 =
    { f[1], p1a, f[1], p1b, f[1] }

p2a =
    { s[b], r[22.5], f[1], r[-22.5], f[1], r[-22.5], f[1], s[e] }

p2b =
    { s[b], r[-22.5], f[1], r[22.5], f[1], r[22.5], f[1], s[e] }

p2 =
    { f[1], r[22.5], p2a, r[-22.5], p2b }

p3a =
    { }

p3b =
    { s[b], r[22.5], f[1], s[e] }

p3c =
    { }

p3d =
    { s[b], r[-22.5], f[1], s[e] }

p3 =
    { f[1], b[.8, p3a, p3b], f[1], b[.7, p3c, p3d], f[1] }

p5a[ang_] =
    { }

p5b[ang_] =
    { s[b], r[ang], f[1], r[-ang], f[1], r[-ang], f[1], s[e] }

p5c[ang_] =
    { }

p5d[ang_] =
    { s[b], r[-ang], f[1], r[ang], f[1], r[ang], f[1], s[e] }

p5[ang_] =
    { f[1], r[ang], b[.8, p5a[ang], p5b[ang]],
      r[-ang], b[.7, p5c[ang], p5d[ang]] }

ib =
    {f[10], r[45], f[5], r[-90], i[3, b[.5, koch, cog]], r[45], f[10]}

bi =
    {f[10], r[45], f[5], r[-90], b[.5, square2, equilateral1], r[45], f[10]}

ibf =
    {f[10], r[45], f[5], r[-90], i[3, b[.5, koch, dash7]], r[45], f[10]}

bif =
    {f[10], r[45], f[5], r[-90], b[.5, square2, dash7], r[45], f[10]}

ib2 =
    {f[10], r[45], i[2, b[.5, koch, cog]], r[-90],
        i[3, b[.5, koch, cog]], r[45], f[10]}

ibf2 =
    {i[2, b[.5, cog, dash7]], r[45], f[5], r[-90],
        i[3, b[.5, koch, dash7]], r[45], f[10]}

rib =
    {f[10], r[45], f[5], r[-90], i[3, b[.5, koch, i[3, b[.5, koch, cog]]]]],
        r[45], f[10]}

rbb =
    {f[10], r[45], f[5], r[-90],
        b[.5, b[.5, square2, equilateral1], equilateral1], r[45], f[10]}

ribif =
    {f[10], r[45], f[5], r[-90], i[3, b[.5, koch, i[3, b[.5, koch, dash7]]]]],
        r[45], f[10]}

rbiif =
    {f[10], r[45], f[5], r[-90], b[.5, square2, i[4, dash7]], r[45], f[10]}

rib2 =
    {f[10], r[45], i[2, b[.5, koch, cog]], r[-90],
        i[3, b[.5, koch, cog]], r[45], f[10]}

ribf2 =
    {i[2, b[.5, cog, dash7]], r[45], f[5], r[-90],
        i[3, b[.5, koch, dash7]], r[45], f[10]}


(********************  EXAMPLE PATH MULTIPLICATIONS  ********************)


  koch21 =
    {m[ koch, koch ]}

  koch21a =
    {m[ {koch, koch} ]}

  koch22 =
    {m[2, koch]}

  koch23 =
    {r[45], f[1], r[45], m[koch, koch], r[90], f[10]}

  koch24 =
    {f[1], r[45], m[2, koch], r[90], f[1]}

  koch31 =
    {m[ koch, koch, koch ]}

  koch32 =
    {m[ {koch, koch, koch} ]}

  koch33 =
    {m[ koch, {koch, koch} ]}

  koch34 =
    {m[ {koch, koch}, koch ]}

  koch41 =
    {m[ koch, koch, koch, koch ]}

  koch42 =
    {m[4, koch]}

  koch53 =
    {f[12], r[45], m[koch, koch, koch], r[45], f[10]}

  koch54 =
    {f[10], r[45], f[5], m[3, koch], r[45], f[15]}

  koch55 =
    {f[16], r[45], m[2, koch, {f[1], r[90], f[1]}, koch], r[45], f[21]}

  koch56 =
    {f[31], r[45], m[2, koch32, {f[1], r[90], f[1]}, koch], r[45], f[13]}

  koch60 =
    {m[2, i[3, b[.5, koch, cog]]]}

  koch2421 =
    {f[10], r[45], m[2, koch21], r[90], f[10]}

  koch2431 =
    {f[10], r[45], m[2, koch31], r[90], f[10]}

  koch2432 =
    {f[10], r[45], m[2, koch32], r[90], f[10]}

  koch2433 =
    {f[10], r[45], m[2, koch33], r[90], f[10]}

  koch2434 =
    {f[10], r[45], m[2, koch34], r[90], f[10]}

  koch24mi =
    {f[10], r[45], m[2, i1], r[90], f[10]}

  koch24mb =
    {f[10], r[45], m[2, b4], r[90], f[10]}

  koch2433im =
    {f[1], r[45], i[4, {f[10], r[45], m[2, koch33], r[90], f[10]}], r[45], f[1]}

  koch2434ib =
    {f[1], r[45], b[.5, {f[10], r[45], m[2, koch34], r[90], f[10]},
      f[5]], r[45], f[1]}

  koch3eq =
    { m[koch, koch, koch, equilateral1] }

  koch3eqm =
    { m[ m[3, koch], equilateral1 ] }

  koch3eqi =
    { m[ i[3, koch], equilateral1 ] }

  cloud4 =
    {m[4, cloud]}                      (* mode = point *)
```

```
coggon =
  {m[ cog, gon2[30] ]}

island2 =
  {m[2, island]}

box2 =
  {m[2, box]}

box4 =
  {m[4, box]}

koch2box =
  {m[ koch, box, box ]}

koch2boxi =
  {m[ koch, i[2, box]]}

isgonboxkoch =
  {m[ island, gon2[8], box, koch ]}

kochpeabaypea =
  {m[ koch, peano, bay, peano ]}

ccurve12 =
  {m[12, ccurve1]}

cwave2 =
  { m[2, cwave[30,12] ]}

cwavetpol =
  {m[ cwave[72,12], truncpoly[5,144] ]}

cwave2a =
  { m[2, cwave[12,30] ]}

polyspi88 =
  polyspi[1, 88, 100]                (* $RecursionLimit = 1000 *)

polyspi90sq =
  {m[ polyspi[1,90,20], square1 ]}

spiro144 =
  spiro[1, 144, 5, 1, 50]

p33 =
  {m[3, p3]}

rankoch4eq =
  {m[ i[4, randomkoch ], equilateral1 ]}    (* model = simple *)

dashstar =
  {m[ dash, star ]}

dashstargon =
  {m[ dash, star2, gon2[20] ]}

dash2gon2 =
    {m[ dash2, gon2[40] ]}

  rcray =
    {m[ {f[1], r[90], f[1]}, cray[20,20,15] ]}

  koch6   = { m[koch, koch, koch, koch, koch, koch] }

  koch6s  = { m[6, koch] }

  koch3x2 = { m[2, m[3, koch]] }

  koch2x3 = { m[3, m[2, koch]] }

  koch141 = { m[koch, m[4, koch], koch] }

  koch222 = { m[m[2, m[2, koch]], m[2, koch]] }


(*********************  MORE PATH SPECIFICATIONS      *********************)


fish1 =
    {r[120], f[1], r[-165], f[1], r[60], f[1], r[-30], f[1],
     r[-15], f[1], r[-135], f[1], r[-15], f[1], r[-30], f[1],
     r[60], f[1], r[-165], f[1], r[-45]}

fish2 =
    {r[120], f[1], r[-165], f[1], r[60], f[1], r[-30], f[1],
     r[-15], f[1], r[-135], f[1], r[-15], f[1], r[-30], f[1],
     r[60], f[1], r[-165], f[1], r[-45], p[u], f[4], p[d]}

fish[ang_] =
    {r[8 ang], f[1], r[-11 ang], f[1], r[4 ang], f[1],
     r[-2 ang], f[1], r[- ang], f[1], r[-9 ang], f[1],
     r[-ang], f[1], r[-2 ang], f[1], r[4 ang], f[1],
     r[-11 ang], f[1], r[-3 ang], p[u], f[4], p[d]}

kbase[ang_] =
    { f[1], r[ang], f[1], r[ang], f[1], r[ang], f[1] }

kocha[ang_] =
    { f[1], f[1], r[ang], f[1], r[ang], f[1], r[ang],
      f[1], r[ang], f[1], r[ang], f[1], r[-ang], f[1] }

kochb[ang_] =
    { f[1], f[1], r[ang], f[1], r[ang], f[1], r[ang],
      f[1], r[ang], f[1], f[1] }

kochc[ang_] =
    { f[1], f[1], r[ang], f[1], r[-ang], f[1], r[ang],
      f[1], r[ang], f[1], f[1] }

kochc2[ang_] =
    { f[1], f[1], r[ang], f[1], r[-ang], f[1], r[ang],
      f[1], r[ang], f[1], f[1], r[-2 * ang] }

kochd[ang_] =
    { f[1], f[1], r[ang], f[1], r[2 * ang], f[1], r[ang], f[1] }
```

```
koche[ang_] =
    { f[1], r[ang], f[1], f[1], r[2 * ang], f[1], r[ang], f[1] }

kochf[ang_] =
    { f[1], r[ang], f[1], r[-ang], f[1], r[ang], f[1], r[ang], f[1] }

kochf2[ang_] =
    { f[1], r[ang], f[1], r[-ang], f[1], r[ang],
      f[1], r[ang], f[1], r[-2 * ang] }

kochg[ang_] =
    { f[1], f[1], r[ang], s[b], r[ang], f[1], r[-ang],
      f[1], r[-ang], f[1], s[e], r[-ang], s[b], r[-ang],
      f[1], r[ang], f[1], r[ang], f[1], s[e] }

bug5 =
    {p[u], f[2], p[d], s[b], r[120], f[1], r[60], f[1], s[e],
     s[b], r[180], f[1], r[60], f[1], s[e], s[b], r[240], f[1], r[60],
     f[1], s[e], r[60], f[1], r[-120], f[1], s[b], r[0], f[1], r[-60],
     f[1], s[e], s[b], r[60], f[1], r[-60], f[1], s[e], s[b], r[120],
     f[1], r[-60], f[1], s[e], r[-60], f[1], r[-120], f[1], r[120],
     p[u], f[3], p[d] }

bug6 =
    {p[u], f[4], p[d], s[b], r[120], f[1], r[60], f[1], s[e],
     s[b], r[180], f[1], r[60], f[1], s[e], s[b], r[240], f[1], r[60],
     f[1], s[e], r[60], f[1], r[-120], f[1], s[b], r[0], f[1], r[-60],
     f[1], s[e], s[b], r[60], f[1], r[-60], f[1], s[e], s[b], r[120],
     f[1], r[-60], f[1], s[e], r[-60], f[1], r[-120], f[1], r[-120],
     p[u], f[5], p[d] }

bug7 =
    {p[u], f[2], p[d], s[b], r[120], f[1], r[60], f[1], s[e],
     s[b], r[150], f[1], r[60], f[1], s[e], s[b], r[180], f[1], r[60],
     f[1], s[e], r[60], f[1], r[-120], f[1], s[b], r[60], f[1], r[-60],
     f[1], s[e], s[b], r[90], f[1], r[-60], f[1], s[e], s[b], r[120],
     f[1], r[-60], f[1], s[e], r[-60], f[1], r[-120], f[1], r[-120],
     p[u], f[3], p[d] }

mush9 =
    {p[u], f[2], p[d], r[80], c[b], i[10, {f[1], r[-20]}], r[-80],
     f[1], r[20], f[1], r[100], f[1], r[-20], f[1], r[-80],
     f[2], r[-120], f[1], r[40], f[1], r[80], f[1], r[20],
     f[1], c[e], r[160], p[u], f[8], p[d] }

p6a =
    { s[b], r[22.5], f[1], r[-22.5], f[1], r[-22.5],
      f[1], r[-22.5], f[1], s[e] }
p6b =
    { s[b], r[-22.5], f[1], r[22.5], f[1], r[22.5], f[1], s[e] }
p6 =
    {f[1], r[22.5], p6a, r[-45], p6b}

tree[ang_] =
    {f[1], s[b], r[2 * ang], f[1], s[e], s[b], r[-ang],
     f[1], f[1], s[e] }
tree1 =
    {f[1], s[b], r[40], f[1], s[e], s[b], r[-20], f[1], f[1], s[e] }
```

```
tree2 =
    {f[1], f[1], s[b], r[40], f[1], s[e], s[b], r[-20],
     f[1], f[1], s[e] }

p8a[ang_] =
    { s[b], r[ang], f[1], r[-ang], f[1], r[-ang],
      f[1], r[-ang], f[1], s[e] }

p8b[ang_] =
    { s[b], r[-2 * ang], f[1], r[ang], f[1], r[ang], f[1], s[e] }

p8[ang_] =
    { f[1], f[1], p8a[ang], p8b[ang] }

tree3[leaf_] =
    {f[1], s[b], r[40], f[1], leaf, s[e], s[b], r[-20],
     f[1], f[1], leaf, s[e] }

mtree3[ang_] =
    {f[2], s[b], r[ang], f[1], s[e], s[b], r[-ang], f[2], r[ang],
     f[1], s[e], f[1], s[b], r[ang], f[2], r[-ang], f[1], s[e],
     f[1], s[b], r[-ang], f[1], s[e],  f[1]}

mtree[ang_, leaf_] =
    {f[2], s[b], r[ang], f[1], leaf, s[e], s[b], r[-ang], f[2], r[ang],
     f[1], leaf, s[e], f[1], s[b], r[ang], f[2], r[-ang], f[1], leaf,
     s[e], f[1], s[b], r[-ang], f[1], leaf, s[e],  f[1], leaf }

leaf1 =
    { c[b], r[30], f[1], r[-60], f[1], r[-120], f[1], r[-60],
      f[1], r[-150], c[e], p[u], f[2], p[d] }
ped3 =
    {f[1], c[b], r[60], f[1], r[-60], r[-60], f[1], r[-60], f[1],
     r[-60], r[-60], f[1], c[e], r[-60], r[-60], p[u], f[1], p[d], f[1]}
flower =
    { i[12, {r[30], s[b], r[90], ped3, s[e] } ] }
layout[d1_,d2_,d3_,d4_,d5_,d6_,d7_,d8_,d9_] =
    {d1, s[b], p[u], r[90], f[4], r[-90], p[d], d2, s[e],
     s[b], p[u], r[90], f[6], r[-90], f[4], p[d], d3, s[e],
     s[b], p[u], r[90], f[2], r[-90], f[2], p[d], d4, s[e],
     s[b], p[u], f[6], r[90], f[2], r[-90], p[d], d5, s[e],
     s[b], p[u], r[-90], f[2], r[90], f[12], p[d], d6, s[e],
     s[b], p[u], f[18], p[d], d7, s[e],
     s[b], p[u], r[-90], f[4], r[90], f[20], p[d], d8, s[e],
     p[u], r[-90], f[8], r[90], f[16], p[d], d9 }
 ln =
    {f[1]}
 ltree =
    { p[u], f[1], r[90], p[d], s[b], f[1], s[b], r[40], f[1], s[e],
      s[b], r[-20], f[1], f[1], s[e], s[b], r[40], f[1], s[b], r[40],
      f[1], s[e], s[b], r[-20], f[1], f[1], s[e], s[e], s[b], r[-20],
      f[1], s[b], r[40], f[1], s[e], s[b], r[-20], f[1], f[1], s[e],
      f[1], s[b], r[40], f[1], s[e], s[b], r[-20], f[1], f[1], s[e],
      s[e], s[e], p[u], r[-90], f[1], p[d] }


(*********************  FUNCTIONS                    ********************)


rotate[angle_, drawing_] =
    { r[angle], drawing }

skip[n_] =
    { p[u], f[n], p[d] }

place[space_, drawing_] =
    { p[u], f[space], p[d], s[b], drawing, s[e], p[u], f[space], p[d] }

join[draw1_, draw2_] =
    Join[draw1, {p[u], f[1], p[d]}, draw2]

join2[draw1_, draw2_] =
    Join[draw1, draw2]

join3[draw1_, draw2_, draw3_] =
    Join[draw1, draw2, draw3]



(*********************  NON PARAMETRIC LANGUAGE       *********************)


xdash =
    {fFfFf}
xang =
    {ftf}
xt =
    {fplfqf}
xkbase =
    {flflflf}
xka =
    {fflflflflflfrf}
xkb =
    {fflflflflff}
xkc =
    {fflfrflflff}
xkc2 =
    {fflfrflflffrr}
xkd =
    {fflfllflf}
xke =
    {flffllflf}
xkf =
    {flfrflflf}
xkf2 =
    {flfrflflfrr}
xkg =
    {fflplfrfrfqrprflflfq}
xtree =
    {fpllfqprffq}
xped1 =
    {bflflflfd}
xped2 =
    {bflflfd}
xped3 =
    {fblfrrfrfrrfdrrFf}
xped4 =                                          (*45*)
    {fplfrfrfrfq}
xbug1 =                                          (*30*)
    {lfplllfrrrfqpllfrrrfqplfrrrfqrrfrrrrfplflllfqpllflllfqplllflllfqrrf}
```

```
xbug2 =                                         (*30*)
    {lfplllfrrrfqpllfrrrfqplfrrrfqrrfrrrrfplflllfqpllflllfqplllflllfqrrfrrrrr}
xbug3 =
    {lfplllfrrrfqpllfrrrfqplfrrrfqrrfrrrrfplflllfqpllflllfqplllflllfqrrfrrrrrFF}
xp7 =
    { ffllfrfrfrfaFlFlFlFrarrrflflfaFrFrFla }
xp7a =
    { lfrfrfrfaFlFlFlFra }
xp7b =
    { rflflfaFrFrFla }
xfish3 =                                        (*15*)
    {llllllllfrrrrrrrrrrrfllllfrrfrfrrrrrrrrrfrfrrfllllfrrrrrrrrrrrfrrrFFFF}
xmush7 =                                        (*20*)
    {llllbfrfrfrfrfrfrfrfrfrfrrrrrflflllllfrfrrrrffrrrrrrfllfllllflfd}
xmush8 =
    {llllbfrfrfrfrfrfrfrfrfrfrrrrrflflllllfrfrrrrffrrrrrrfllfllllflfdllllllll}
xmush9 =
    {FllllbfrfrfrfrfrfrfrfrfrfrrrrrflflllllfrfrrrrffrrrrrrfllfllllflfdllllllllFFFFFFF}
xsq1 =                                      (*90*)
    {bffflffflffflfffld}
xsq2 =                                      (*90*)
    {bffflffflffflfffldpflFrbflflflfdFlFF}
xsq3 =                                      (*90*)
    {ffflffflffflffflpflFrflflflfFlFF}
xtri1 =                                         (*60*)
    {fplfqf}
xmtree3 =                                       (*90*)
    {ffplfqprfflfqfplffrfqfprfqf}
```