COMS 4115
Programming Languages and Translators

IpsoFracto: Fractal Generation Language
Language Reference Manual

Leo Gertsenshteyn (lpg2006@columbia.edu)
Sarah Gilman (srg2104@ columbia.edu)
Rob Notwicz (rcn15@ columbia.edu)
Anya Robertson (alr1@cs. columbia.edu)

Introduction

IpsoFracto will be a language for defining fractals. Code will be compiled into bitmap images so a programmer can easily see and alter the fractal. Tokens in the language will include identifiers, keywords, operators and separators. Whitespace will not be considered meaningful. A token will be recognized as the longest matching sequence possible in the code.

In the beginning there was Thegroup.
- IpsoFracto revolves around groups of lines, multi-lines and polygons
- Importance of the group is its use in iteration. Particularly the ability to apply kinks to a group (thus applying them to every line in it)
- There is an implied group of everything created called Thegroup
- Programmers can also create sub-groups

Conventions

Java's standard comments will be supported in IpsoFracto. Multi-line comments will be enclosed in /* and */, while single line comments start with // and end with a new line. Single line comments may appear on the same line after code.

Code blocks, such as control statements or loops, are separated by the { } symbols. Single statements such as declarations and function calls to apply kinks to lines are ended with an exclamation point. Function arguments immediately follow a function name, are enclosed in parenthesis and separated by commas.

Keywords

IpsoFracto's reserved words are:

Word	*Type*	*Purpose*
add	function	Adds an line to a multiline or an object to a group
apply	function	Function name for applying kinks to a line
Bool	data type	Data type for boolean declarations
else	control flow	Optional portion of the if conditional statement
Float	data type	Data type for floating point number declarations
Group	data type	Data type for encapsulating other data types
if	control flow	Conditional statement
Int	data type	Data type for integer declarations
iter	control flow	Loop statement
Line	data type	Data type for line declarations
Point	data type	Data type for point declarations

Polygon	data type	Data type for polygon declarations

Keywords in IpsoFracto are case sensitive.

Predefined constants

Word	*Type*	*Description*
CENTER	imperative constant	Imperative that kink should be applied to the center of the line
LEFT	imperative constant	Imperative that kink should be applied to the left of the line
NEG	imperative constant	Imperative that kink should be applied to the negative orientation of the line
FALSE	constant	Literal for the Boolean false
POS	imperative constant	Imperative that kink should be applied to the positive orientation of the line
RIGHT	imperative constant	Imperative that kink should be applied to the right of the line
TRUE	constant	Literal for the Boolean true
KINK	flag constant	Indicator that a line segment is the result of a kink applied at the most recent iteration
KRIGHT	flag constant	Indicator that a line segment is on the right side of a kink applied at the most recent iteration
KLEFT	flag constant	Indicator that a line segment is on the left side of a kink applied at the most recent iteration
KCENTER	flag constant	Indicator that a line segment is at the center of a kink applied at the most recent iteration
NKRIGHT	flag constant	Indicator that a line segment is on a kinked line, to the right of a kink applied at the most recent iteration
NKLEFT	flag constant	Indicator that a line segment is on a kinked line, to the left of a kink applied at the most recent iteration
GROUPED	flag constant	Indicator that an object is part of an explicit group
IGNORE	flag constant	Indicator that flagged object should not participate in kinks

Data Types

Atomics

IpsoFracto has three types of literals: integers, floating point numbers and booleans. Integers are any sequence of numbers 0-9 repeated. Floating point numbers is any integer followed by a decimal point, followed by another integer. The exponent functionality available for floating points numbers in many languages is not included in IpsoFracto. (Any factors of ten that take more characters to type out than to write in e-notation would either be indiscernible or dominate the screen.) Booleans consist of the

predefined constants TRUE and FALSE. When operators are applied to different types, all literals are promoted to the more general type. Literals may be defined with the keywords Int, Float, and Bool.

Each type of literal may, additionally, exist as an atomic named variable. This is similar to the concept of a scalar variable in Perl, excepting that there are no facilities for strings in IpsoFracto.

Moleculars

There are five types that are composed of atomics in IpsoFracto: points, lines, multilines, polygons and groups. All of these types can be associated with variables, whose names follow the identifier rules as shown below, and should be

Points are defined by pair of Ints or Floats, such as [10, 20]. A point variable may be defined using the Point keyword. The basic mathematical operators (+ - * and /) can be applied to points. They are applied to each of the coordinates, so (x1, y1) + (x2, y2) equals (x1+y1, x2+y2). The unary operators (++ and --) can also be applied to points in a similar fashion, as parallels the group operating paradigm of the language. Dynamic points consist of: [# , #]

Lines are a pair of points with one tagged as the start point and the other as the end point. The direction of the line from the start to end, and the positive orientation of the line is the left side when facing in the direction of the line. A line variable may be defined with the keyword Line. The operators -> and <- define the relationship between the points and therefore the direction of the line, in the visually expected manner and provide a "sense" for the orientation flags. Dynamic lines consist of: <Point> (->|<-) <Point>

Polygons are, in essence, lists of points, with the implication of lines in a continuous single orientation connect them. Polygon variables are specified with the keyword Polygon, and must consist of at least three points. There is always an implicit final connection from the last point specified to the first point specified. Dynamic polygons consist of: <Point>, (<Point>,)+ <Point>

Groups are a way of organizing lines. Groups do not honor sub-hierarchy, so even though two polygons may be members of a group, to operations that work on or through groups, they may just as well have been individual lines with the appropriate endpoints and orientations. A group may be defined with the keyword Group. A dynamic group consists of ((<molecular label>,)* <molecular label>)!

Operators

The following operators are defined, and are in order of decreasing precedence:

Operator	*Purpose*
{ } ( )	Separators for code control blocks, ( ) also specifies points
[]	Dynamic array definition

,	Separates items in dynamic lists
.	Used to associate a function to an object
%	Mod – valid on integers
* /	Multiply and Divide – valid on integers, floating point numbers and points
+ -	Plus and Minus – valid on integers, floating point numbers and points
&& || ~	And, Or and Not – valid on Booleans
== <= >= < >	Equals, Less than or equal to, Greater than or equal to, less than, greater than – valid on Booleans
..	The yadda operator, produces items of a dynamic list
-> <-	The orientation operators
=	Assignment operator
++ --	Post-inc, Post-dec – valid on integers and floating point numbers

Identifiers and Declarations
Specific instances of all data types can be declared using the keywords mentioned, followed by an identifier, =, and a dynamic version of the appropriate data type. Identifiers are any combinations of upper and lower case letters, digits and underbars starting with a letter. The first 50 characters are significant.

Declaration Examples
Point startPoint = [10,20]!
Line singleLine = startPoint → [20, 30]!
Polygon Polly3 = startPoint → (10, 50) → (30, 30)!
Group mygroup = (myLine, myLine2)!

Scope
All variable declarations are valid with the { } separators in which they are defined. Declarations appearing at the top of a program (outside a {}) are valid throughout the entire program.

Control Flow

Conditional Statement
IpsoFracto supports the basic if statement. This includes a single conditional, an optional else and one or more statements for both conditions enclosed in { }. If using only single statements, the {} are optional.

Loops
IpsoFracto defines a looping function, iter. The keyword iter should be followed by a parenthesized block, followed by the name of a variable, followed by an open brace(required), an arbitrary number of statements, and a close brace. The parenthesized block should be of the form (#1 .. #2, #3), where each # is a numeric value. Iter will then repeat all statements inside the iter scope for values of the variable set in turn from #1 to

#2 inclusive, (evaluated once at the beginning of the loop and not dynamically updated,) in steps of #3 size. The loop variable may be first instantiated in the iter command and it will be of valid scope throughout its execution. The end point and iteration change may be left out and assumed to be 0 and 1.

Functions
There is one predefined function in IpsoFracto: apply. This is used to apply a kink to a line and is the pivotal function of the language. The function requires that the user specify the following arguments, comma separated, in parenthesis: position in the line of the kink (CENTER, LEFT or RIGHT), the orientation of the kink (POS or NEG), the kink to be applied, and the number of kinks applied "in parallel". The kinks may be defined inside the function call although for cases of repeating a single style of kink in several contexts, it is preferable to declare them outside it.

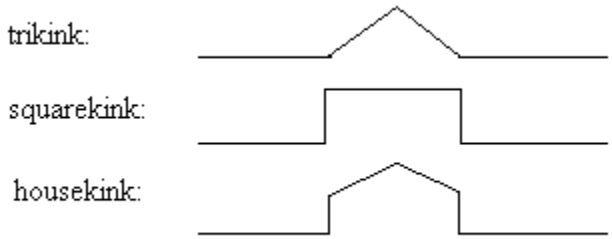
Additional keywords are specified by the libraries. Specifically, any library will have keywords for each type of kink defined in that library. The standard libraries will include routines for: trikink, squarekink, housekink.

Kink Library

The initial library for IpsoFracto will include three kinks: trikink, squarekink and housekink. These can be declared using the Kink keyword. Declarations for these kinks require two integers from 1-10 indicating the relative height and width of the kink compared to the line. The syntax is:

trikink (5, 5)

The kinks looks like:



Syntax summary

<statement> =>
All syntax options secified below are statements.

<label> =>

Any alpha-numeric string of 30 or less characters starting with a letter

<line> =>
Line <label> (<line definition predicate>)!

<line definition predicate> =>
<point definition> (->|<-) <point definition>

<instantiation> =>
<type keyword> <assignment>

<assignement> =>
<label> = <dynamic instance of appropriate type>!

<if> =>
if (<boolean statement>) {
<statement>! +
} <else>

<else> =>
\epsilon | else {<statement>! +} | else <statement>!

<iter> =>
iter (<number>..<number>, <number>) <atomic label>{
<statement>! +
}
| iter (<number>..<number>, <number>) <atomic label>
<statement>!

<apply>=>
apply (<molecular label>, <imperative constant>, <imperative constant>, <kink defined in libraries>, <atomic>)!

Sample Program

```
Line myline = [20, 0] → [50, 0]!
iter (2..0, -1) x
{
    if ( x % 2 == 0 )
    { apply( Thegroup, CENTER, POS, trikink(3,3), 2 )! }
    else
    { apply (Thegroup, LEFT, POS, squarekink(5,5), 1 )!
}
```

Output: