Point-To-Point (PTP)

Language Reference Manual

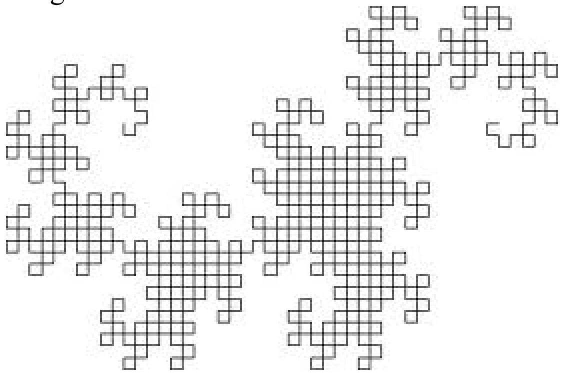
PTP - Minimalistic Programming Language for generating graphics programmatically

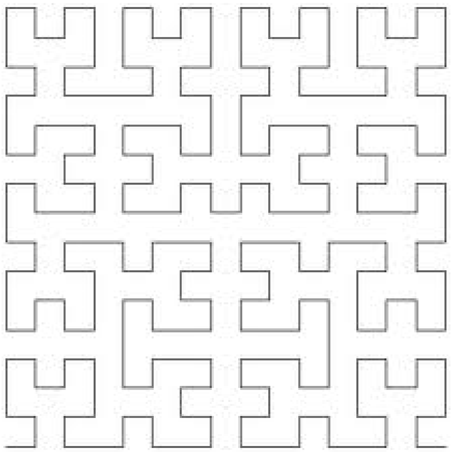
COMS 4115, Columbia University, Summer 2011
Venera Varbanova (vv2200@columbia.edu)

1. Overview of PTP

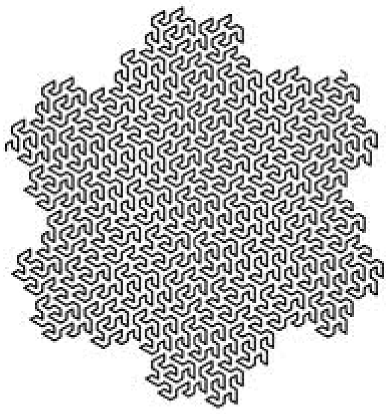
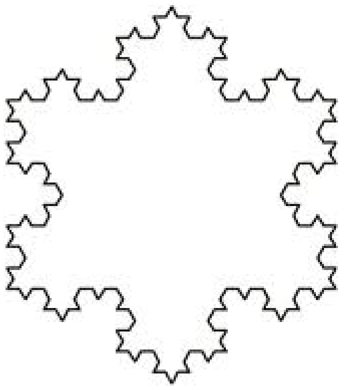
PTP, which stands for Point-To-Point, is a simple, yet powerful language for generating graphics programmatically. Its build-in commands simulate the moments of a human hand holding a pen over a white rectangular canvas. The hand can change position and orientation, and move forward and back while holding a pen with certain width and color up or down on the canvas, leaving a trace when pen is down. The result of running a PTP program is a PostScript file, containing image produced by the movements. PostScript files can easily be converted to widely used document types such as PDF, which means with little coding effort one could create easily viewable complex graphical images such as:

Dragon Curve:





Hilbert Curve:



Koch Curve:

Gosper Curve:

Lexical Conventions

Comments
Begin with //, end at the end of the line.

Identifiers

PTP is case-sensitive! Example: A is not the same as a.

There are two types of identiers: routine identifiers and variable identifiers.

Routine identifiers are strings of alpha-numeric characters, beginning with a letter,
can also contain the underscore character '_'.

Variable idenfitiers are the same as routine identifiers, but with the '@' character prepended.

Keywords
A small number of special identifiers are reserved for the language and its built-in features:

	● setc
● to	● setw
● twd	● return
● fw	● rep
● bk	● while
● lt	● if
● rt	● else
● pu	● routine
● pd	● print

Constants and Literals

Only integer constants are defined, which are a sequence of one of more digits, starting with non-zero digit.

Operators

Note: the operators allowed applied to integers produce integers!

Addition with +, left-associative

Subtraction with –, left associative

Multiplication with * , left associative

Punctuators

Statements end with semi-colon

Blocks of statements and routine bodies are enclosed in curly braces

Routine inputs as enclosed in parenthesis

Inputs in the input lists are separated by comma

Nesting of routines: not allowed

Meaning of Identifiers

Scope

All identifiers are local (are visible and recognized inside the block they are defined in), there are no global identifiers. Routines are recognized only if they have been defined before they are called.

Storage Duration

Storage lasts until control exits the function.

Types

The only supported type is integer.

Expressions and Operators

Precedence and Associativity Rules

Highest to lowest:

Multiplicative operators: * (multiplication) , left associative

Additive operators: + (addition), - (subtraction), left associative

Relational operators: < (less than), > (greater than), <= (less than or equal), >= (greater than or equal), left associative

Equality operators: == (equal), != (not equal), left associative

Assignment: = (equal sign), right associative

Primary Expressions

Simple integers like 0,1,2,3, etc or variables which have integer values)

Multiplicative Operators (*)
    Expression*expression

Additive Operators (+,-)
    Expression + expression
    Expression - expression

Relational Operators (>, <, <=, >=)
    Evaluate to 1 if condition is true, to 0 otherwise
    Expression > expression
    Expression < expression
    Expression >= expression
    Expression <= expression

Equality Operators (==, !=)
    Evaluate to 1 if condition is true, to 0 otherwise
    Expression == expression
    Expression != expression

Logical Operators (and/or): no supported (neither AND nor OR is supported)

Assignment Operator (=)
    Variable = <expression>;
    Example: @a = 5;

Constant Expressions

Cannot contain assignments or function calls, must evaluate to a constant that
is in the range of representable values for its type. Since PTP only supports
integers, every constant expression must evaluate to an integer, otherwise behavior is undefined.

Declarations

Type Specifiers
There are no type specifiers, integer is implied.

Declaring and Initializing Variables

All variable identifiers begins are just like regular identifiers, but are prefixed with @.
@<identifier> = <initial int value or expression evaluating to int value>;
If the variable is not explicitly initialized, it is implicitly initialized to 0.

Example: @a; //declares and initializes variable @a of type int to 0.

Function Declarations/Definitions (routine foo(@a1, @a2) { <statements> })

routine <identifier>(<optional arg list>) { statements }
The functions in PTP do not return anything, their nature is procedural.

Calling functions in PTP

Routines in PTP can call themselves recursively.

When calling a routine, the number and type of arguments supplied must correspond to the function definition / declarations.

Function inputs are passed by value, so no side effects possible on the inputs.

Type Names: no type names, integer type is implied.

Initialization (@a = 4;)
<variable identifier> = <integer literal or constant expression>;

Statements

Expression Statements: evaluate to an integer (+,-,*, >, <, <=, >=, ==, !=)
@a = 5+@b;

Compound Statements or Block: { … }

{@a = 5+@b; @b = @a-3;}

Selection Statements: if

if (<expression>) <statement executed if expression evaluates to non-zero>
else <statement executes if expression evaluates to 0>

Note: else clause is optional.

Iteration Statements: rep, while

rep <expression> <statement>
Repeatedly execute <statement> <expression> number of times.
If the expression evaluates to a negative number, the statement is not executed.

Example: rep 5 { @a= @a+1; }

while <expression> <statement>
Repeatedly execute <statement> while expression evaluated to anything other than 0.
Example: while (@a) { @a = @a-1;}

<u>Jump Statements</u>: return

Exits the current routine
Example: routine abc() { … return; … }

<u>The main() routine</u>

Program execution begins from the main routine.
The main routine can call other routines defined above it.
The main routine does not take any inputs, and does not return a value.
After the main routine exits, the program exists, and output.ps is generated in the local dir, containing the generated image, if any.

<u>Build-in Routines</u>

- to(@x, @y); //move to an absolute position on the canvas, where 0,0 is bottom left
- twd(@degrees); //position towards an absolute angle (0 is towards the bottom right corner, 90 towards top left, etc)
- fw(@distance); //move forward distance number of points
- bk (@distance); // move back distance number of points
- lt (@degrees); //turn left degrees number of degrees
- rt(@degrees); //turn right degrees number of degrees
- pu(); //pen up, or do not leave trace when moving
- pd(); //pen down, or leave trace when moving
- setc(@r, @g, @b); //set the color of the pen using red, green and blue values b/w 0 and 100
- setw(@w); //set the width of the pen to be w points
- print(@v); //print a value to the console, for debugging purposes

<u>Program structure</u>

Any number of uniquely named routines followed by the main routine.

<u>State at start-up</u>
Hand is at absolute position 0,0 (lower left corner of the canvas) at absolute angle 0 (such that forward movement would be towards the lower right corner, if done).

The pen is up (would not leave trace if hand is moved).

Pen is the default pen (width 1, color black).

Handling of drawing-beyond-canvas-borders situations

Anything drawn beyond the canvas borders will not show up in the final graphic produced.

<u>Result of program execution</u>

Program execution begins from the main routine.

After the main routine completes (that is, executes its last statement and returns), the program exists, and a PostScript file names output.ps is generated in the current folder / directory. If the file already exists, it will be over-written.

Sample program

sample.ptp

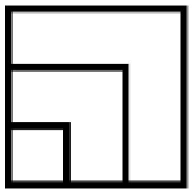
```
//draw a single square, lower left corner at current location
//side effect on position and orientation after executing: none
function square_fd_lt(@side)
{
    repeat 4
   {
      fd( @side );
       lt(90);
    }
}

function main()
{
    to(10,10); //go to position 10,10
    pd();//   put pen down

    //draw 3 squares, sides 10,20, and 30, with lower left corner at 10,10
    @a = 10;
    while (40 - @a)
    {
             square_fd_lt( @a );  //draw a square with lower left corner at current position
             @a = @a +10;
    }
}
```

Output

output.ps looks like:



Semi-Formal Syntax of the language:

```
Program:
    Main_routine
|   Functions Main_routine

Functions:
  Function
| Functions Function
```

```
Main_routine:
      routine main ( ) Block_of_statements

Function:
      routine ID ( Param_list ) Block_of_statements

Block_of_statement:
      { Statements }

Statements:
  Statement
| Statements Statement

Param_list:
| VAR
| Param_list , VAR

Statement:
ID ( Arg_list );
| VAR = Expression;
| If_statement
| Repeat_statement
| While_statement
| return;

Arg_list:
| Expression
| Arg_list , Expression

If_statement:
  if ( Expression ) Block_of_statements
| if ( Expression ) Block_of_statements else Block_of_statements

Repeat_statement:
  repeat Expression Block_of_statements


While_statement:
  while ( Expression ) Block_of_statements

Expression:
  INT_LITERAL
| VAR
| Expression + Expression
| Expression - Expression
| Expression * Expression
| Expression < Expression
| Expression <= Expression
| Expression > Expression
| Expression == Expression
```

| Expression != Expression

INT_LITERAL pattern: [‘1’-’9’]+[‘0’-’9’]*
VAR pattern: ‘@’[‘a’-’z’ ‘A’-’Z’]+[‘0’-’9’ ‘a’-’z’ ‘A’-’Z’ ‘_’]*
ID pattern: [‘a’-’z’ ‘A’-’Z’]+[‘0’-’9’ ‘a’-’z’ ‘A’-’Z’ ‘_’]*