Point-To-Point (PTP)

Final Report

PTP - Minimalistic Programming Language for generating graphics programmatically

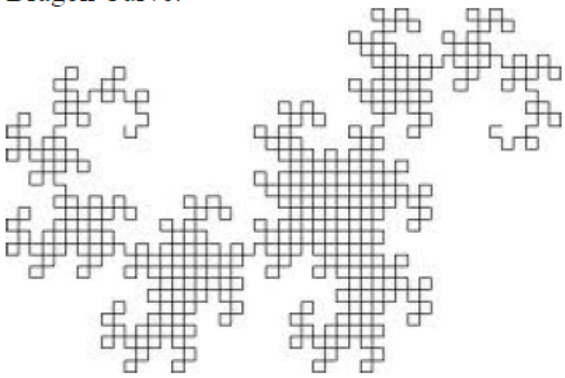
COMS 4115, Columbia University, Summer 2011

Venera Varbanova (vv2200@columbia.edu)

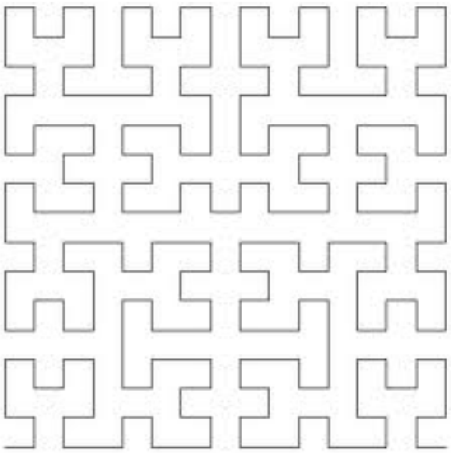
1. Introduction

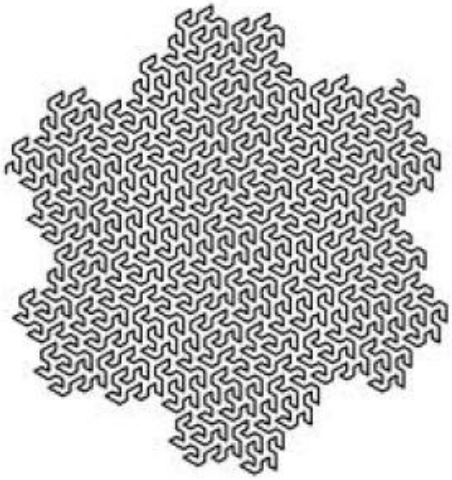
PTP, which stands for Point-To-Point, is a simple, yet powerful language for generating graphics programmatically. Its built-in commands simulate the movements of a human hand holding a pen over a white rectangular canvas. The hand can change position and orientation, and move forward and back while 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 the image produced by the movements. PostScript files can be rendered using a viewer such as GSView, and the 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:



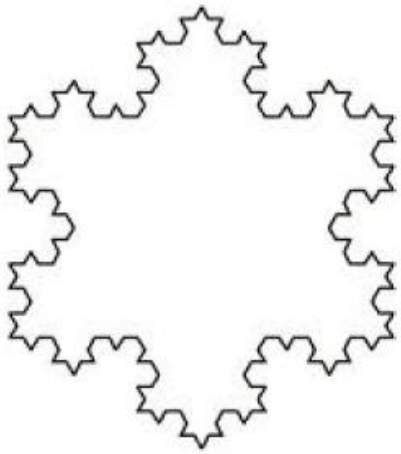
Hilbert Curve:





Gosper Curve:

Koch Curve:



2. Language reference manual

2.1. Comments

2.1.1. Begin with /*, end with */

2.2. Identifiers

2.2.1. Alpha-numeric, begin with a letter

2.2.2. Case sensitive

2.2.3. Used for identifying routine names, local, and global variables

2.3. Keywords / Built-in features

2.3.1. Generic

2.3.1.1. print – prints an integer value to stdout

2.3.2. Drawing-related

2.3.2.1. pendown – set the pen's drawing mode to ON (leaves a trace when moving)

2.3.2.2. penup – set the pen's drawing mode to OFF (does not leave a trace when moving)

2.3.2.3. fwd – moves forward a given via argument number of points in the direction of the current orientation

2.3.2.4. left – changes the orientation of the hand by a turning it a supplied via an argument number of degrees to the left

2.3.3. Flow-control related

2.3.3.1. return – returns control to the call, always returns an integer value

2.3.3.2. if – if predicate is true, control goes to the block following the predicate

2.3.3.3. else – must be used with if, and if predicate is false, control goes to the block following the else

2.3.3.4. for – execute a block while predicate is true, supplying int expression, predicate expression , and change expression

2.3.3.5. while – execute a block while the predicate evaluates to true

2.4. Constants and Literals

2.4.1. Only integers are supported

2.5. Arithmetic Operators – operate on ints, and return ints

2.5.1. Plus (+)

2.5.2. Minus (-)

2.5.3. Times (*)

2.5.4. Divide (/)

2.6. Boolean / True-False/ 0-1 Operators

2.6.1. Equal (==)

2.6.2. Not Equal (!=)

2.6.3. Greater than (>)

2.6.4. Greater or equal (>=)

2.6.5. Less than (<)

2.6.6. Less than or equal (<=)

2.7. Other operators / Punctoators

2.7.1. Comma – used to separate items in lists of function arguments

2.7.2. Semi-column – used to sequencing of instructions, separate statements from one another, as well as in the for loop construct

2.7.3. Curly braces – used for combiniting several instructions in a block, including function bodies

2.7.4. Parens – used for enclosing function arguments and if/for/while predicates

2.8. Meaning of Identifiers

2.8.1. Scope

2.8.1.1. global variables and functions are visible everywhere in the program, local variables and function inputs only visible inside the function body, no nesting of functions allowed

2.8.2. Storage Duration

2.8.2.1. Storage of locals lasts until control exits the function.

2.8.2.2. Storage of globals lasts throught the program execution

2.8.2.3. Storage of the resulting postscript file is persistent (on disk, inside the same location the program was run)

2.8.3. Types

2.8.3.1. Only supposed type is integer / int.

2.9. Expressions and Operators

2.9.1. Precedence and Associativity

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

2.10. Primary Expressions

- Simple integers like 0,1,2,3, etc or variables which have integer values)
- Multiplicative Operators (*, /)

.1. Expression * expression

.2. Expression / expression (integer division)

- Additive Operators (+,-)
    .1. Expression + expression
    .2. Expression - expression
- Relational Operators (>, <, <=, >=)
    .1. Evaluate to 1 if condition is true, to 0 otherwise
    .2. Expression > expression
    .3. Expression < expression
    .4. Expression >= expression
    .5. Expression <= expression
- Equality Operators (==, !=)
    .1. Evaluate to 1 if condition is true, to 0 otherwise
    .2. Expression == expression
    .3. Expression != expression
- Logical Operators (and/or): no supported (neither AND nor OR is supported)
- Assignment Operator (=)
    .1. Variable = <expression>;
    .2. Example: a = 5;
- Variable declaration
    .1. int <varname>;
    .2. Example: int a;

2.11. 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.

2.12. Declarations

2.12.1. Type Specifiers

2.12.1.1. There are no type specifiers, integer is implied.

2.12.2. Declaring and Initializing Variables

2.12.2.1. Declare: int <varname>;

2.12.2.1.1. Example: int a;

2.12.2.2. Initialize: <varname> = <expr>;

2.12.2.2.1. Example: a = 4;

2.13. Functions

2.13.1. <identifier>(<optional comma-separated arg list>) { statements }

2.13.2. All functions return an integer value, If no value is explicitly returned, 0 is imlicitly returned.

2.13.3. Function calls are expressions, evaluate to an integer, which could be captured or dropped.

2.13.4. Functions are called by supplying the function name, followed by a set of parens, inside which should be the arg list as defined by the function signature. Num args supplied should be the same as the number of args in the function declaration.

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

2.14. Statements and Conditionals

2.14.1. Expression statements

2.14.1.1. Evaluate to an integer

2.14.1.1.1. (+,-,*, /)

2.14.1.2. Evaluate to 0/1

2.14.1.2.1. (>, <, <=, >=, ==, !=)

2.14.2. Compound Statements or Block

2.14.2.1. { <statements> }

2.14.3. If Statement

2.14.3.1. If (predicate) then <true block>

2.14.3.2. If (predicate) then <true block> else <false block>

2.14.4. Iterator Statements

2.14.4.1. Repeadly execute a block of code while the predicate is true / 1

2.14.4.2. For Statement

2.14.4.2.1. For (init; predicate; effect) <block>

2.14.4.3. While Statement

2.14.4.3.1. While (predicate) <block>

2.14.5. Jump Statements

2.14.5.1. Return – exist the current function, returns an int

2.14.5.2. Example: foo() { … return 1; … }

2.14.6. The main() routine

2.14.6.1. 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.

2.14.7. Built-in Routines

2.14.7.1. Print(arg);

2.14.7.1.1. Prints the argument to stdout

2.14.7.2. Penup();

2.14.7.2.1. Sets the pen’s mode to OFF (non-drawing mode)

2.14.7.3. Pendown();

2.14.7.3.1. Set's the pen's mode to ON (drawing mode)

2.14.7.4. Fwd(steps);

2.14.7.4.1. Move the hand a given number of steps in the current hand's orientation. If the pen's mode is ON, a trace is left and written to the output postscript file.

2.14.7.5. Left(degrees);

2.14.7.5.1. Changes the orientation of the hand by the given number of degrees to the left

2.14.8. Program Structure

2.14.8.1. Any number of uniquely named global variables, any number of uniquely named functions, and a mandatory main function.

2.14.9. State at start-up

2.14.9.1. 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).

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

2.14.9.3. Pen is the default pen (width 2, color black).

2.14.10. Result of program execution

2.14.10.1. 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 named output.ps is generated in the current folder / directory where the program was run. If the file already exists, ti will be overwritten.

2.14.11. Sample program

sample.ptp

```
/* draw a single square, lower left corner at current location
   side effect on position and orientation after executing: none */
```

```
square_fd_lt(side)
{
   int i;
   i = 4;
  while (i > 0){
            fwd( side );
            left(90);
            i = i – 1;
  }
}
 main()
{
    int a; /* declare helper variable a */

    fwd(400); /* go to position 400,0, facing east , along the x axis in cartesian plane */

     pendown(); /*  put pen down */
     /* draw 3 squares, sides 100, 200, and 300  */
     a = 100;
    while (400 - a)
    {
        square_fd_lt( a );  /* draw a square, lower left corner at 400,0 */
        a = a +100;
    }
}
```

Output from this program (seen by opening output.ps in GSView or converting output.ps to output.pfd):

3. Project Plan

   3.1. June - planning, research; vague idea of the language

   3.2. July – scanner, parser, reevaluating initial plans; syntax and semantics of the language

   3.3. August – compiler, testing, debugging; the actual implementation of the language

4. Design

   4.1. My project is based on the microc compiler discussed in class, and as such, it follows the same design as microc.

   Scanner ->Parser -> Ast -> Bytecode -> Execution

   The requirement was to build a compiler rather than an interpreter, and so I did not enhance the interpreter part of microc, just the compiler part. It would be very easy to enhance the interpreter part, if needed.

5. Test Plan

   I prepared a bunch of ptp programs for which I hand-calculated the expected output, and this output I saved to specially named .ans files, and I wrote a .bat file to compile and run these programs and verify the output againts the corresponding to the program's name .ans file.

   This made it possible to test all features on ptp in less than a minute, which was helpful because I needed to test very often, basically after every little change I made anywhere in the code.

6. Lessons Learned / Advice for Students Taking the Class in the future (esp. CVN aka 'TV Land' students)

   6.1. Start early so that you have plenty of time to make a choice about what you want to spend the rest of the semester working on. You really want to avoid picking something quickly before the proposal is due, and then realizing you are stuck with it for the rest of the semester.

   6.2. Once you've settled on the general idea of your language, think, think again, think it over all over again, and only then do. Starting doing / coding without having thought through the entire thing completely is just going to waste your time in redoing it all over again from scratch many times.

   6.3. Being a CVN student, it's hard to not have a TA to consult with in their office hours when in doubt whether you're going in the right direction. If you have a choice, opt for taking the class "live".

6.4. Working alone has its advantages (not having the overhead to coordinate and divide work), but also has its disadvantages (like not having someone to discuss your ideas with and be sure you're not going totally off track). By explaining what your idea is to someone else, you better understand your own idea and often realize its flaws or strong points. So consider finding a team, even if you are a CVN student, I think your project will go better.

6.5. Build the language incrementally. Choose simple but doable over complex and hardly doable. First, make sure the most basic thing about your language works. Then add a second. Make sure both still work together. Hopefully automate your tests via a script else they will take you forever to execute by hand every time you change something. Only then add another feature, make sure the 3 work together, and so on. Once you get the first feature done, the process you go through to add the rest is very similar and feels familiar and easy. As a principle, don't go on to the next feature until you have written and ran the test case for the previous.

6.6. Thinking in functional programming terms is hard if you have not done it before. Watch the lambda calculus lecture in advance of doing the project, if you can. It helped me grasp functional programming a bit better.

6.7. Working on a programming project on Windows is really tough. Unfortunately , my laptop runs Windows 7 (64bit) and it was a bit hard to get everything to run. I had to learn windows shell scripting, which only made me realize how doing any school work on windows is not a good idea. So if you have a choice, do your project on unix. You'll find the setup process easier, and more examples that are made for unix.

6.8. Use OcaIDE extention for Eclipse. I tried all kinds of IDE for ocaml, and this one was definitely the best, especially the auto-build project feature that auto-recompiles the needed parts

immediately after you change something, so errors are caught on the stop. Also shows you the types and signatures, which you can't see in a simple text editor.

6.9. Put trace statements in the compiler code as a debugging tool. Look at bytecode and inspect the values of PC, SP and FP and see if they make sense in the trace you get from compiling and executing your program. Sometimes when you call the built-in function with the incorrect number of args for them (like print() with no args), strange things happen (in this case, because you execute drop without having pushed anything on the stack, not even 0, the fix: push 0 on the stack when there are no args to a built-in function in a microc-like language).

Appendix

1.1. Scanner.mll

```
(* 1) Token rules *)
{ open Parser } (* Get the token types *)

rule token = parse
[' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*" { comment lexbuf } (* Comments *)
| '(' { LPAREN }
| ')' { RPAREN } (* punctuation *)
| '{' { LBRACE }
| '}' { RBRACE }
| ';' { SEMI }
| ',' { COMMA }
| '+' { PLUS }
```

```
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIVIDE }
| '=' { ASSIGN }
| "==" { EQ }
| "!=" { NEQ }
| '<' { LT }
| "<=" { LEQ }
| '>' { GT }
| ">=" { GEQ }
| "if" { IF } (* keywords *)
| "else" { ELSE }
| "for" { FOR }
| "while" { WHILE }
| "return" { RETURN }
| "int" { INT }
| eof { EOF } (* End-of-file*)
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) } (* integers *)
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| _ as char { raise (Failure("illegal character " ^

        Char.escaped char)) }
and comment = parse
"*/" { token lexbuf } (* End-of-comment*)
| _ { comment lexbuf } (* Eat everything else *)
```

1.2. Parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA PLUS MINUS TIMES
DIVIDE
%token ASSIGN EQ NEQ LT LEQ GT GEQ RETURN IF ELSE FOR WHILE INT EOF
%token <int> LITERAL
%token <string> ID

%nonassoc NOELSE
%nonassoc ELSE

%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%
```

```
program:
/* nothing */ { [], [] }
| program vdecl { ($2 :: fst $1), snd $1 }
| program fdecl { fst $1, ($2 :: snd $1) }

fdecl:
    ID LPAREN formals_opt RPAREN
    LBRACE vdecl_list stmt_list RBRACE
    { { fname = $1;
    formals = $3;
    locals = List.rev $6;
    body = List.rev $7 } }

formals_opt:
/* nothing */ { [] }
| formal_list { List.rev $1 }

formal_list:
ID { [$1] }
| formal_list COMMA ID { $3 :: $1 }

vdecl_list:
/* nothing */ { [] }
| vdecl_list vdecl { $2 :: $1 }

vdecl:
```

```
INT ID SEMI { $2 }

stmt_list:
/* nothing */ { [] }
| stmt_list stmt { $2 :: $1 }

stmt: expr SEMI { Expr($1) }
| RETURN expr SEMI { Return($2) }
| LBRACE stmt_list RBRACE { Block(List.rev $2) }
| IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
| IF LPAREN expr RPAREN stmt ELSE stmt { If($3, $5, $7) }
| FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
{ For($3, $5, $7, $9) }
| WHILE LPAREN expr RPAREN stmt { While($3, $5) }

expr:
LITERAL { Literal($1) }
| ID { Id($1) }
| expr PLUS expr { Binop($1, Add, $3) }
| expr MINUS expr { Binop($1, Sub, $3) }
| expr TIMES expr { Binop($1, Mult, $3) }
| expr DIVIDE expr { Binop($1, Div, $3) }
| expr EQ expr { Binop($1, Equal, $3) }
| expr NEQ expr { Binop($1, Neq, $3) }
| expr LT expr { Binop($1, Less, $3) }
| expr LEQ expr { Binop($1, Leq, $3) }
```

```
| expr GT expr { Binop($1, Greater, $3) }
| expr GEQ expr { Binop($1, Geq, $3) }
| ID ASSIGN expr { Assign($1, $3) }
| ID LPAREN actuals_opt RPAREN { Call($1, $3) }
| LPAREN expr RPAREN { $2 }

expr_opt:
/* nothing */ { Noexpr }
| expr { $1 }

actuals_opt:
/* nothing */ { [] }
| actuals_list { List.rev $1 }

actuals_list:
expr { [$1] }
| actuals_list COMMA expr { $3 :: $1 }
```

1.3. Ast.ml

```
(* 3) Abstract syntax tree type and pretty printer *)

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq


type expr = (* Expressions *)
                Literal of int (* 42 *)
          | Id of string (* foo *)
          | Binop of expr * op * expr (* a + b *)
          | Assign of string * expr (* foo = 42 *)
          | Call of string * expr list (* foo(1, 25 *)
          | Noexpr (* for (;;) *)


type stmt = (* Statements *)
                Block of stmt list (* { ... } *)
          | Expr of expr (* foo = bar + 3; *)
          | Return of expr (* return 42; *)
          | If of expr * stmt * stmt (* if (foo == 42) {} else {} *)
          | For of expr * expr * expr * stmt (* for (i=0;i<10;i=i+1) { ... } *)
          | While of expr * stmt (* while (i<10) { i = i + 1 } *)


type func_decl = {
          fname : string; (* Name of the function *)
          formals : string list; (* Formal argument names *)
          locals : string list; (* Locally defined variables *)
          body : stmt list;
}
```

```
type program = string list * func_decl list (* global vars, funcs *)


let rec string_of_expr = function
                Literal(l) -> string_of_int l
        | Id(s) -> s
        | Binop(e1, o, e2) ->
                        string_of_expr e1 ^ " " ^
                        (match o with
                                        Add -> "+"
                                | Sub -> "-"
                                | Mult -> "*"
                                | Div -> "/"
                                | Equal -> "=="
                                | Neq -> "!="
                                | Less -> "<"
                                | Leq -> "<="
                                | Greater -> ">"
                                | Geq -> ">=")
                        ^ " " ^ string_of_expr e2
        | Assign(v, e) -> v ^ " = " ^ string_of_expr e
        | Call(f, el) ->
                        f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
        | Noexpr -> ""


let rec string_of_stmt = function
```

```
        Block(stmts) ->
                "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
    | Expr(expr) -> string_of_expr expr ^ ";\n";
    | Return(expr) -> "return " ^ string_of_expr expr ^ ";\n";
    | If(e, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^ string_of_stmt s
    | If(e, s1, s2) -> "if (" ^ string_of_expr e ^ ")\n" ^
                string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
    | For(e1, e2, e3, s) ->
                "for (" ^ string_of_expr e1 ^ " ; " ^ string_of_expr e2 ^ " ; " ^
                string_of_expr e3 ^ ") " ^ string_of_stmt s
    | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let string_of_vdecl id = "int " ^ id ^ ";\n"

let string_of_fdecl fdecl =
    fdecl.fname ^ "(" ^ String.concat ", " fdecl.formals ^ ")\n{\n" ^
    String.concat "" (List.map string_of_vdecl fdecl.locals) ^
    String.concat "" (List.map string_of_stmt fdecl.body) ^
    "}\n"

let string_of_program (vars, funcs) =
    String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
    String.concat "\n" (List.map string_of_fdecl funcs)
```

1.4. Interpret.ml (should probably not be included since I built a compiler for ptp, but I could have added code to this file to make it an interpreter)

```
(* 4) AST interpreter *)
open Ast

module NameMap = Map.Make(struct
        type t = string
        let compare x y = Pervasives.compare x y
    end)

exception ReturnException of int * int NameMap.t

let _pen = 0;;
let _x = 0;;
let _y = 0;;
let _a = 0;;


(* Main entry point: run a program *)

let run (vars, funcs) =
    (* Put function declarations in a symbol table *)
```

```
let func_decls = List.fold_left
                (fun funcs fdecl ->
                                NameMap.add fdecl.fname fdecl funcs)
                NameMap.empty funcs
in

(* Invoke a function and return an updated global symbol table *)
let rec call fdecl actuals globals =

        (* Evaluate an expression and return (value, updated environment) *)
        let rec eval env = function
                        Literal(i) -> i, env
                | Noexpr -> 1, env (* must be nonzero for the for loop predicate *)
                | Id(var) ->
                                let locals, globals = env in
                                if NameMap.mem var locals then
                                        (NameMap.find var locals), env
                                else if NameMap.mem var globals then
                                        (NameMap.find var globals), env
                                else raise (Failure ("undeclared identifier " ^ var))
                | Binop(e1, op, e2) ->
                                let v1, env = eval env e1 in
                                let v2, env = eval env e2 in
                                let boolean i = if i then 1 else 0 in
                                (match op with
                                                Add -> v1 + v2
```

```
                                | Sub -> v1 - v2
                                | Mult -> v1 * v2
                                | Div -> v1 / v2
                                | Equal -> boolean (v1 = v2)
                                | Neq -> boolean (v1 != v2)
                                | Less -> boolean (v1 < v2)
                                | Leq -> boolean (v1 <= v2)
                                | Greater -> boolean (v1 > v2)
                                | Geq -> boolean (v1 >= v2)), env
            | Assign(var, e) ->
                        let v, (locals, globals) = eval env e in
                        if NameMap.mem var locals then
                                v, (NameMap.add var v locals, globals)
                        else if NameMap.mem var globals then
                                v, (locals, NameMap.add var v globals)
                        else raise (Failure ("undeclared identifier " ^ var))
            (* TODO: add the code for the PTP built-in routines in a similar way, have
globals for position and orientation and pen mode *)
            | Call("print", [e]) ->
                        let v, env = eval env e in
                        print_endline (string_of_int v);
                        0, env
            | Call(f, actuals) ->
                        let fdecl =
                                try NameMap.find f func_decls
```

```
                                with Not_found -> raise (Failure ("undefined function "
^ f))
                        in
                        let ractuals, env = List.fold_left
                                (fun (actuals, env) actual ->
                                        let v, env = eval env
actual in v :: actuals, env)
                                ([], env) actuals
                        in
                        let (locals, globals) = env in
                        try
                                let globals = call fdecl (List.rev ractuals) globals
                                in 0, (locals, globals)
                        with ReturnException(v, globals) -> v, (locals, globals)
        in
        (* Execute a statement and return an updated environment *)
        let rec exec env = function
                        Block(stmts) -> List.fold_left exec env stmts
                | Expr(e) -> let _, env = eval env e in env
                | If(e, s1, s2) -> let v, env = eval env e in
                        exec env (if v != 0 then s1 else s2)
                | While(e, s) ->
                        let rec loop env =
                                let v, env = eval env e in
                                if v != 0 then loop (exec env s) else env
                        in loop env
```

```
        | For(e1, e2, e3, s) ->
                        let _, env = eval env e1 in
                        let rec loop env =
                                let v, env = eval env e2 in
                                if v != 0 then
                                        let _, env = eval (exec env s) e3 in
                                        loop env
                                else
                                        env
                        in loop env
        | Return(e) ->
                        let v, (locals, globals) = eval env e in
                        raise (ReturnException(v, globals))
in
(* call: enter the function: bind actual values to formal args *)
let locals =
        try List.fold_left2
                        (fun locals formal actual ->
                                                NameMap.add formal actual locals)
                        NameMap.empty fdecl.formals actuals
        with Invalid_argument(_) ->
                        raise (Failure ("wrong number of arguments to " ^ fdecl.fname))
in
let locals = List.fold_left (* Set local variables to 0 *)
                (fun locals local -> NameMap.add local 0 locals)
                locals fdecl.locals
```

```
        in (* Execute each statement; return updated global symbol table *)
        snd (List.fold_left exec (locals, globals) fdecl.body)

(* run: set global variables to 0; find and run "main" *)
in let globals = List.fold_left
                (fun globals vdecl -> NameMap.add vdecl 0 globals)
                NameMap.empty vars
in try
        call (NameMap.find "main" func_decls) [] globals
with Not_found -> raise (Failure ("did not find the main() function"))
```

1.5. Bytecode.ml

```
(* 5) Byte code type and pretty printer *)

type bstmt =
                Lit of int (* Push a literal *)
        | Drp (* Discard a value *)
        | Bin of Ast.op (* Perform arithmetic on top of stack *)
        | Lod of int (* Fetch global variable *)
        | Str of int (* Store global variable *)
```

```
        | Lfp of int (* Load frame pointer relative *)
        | Sfp of int (* Store frame pointer relative *)
        | Jsr of int (* Call function by absolute address *)
        | Ent of int (* Push FP, FP ->SP, SP += i *)
        | Rts of int (* Restore FP, SP, consume formals, push result *)
        | Beq of int (* Branch relative if top-of-stack is zero *)
        | Bne of int (* Branch relative if top-of-stack is nonzero *)
        | Bra of int (* Branch relative *)
        | Hlt (* Terminate *)

type prog = {
        num_globals : int; (* Number of global variables *)
        text : bstmt array; (* Code for all the functions *)
}

let string_of_stmt = function
                Lit(i) -> "Lit " ^ string_of_int i
        | Drp -> "Drp"
        | Bin(Ast.Add) -> "Add"
        | Bin(Ast.Sub) -> "Sub"
        | Bin(Ast.Mult) -> "Mul"
        | Bin(Ast.Div) -> "Div"
        | Bin(Ast.Equal) -> "Eql"
        | Bin(Ast.Neq) -> "Neq"
        | Bin(Ast.Less) -> "Lt"
        | Bin(Ast.Leq) -> "Leq"
```

```
        | Bin(Ast.Greater) -> "Gt"
        | Bin(Ast.Geq) -> "Geq"
        | Lod(i) -> "Lod " ^ string_of_int i
        | Str(i) -> "Str " ^ string_of_int i
        | Lfp(i) -> "Lfp " ^ string_of_int i
        | Sfp(i) -> "Sfp " ^ string_of_int i
        | Jsr(i) -> "Jsr " ^ string_of_int i
        | Ent(i) -> "Ent " ^ string_of_int i
        | Rts(i) -> "Rts " ^ string_of_int i
        | Bne(i) -> "Bne " ^ string_of_int i
        | Beq(i) -> "Beq " ^ string_of_int i
        | Bra(i) -> "Bra " ^ string_of_int i
        | Hlt -> "Hlt"

let string_of_prog p =
        string_of_int p.num_globals ^ " global variables\n" ^
        let funca = Array.mapi
                        (fun i s -> string_of_int i ^ " " ^ string_of_stmt s) p.text
        in String.concat "\n" (Array.to_list funca)
```

1.6. Compile.ml

```
(* 6) AST-to-byte code compiler *)
```

```
open Ast
open Bytecode

module StringMap = Map.Make(String)

(* Symbol table: Information about all the names in scope *)
type env = {
    function_index : int StringMap.t; (* Index for each function *)
    global_index : int StringMap.t; (* "Address" for global vars *)
    local_index : int StringMap.t; (* FP offset for args, locals *)
}

(* enum: get a list of things, and 2 ints (step, start), *)
(* and produce another list of tuples (int, thing) *)
(* if start is 1, and step is 1, the list a,b,c *)
(* will become (a,1),(b,2)(c,3) *)
(* val enum : int -> int -> 'a list -> (int * 'a) list = <fun> *)
(* aka. world's simplest memory allocator*)

(* this would have to change if we had different types of variables in the language, *)
(* but here all vars are ints, so step is passed as constant *)
(* and we assume all vars are of size step*)
let rec enum step start = function
        [] -> []
    | hd:: tl -> (start, hd) :: enum step (start + step) tl
```

```
(* string_map_pairs:StringMap 'a -> (int * 'a) list -> StringMap 'a *)
(* val string_map_pairs :
  'a StringMap.t ->
                ('a * StringMap.key)
      list -> 'a StringMap.t = <fun>
# *)
let string_map_pairs map pairs =
      List.fold_left (fun m (i, n) -> StringMap.add n i m) map pairs

(** Translate a program in AST form into a bytecode program.
 Throw an exception if something is wrong,
e.g., a reference to an unknown variable or function *)
let translate (globals, functions) =

      (* Allocate "addresses" / indexes  for each global variable *)
      let global_indexes =
                string_map_pairs StringMap.empty (enum 1 0 globals)
      (* step = 1, start = 0 *)

      in

      (* Assign "addresses" / indexes to function names; *)
      (* builtin functions like "print", "penup", "pendown", etc*)
      (* are special *)
      let built_in_functions =
                StringMap.add "print" (-1) StringMap.empty in
```

```
let built_in_functions =
        StringMap.add "pendown" (-2) built_in_functions  in
let built_in_functions =
        StringMap.add "penup" (-3) built_in_functions  in
let built_in_functions =
        StringMap.add "fwd" (-4) built_in_functions  in
let built_in_functions =
        StringMap.add "left" (-5) built_in_functions  in

let function_indexes = string_map_pairs built_in_functions
                (enum 1 1 (List.map (fun f -> f.fname) functions))

in

(* Translate an AST function to a list of bytecode statements *)
let translate env fdecl =

        (* Bookkeeping: FP offsets for locals and arguments *)
        let num_formals = List.length fdecl.formals
        and num_locals = List.length fdecl.locals


        and local_offsets = enum 1 1 fdecl.locals (* locals go down the stack, starting with at 1,
        with stride 1, so local1 at 1, local2 at 2, and so on *)

        (* the args of the function do up the stack, starting at -2, with stride -1, *)
```

```
(* so arg1 is at -2, arg2 is at -3, arg3 is at -4, and so on *)
and formal_offsets = enum (-1) (-2) fdecl.formals in

(* NOTE: use of 'with' as part of the env type *)
let env = { env with local_index = string_map_pairs
                        StringMap.empty (local_offsets @ formal_offsets) } in

let rec expr = function
        Literal i -> print_endline("Literal"); [Lit i]
    | Id s -> print_endline("Id");
        (* first, try to find the identifier *)
        (* in the local vars or the function args *)
            (try [Lfp (StringMap.find s env.local_index)]

            (* if it is not among local vars and args, *)
            (* try global vars and other functions *)
            with Not_found -> try
                                [Lod (StringMap.find s
env.global_index)]
                                (* not found among args, locals, globals, other
funcs,*)
                                (*  so not found at all anywhere, error *)
                                with Not_found ->
```

```
                                                raise (Failure ("undeclared
variable " ^ s)))

                (* evaluate e1, then e1, combine them *)
                | Binop (e1, op, e2) -> print_endline("Binop");expr e1 @ expr e2 @ [Bin op]

                | Assign (s, e) -> print_endline("Assign");expr e @

                        (* try to assign to local or arg first *)
                        (try [Sfp (StringMap.find s env.local_index)]
                        with Not_found -> try
                                        (* try to assign to global if local or arg
was not found*)
                                        (* under the given name *)
                                        (* TEST: assign value to a function in
the global index??? *)
                                        [Str (StringMap.find s
env.global_index)]
                                with Not_found ->
                                        raise (Failure ("undeclared
variable " ^ s)))

                (* call a function by its name, supply list of inputs *)
                (* NOTE: inputs evaluated right to left!!! *)
                (* beware of side-effects *)
```

```
                | Call (fname, actuals) -> print_endline((string_of_int
num_formals)^"Call"^(string_of_int (List.length actuals)));

                                (try
                                        let idx = StringMap.find fname env.function_index in
                                        (* evaluate each of the args, form a list of the *)
                                        (* evaluated args and append a jump to the function
location *)
                                        (* as the last instruction in the list *)
                                        (List.concat (List.map expr (List.rev actuals))) @
                                        [Jsr idx ] @
                                        (* NOTE: special addition for built-in functions *)
                                        (* before I added this, print(); followed by print(); would
loop for ever *)
                                        (if (idx < 0 && ((List.length actuals) = 0)) then
                                                [Lit 0;] else [])
                                with Not_found ->
                                        raise (Failure ("undefined function " ^ fname)))
                | Noexpr -> print_endline("Noexpr");[] (* so that the for loop works *)

        (* Translate a statement *)
        in let rec stmt = function
                        Block sl -> List.concat (List.map stmt sl)
                        (* if it is block consisting of list of statements, for each item *)
                        (* in the list, process it and the append to the end *)
```

```
| Expr e -> print_endline("expr+drop");expr e @ [Drp] (* Discard result *)

(* NOTE: here we use NUM_FORMALS (the number of args passed)*)
(* to calculate where to put the end result of *)
(* evaluting the expr, because this location is where*)
(* the caller will look for the result *)
| Return e -> expr e @ [Rts num_formals]

| If (p, t, f) ->
        (* generate the byte code instructions for the true case *)
            let t' = stmt t

            (* generate the byte code instructions for the false case *)
            and f' = stmt f in

            (* generate the byte code instructions *)
            (* for the condition check, *)
            (* put it first in the list *)
            (* of generated byte code *)
            (* instructions to be returned *)
            expr p

            @

            (* jump over the then/true case code if needed *)
            [Beq(2 + List.length t')]
```

```
                @

                (* the byte code for the true/then case *)
                t'

                @

                (* branch unconditionally past the false case *)
                (* cause we can reach this place only if we already executed *)
                (* the true case byte code, we don't want to execute both, right?
*)
                [Bra(1 + List.length f')]

                @

                (* last, append the byte codes for the false case*)
                (* to the list of byte code instructions to be returned *)
                f'

        | For (e1init, e2stop, e3change, blk) ->
                (* Rewrite into a while statement *)
                stmt (
                        Block(
                        [
                                Expr(e1init);
```

```
                While(e2stop, Block([blk; Expr(e3change)]))
              ]
              ))

| While (e, b) ->

              (* generate byte codes for the block of code *)
              (* inside the while loop, *)
              (* we need to know its length, that is*)
              (* how many byte code instructions *)
              (* it translated into, so that we can jump over them*)
              (* using the correct offset in the *)
              (* jump relative bytecode instruction *)
              let b' = stmt b

              (* generate the byte code to evaluate the predicate *)
              and e' = expr e

              in

              (* build the list of byte codes to be returned for while loop *)

              (* standard way of doing things: branch unconditionally *)
              (* past the code, avoids 2 branch instructions *)
              [Bra (1 + List.length b')]
```

```
@

(* the byte code instuctions for the code inside the while loop*)
(* come next *)
b'

@

(* the byte code for evaluating the loop predicate comes after*)
(* the block of whatever was inside the while *)
(* so basically we are turning a while-do *)
(* into an equivalent do-while because *)
(* do-while is more efficient *)
e'

@

(* after evaluating the loop predicate *)
(* if it is true, we need to go back to the beginning*)
(* of the block of code and run it *)
(* since we generated the byte codes for the block and the*)
(* for evaluating the predicate beforehand, *)
(* we know exactly where to jump *)
(* so that we land right in the beginning *)
(* of the byte codes for the while-loop block *)
[Bne (- (List.length b' + List.length e'))]
```

```
        (* Translate a whole function *)
        in

        (* Entry: allocate space for locals *)
        [Ent num_locals]

        @

        (* generate the byte code for the function body *)
        stmt (Block fdecl.body)

        @
        (* put the result of executing the function *)
        (* where the caller expects it *)
        [Lit 0; Rts num_formals] (* Default = return 0 *)

in

(* now that we have generated the byte code for the functions, *)
(* we know how many instruction everything is, *)
(* so we can calculate the indexes / offsets / addresses*)
(* and create the env as it is*)
(* when we enter the main function *)
(* in the very beginning *)
let env = {
```

```
        function_index = function_indexes;
        global_index = global_indexes;
        local_index = StringMap.empty }

in

(* Code executed to start the program: Jsr main; halt *)
let entry_function = try
                (* very classic, x86 does just that: *)
                (* in the beginnin, just to main, then halt *)
                (* well, x86 does some extra stuff like setting *)
                (* the exit code and such, which we omit to do here *)
                [Jsr
                (* lookup the address / index / location of the main function *)
                (* in our bytecode array *)
                (StringMap.find "main" function_indexes);
                (* classic: call main, then exit *)
                Hlt]

        (* TEST: make sure we can;t have 2 function with the same name *)
        (* TEST: try 2 function with same name, diff arg list *)
        (* TEST: try a global var and a function having the same name *)
        (* TEST: try to declare a function inside a function *)
        with Not_found -> raise (Failure ("no \"main\" function"))

in
```

```
(* Compile the functions *)
(* func_bodies is a list of lists of bytecode statements *)
let func_bodies =
        (* put the bytecode for startig the program *)
        (* which means 1 jump instr to the offset where*)
        (* main is in our bytecode array and then exit the program *)
        entry_function (* this is a list of 2 byte code instr: jump and halt *)

        :: (* this operator is making the entry function list the first *)
        (* element in the list of lists of bytecode statements *)
        (* each list in this list of lists containts the bytecode instuctions*)
        (* for a given function *)

        (* put the bytecode for the other, user-defined,*)
        (* non-mandatory like main, functions next *)
        List.map (translate env) functions

in

(* Calculate function entry points by adding their lengths *)
(* fun_offset_list is a list of ints which represent offsets *)
let (fun_offset_list, _) = List.fold_left
                (fun (l, i) f -> (i :: l, (i + List.length f))) ([],0)
                        func_bodies
in
```

```
(* replacing labels/indexes with actual offsets *)
(* after we have generated all bytecode and we know *)
(* what these offsets are *)

let func_offset =
        Array.of_list
        (List.rev fun_offset_list) (* NOTE: we reverse the list *)

in
(* Concatenate the compiled functions *)
        (* and replace the function indexes in  *)
        (* Jsr statements with PC values *)

{ num_globals = List.length globals;

        text = Array.of_list ( List.map (function Jsr i
                (* NOTE: Jsr with negatives are the special built in function*)
                (* leave them alone *)
                (* only calculate Program Counter (~ line number) *)
                (* values for function coming from user-defined code *)
                when i > 0 -> Jsr func_offset.(i)
                | _ as s -> s) (* no change for non-Jsr byte codes *)

                (* apply this to all generated byte instruction *)
                (* for all user defined function *)
```

```
                (List.concat func_bodies)

            )
    }
```

1.7. Execute.ml

```
    (* 7) Bytecode interpreter *)
    open Ast
    open Bytecode
    open Printf

    let file = "output.ps";;

    let execute_prog prog =
        let stack = Array.make 1024 0
        and globals = Array.make prog.num_globals 0
        and pos = Array.make 4 0 (* 0 is pen, 1 is x, 2 is y, 3 is orientation *)
        and oc = open_out file

        in

        let rec exec fp sp pc =
            if sp > 100 then raise (Failure("stack problem"));
            print_endline ("exec,fp="^string_of_int(fp)
```

```
^",sp="^string_of_int(sp)
^",pc="^string_of_int(pc)); match prog.text.(pc) with
    Lit i ->
        print_endline("LIT_"^string_of_int i);
        stack.(sp) <- i; (* put the literat on the stack *)
        exec fp (sp +1) (pc +1) (* increase the stack pointer and the
program counter *)
  | Drp ->
    print_endline("DRP");
    exec fp (sp -1) (pc +1) (* discard the top of the stack *)
  | Bin op ->
    print_endline("BIN");
    let op1 = stack.(sp -2)
    and op2 = stack.(sp -1) in
    stack.(sp - 2) <-
    (let boolean i = if i then 1 else 0 in
        match op with
            Add -> op1 + op2
        | Sub -> op1 - op2
        | Mult -> op1 * op2
        | Div -> op1 / op2
        | Equal -> boolean (op1 = op2)
        | Neq -> boolean (op1 != op2)
        | Less -> boolean (op1 < op2)
        | Leq -> boolean (op1 <= op2)
        | Greater -> boolean (op1 > op2)
```

```
                                | Geq -> boolean (op1 >= op2)) ;
                            exec fp (sp - 1) (pc +1) (* stack is now 1 less since we don't need
the top of it *)
                            (* we add into where the first arg was, so now the second is
dropped *)
        | Lod i ->
                print_endline("LOD_"^string_of_int i);
                stack.(sp) <- globals.(i) ;
                exec fp (sp +1) (pc +1) (* put a global with a given index on the stack,
stack grows *)
        | Str i ->
                print_endline("STR_"^string_of_int i);
                globals.(i) <- stack.(sp -1);
                exec fp sp (pc +1)
                (* store whatever is on top of the stack in a global, *)
                (* stack stays the same *)
        | Lfp i ->
                print_endline("LFP_"^string_of_int i);
                stack.(sp) <- stack.(fp + i);
                exec fp (sp +1) (pc +1) (* load frame pointer relative on top of stack,
stack grows *)
        | Sfp i ->
                print_endline("SFP_"^string_of_int i);
                stack.(fp + i) <- stack.(sp -1);
                exec fp sp (pc +1) (* put whatever is on top of the stack in fp relative
location, *)
```

```
                    (* stack stays the same *)


(* built-in function PRINT *)
                | Jsr(-1) ->
                    print_endline("PRINT");
                    print_endline ("===========> print "^string_of_int stack.(sp -1));
                    (* vv: was sp, not sp+1 *)
                    exec fp sp (pc +1)


(* built-in function PENDOWN *)
                | Jsr(-2) ->
                    print_endline("PENDOWN");
                    pos.(0) <- 1;
                    print_endline ("pendown " ^(string_of_int stack.(sp -1)));
                    exec fp sp (pc +1)


(* built-in function PENUP *)
                | Jsr(-3) ->
                    print_endline("PENUP");
                    pos.(0) <- 0;
                    print_endline ("penup " ^ (string_of_int stack.(sp -1)));
                    exec fp sp (pc +1)


(* built-in function FWD *)
                | Jsr(-4) ->
                    print_endline("FWD" ^ (string_of_int stack.(sp -1)));
```

```
                    (* TODO: improve this code, make it less imperative, use match or when
*)
                    if pos.(0) > 0 then print_endline ("=======> moveto " ^ (string_of_int
pos.(1)) ^ " " ^ (string_of_int pos.(2)));
                    if pos.(0) > 0 then fprintf oc "newpath\n%s\n"
                    ("" ^ (string_of_int pos.(1)) ^ " " ^ (string_of_int pos.(2)) ^  " moveto");
                    if pos.(3) = 0 then  pos.(1) <-  pos.(1) + stack.(sp -1);
                    if pos.(3) = 90 then pos.(2) <-  pos.(2)+ stack.(sp -1);
                if pos.(3) = 180 then pos.(1) <-  pos.(1) - stack.(sp -1);
                    if pos.(3) = 270 then pos.(2) <-  pos.(2) - stack.(sp -1);
                    if pos.(0) < 0 then raise (Failure ("drawing into negative x coord"));
                    if pos.(1) < 0 then raise (Failure ("drawing into negative y coord"));
                    if pos.(0) > 0 then print_endline ("===========> lineto " ^
(string_of_int pos.(1)) ^ " " ^ (string_of_int pos.(2)));
                    if pos.(0) > 0 then fprintf oc "%s\nstroke\n" ("" ^ (string_of_int pos.(1))
^ " " ^ (string_of_int pos.(2)) ^ " lineto");
                    print_endline ("fwd " ^ string_of_int stack.(sp -1));
                    exec fp sp (pc +1)

(* built-in function LEFT *)
                | Jsr(-5) ->
                    print_endline("LEFT"^ (string_of_int stack.(sp -1)));
                    if stack.(sp -1) mod 90 != 0 then raise (Failure ("bad orientation, must be
multiple of 90"));
                    pos.(3) <- pos.(3) + 90; if (pos.(3) = 360) then pos.(3) <- 0;
```

```
                print_endline ("left,orientation=" ^ string_of_int(pos.(3)) ^",top of stack
is" ^ (string_of_int stack.(sp -1)));
                exec fp sp (pc +1)

        | Jsr i ->
                print_endline("JSR_"^string_of_int i);
                stack.(sp) <- pc + 1 ;
                exec fp (sp +1) i

        | Ent i ->
                print_endline("ENT_"^string_of_int i);
                stack.(sp) <- fp;
                exec sp (sp + i +1) (pc +1)
        | Rts i ->
                  print_endline("RTS_"^string_of_int i);
                        let new_fp = stack.(fp) and new_pc = stack.(fp -1) in
                        stack.(fp - i -1) <- stack.(sp -1);
                        exec new_fp (fp - i) new_pc
        | Beq i ->
                print_endline("BEQ_"^string_of_int i);
                exec fp (sp -1) (pc + if stack.(sp -1) = 0 then i else 1)
        | Bne i ->
                print_endline("BNE_"^string_of_int i);
                exec fp (sp -1) (pc + if stack.(sp -1) != 0 then i else 1)
        | Bra i ->
                print_endline("BRA_"^string_of_int i);
```

```
                exec fp sp (pc + i)
              | Hlt ->  print_endline("HLT");()
        in
        let _ =   fprintf oc "%s\n\n2 setlinewidth\n" "%!PS"; print_endline("START");exec 0 0 0
        in print_endline("END"); fprintf oc "showpage\n\n%s\n" "%% end-of-output.ps";
close_out oc;
```

1.8. Toplevel.ml

```
open Parser

(* 8) Top-level *)
type action = Ast | Interpret | Bytecode | Compile

let _ =
        let action = if Array.length Sys.argv > 1 then
                        List.assoc Sys.argv.(1) [ ("-a", Ast);
                                ("-i", Interpret);
                                ("-b", Bytecode);
                                ("-c", Compile) ]
                else Compile in
```

```
let lexbuf = Lexing.from_channel stdin in
let program = Parser.program Scanner.token lexbuf in
match action with
        Ast -> let listing = Ast.string_of_program program
                in print_string listing
| Interpret -> ignore (Interpret.run program)
| Bytecode -> let listing = Bytecode.string_of_prog (Compile.translate program)
                in print_endline listing
| Compile -> Execute.execute_prog (Compile.translate program)
```

1.9. Makefile.bat

```
del *.exe
del *.mli
del *.cmo
del *.cmi
del *.cmx
del *.annot

ocamlyacc parser.mly
ocamlc -c ast.ml
ocamlc -c parser.mli
```

```
ocamlc -c parser.ml
ocamllex scanner.mll

ocamlc -c scanner.ml
ocamlc -c interpret.ml
ocamlc -c bytecode.ml
ocamlc -c compile.ml
ocamlc -c execute.ml
ocamlc -c toplevel.ml

ocamlc -o toplevel.exe parser.cmo scanner.cmo ast.cmo interpret.cmo bytecode.cmo compile.cmo
execute.cmo toplevel.cmo

toplevel.exe –b < test.ptp

toplevel.exe –c < test.ptp
```

1.10. Test.ptp

```
gcd(a, b) {
  while (a != b) {
    if (a > b) a = a - b;
    else b = b - a;
    }
```

```
   return a;
}

main()
{

int a;

/* test some math , if and while statements */
print(gcd(2,14));
print(gcd(3,15));
print(gcd(99,121));

fwd(200);
left(90);
fwd(200); /* at 200,200 */
pendown();

/* test drawing and for loop */
/* cube with lower right corner at 200,200 */
for(a = 0; a < 4; a = a+1)
{
 fwd(100);
 left(90);
}
```

```
penup();
}
```

1.11. Unit_tests.bat

```
cls
@echo off
echo ..
echo  Running PTP test suite
echo ..
for %%X in (unit_test_*.ptp) do (
        toplevel.exe  < %%~nX.ptp > %%~nX.ans
        fc %%~nX.out %%~nX.ans > %%nX.diff
        IF ERRORLEVEL 1 echo %%~nX FAIL
        IF ERRORLEVEL 0 echo %%~nX PASS
)
pause
```

Listing on unit tests: dir unit_test_*.ptp

Unit_test_empty_main.ptp

Unit_test_print.ptp

Unit_test_global_var.ptp

Unit_test_local_var.ptp

Unit_test_assign.ptp

Unit_test_function_no_args.ptp

Unit_test_function_args.ptp

Unit_test_return.ptp

Unit_test_math.ptp

Unit_test_if.ptp

Unit_test_ifelse.ptp

Unit_test_while.ptp

Unit_test_pendown.ptp

Unit_test_penup.ptp

Unit_test_fwd.ptp

Unit_test_left.ptp

Unit_test_gcd.ptp

Unit_test_square.ptp

Unit_test_undefined_symbol.ptp

Unit_test_duplicate_symbols.ptp

Unit_test_positive_func_args.ptp

Unit_test_negative_func_agrs.ptp

Also, I did lots of user testing where I open the .ps file in Gsview and make sure it renders as expected.

Easy-to-do and/or interesting possible enhancements to this project:

- Built in functions right, toposition, orient, backwards, erase
- Drawing predefined sharep
- Drawing text in different font styles
- Output file name supplied as arg
- Colors
- Arcs
- Line Widths
- Filling of shares
- Line Closing
- An interpreter for the language
- Define movements in cm or mm
- Extra checking for when the drawing goes beyond a predefined width/heigth of the field
- 3D version of ptp (just add another dimention)