Final Report

EasyDraw

Stanislav Ivaciov

COMS W4115 – Spring 2009 (CVN)
Dr. Edwards

Table of Contents

Introduction

The EasyDraw programming language relies on the microc programming language provided by Dr. Edwards both as reference and an implementation base.

EasyDraw programming language is designed to facilitate development of programming skills by novices. The language provides one-to-one mapping between general programming constructs and concepts and their graphical representation.

It has been shown that for many students it is easier to familiarize themselves with abstract concepts of Computer Science using graphical mnemonics. There is not much of such that is currently offered for undergraduate students just entering the field.

On the other side humanitarian and most other technical studies offer a wide range of visualization tools for their entry-level students.

Therefore EasyDraw strives to bridge the gap between computer and other sciences buy providing students with visualization techniques that can be used for introducing sub-routines, for and while loops, statements, and expressions.

Development and Execution

Programs in EasyDraw can be written with just a simple text editor that is capable saving files in ASCII format. Other that a text editor, such as Notepad, or VI, students will need EasyDraw interpreter that is available free of charge.

The EasyDraw interpreter reads source file and executes it on the fly without translating the source in to any student visible form.

Therefore development model for EasyDraw is as follows: a source file in EasyDraw is edited and saved, and then EasyDraw interpreter is invoke with this file as the parameter. Output is generated on the screen.

Language Manual

EasyDraw language is very simple to use and understand. It is designed specifically for novices and therefore lacks the complexity associated with general programming languages like C or Java.

Higher-Level Primitives

To facilitate understanding and learning EasyDra provides several higher-level primitive constructs such as circle, rectangle, square, and other similarly named primitives execution of which results in display of appropriate geometrical figure on the screen. The programmer specifies the rectangular coordinates of the figure and appropriate parameter as in the following example.

```
circle 50, 70, 45, 0
```

Execution of this statement results in display of a circle centered in (50, 70) x,y-coordinates with radius equals to 45, rendered in black color.
Types

EasyDraw strives to provide simple development experience with all the tools necessary to construct useful and comprehensible programs. For this purpose two types are provided: Number and Color. The language is statically typed.

Number

All variables of type Number are positive integers in the range from 0 to 2^{32} . Every variable of type Number must be declared. Declaration order is not important but it is recommended that students declare all variables in the beginning of a block as in C syntax following by their use. EasyDraw does not count on this, in fact it is similar to ECMAScript in that declarations can follow the use, and students are encourage to explore this feature after they master the C-like syntax.

For example, following code snipplet declares two variables of type Number with names x and y.

```
Number x;
Number y;
```

Variables can be given values in the declarations as so: Number x := 100;. Notice that EasyDraw uses := symbol to indicate assignment.

Color

Color type is a subset of Number type. EasyDraw distinbuishes between 256 colors beginning at 0 that is black and ending at 255 that is white. It statically checs that every variable of type Color does not get below or above this threeshold. There is no coercion or assignment from one type to another as in most other modern languages. This feature is designed to teach students to distinguisnhe between different types and to exercise caution in languages such as C or Java where coercion can lead to subtle bugs.

Following code snipplet declares a one variable of type Color and another variable of type Number followed by an attempt to assign one variable to another that will result in a syntax error.

```
Color c;
Number x;
c := x;  /* syntax error, declared by interpreter */
```

Program Structure

All programs in EasyDraw begin execution in the main method. The interpreted assumes the existence of such method in the source file and not providing it results in a syntax error.

For example following main method if does not appear to do much, begins execution of a EasyDraw program:

```
# file example1.ezd

main ()
{
}
```

If you type the above text in a file named example.ezd, save it, and execute it by passing it as a parameter to EasyDraw interpreter you will see a white screen that indicates successful interpretation of the file and readiness of the interpreter.

Other methods can be defined in a single .esd file. However, single file can be passed to the interpreter, this means that you program has to be written in a single file.

Other methods defined in a .esd file can be called from main method or other methods including recursive calls.

For example following code begins execution in main method, from which method named A is called from which method named B is called.

```
#file example2.esd

main()
{
        A();
}

A()
{
        B();
}

B()
{
}
```

Methods

Methods also known as sub-routines in EasyDraw group code elements in logically connected units. All files i.e., programs, must have at least one method named main to execute otherwise a syntax error will be declared by the interpreted.
Parameters can be passed between methods, but no parameters can be returned from a method, in other words all parameters are passed by value and no return values are allowed.
Following example illustrates previously stated concepts. Specifically, method named main is defined. It calls method A which call method B and passes it parameter x.

```
main()
{
A();
}

A()
{
        B(45);
}

B(Number x)
{
        Color c : = 0;
        Point x, c;
}
```

Scope

EasyDraw is statically scoped. All variables declared in a block of code will “hide” variable with the same name in the outer blocks. Syntax rules of the language are such that there only a single declaration with a particular name is allowed per block of code.

Following example declares two variables of type Number named x and y in the outer code block, following by redeclaration of x in the inner block.

```
{ /* outer block begins */
Number x := 10;
Number y := 20;
{ /* inner block begins */
        Number x := 15;
        /* Now x is 15 and y is 20 */
} /* inner block ends */
/* not x is 10 and y is 20 */
} /* outer block ends */
```

Comments

EasyDraw has single comment syntax. Starting with /* and ending with */ , anything in between the */ and the /* is considered to be a comment and is discarded during interpretation.
As an example consider the following code that declared a vatiable of type Color inside a comment. As is expected this declaration does not result in any execution.

```
/*
        This declaraiont will be ignored by the interpreter because it is inside a comment:
        Color x := 10;
*/
```

However following code will declare and assign number 10 to a variable of type Color during interpretation.

```
/* Following code will be executed because it is outside of the comment block */

Color x := 10;
```

Whitespace

ASCII character new line, tab, space, and new lines are all not considered by the EasyDraw interpreter. In fact they are removed before the interpreter sees the source program.

Following two code fragments are equivalent:

```
/* Fragment 1 */
        Number x := 1;
        Number y := 2;

/* Fragment 2 */
Number x:=1;Number y:=2;
```

However, whitespace is essential for the interpreter to distinguish between different tokens. Following code fragment is invalid because there is no whitespace between type and variable.

```
Numberx:=10; /* Syntax error */
```

Identifiers

Identifiers can be of arbitrary length. They must begin with a letter followed by zero ore more letters and digits. Following are some valid and invalid identifiers.
X /* valid */
xx01 /* valid */
0x /* invalid */
Keywords

Keywords are reserved identifiers. Programmers can not use names that are reserved by the language to name their identifiers.

if
then
else
for
while
number
color

Operators And Punctuation

Following is a precedence table for operators.

*, /	Multiply, Divide
+, -	Add, Subtract
:=	Assignment operation
&	Logical AND operator
|	Logical OR operator
=	Equivalence operator
<,>, <=, >=	Less than operator, Greater than operator, Less than or equal to, Greater that or equal to
!	Logical NOT operator
(,)	Left Parenthesis, Right Parenthesis
;	Statement delimiter.

Logical NOT operator is applied to and expression that has a type Boolean, even though EasyDra does not supply this type explicitly it can be obtained by performing comparisons between variables or literals of the same type.

Such as following comparison of two literal Numbers.

10 = 15; /* is false */

```
Number x := 10;
10 = x  /* is true */
```

Note that two different types can not be compared.

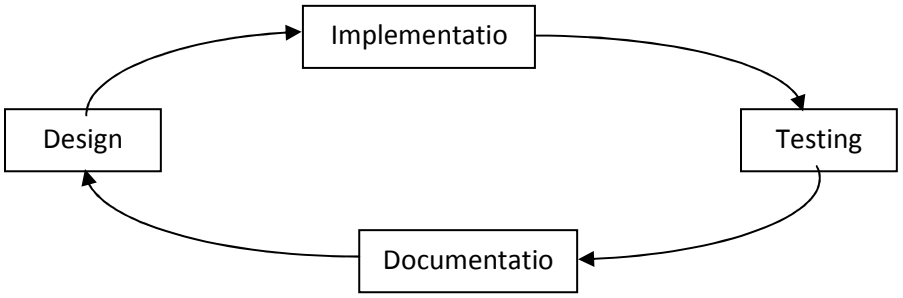
```
Number x := 10;
Color c := 10;

x = c;    /* produces syntax error */
```

Project Plan

Process

For each feature as well as for the project as a whole I tried to follow the iterative process illustrated in the following diagram.



Programming Style Guide

Style for syntax analyzer, because it is written as a set of patterns and has to confirm to the standards of OCamlLex tool following style has been used.

The file format is the header enclosed in to the braces '{' and '}', followed by rules for identifying tokes. Each rule consisteng of a regex and associated OCaml code for the action that returns to the parser. There is single rule that ignores input, which is the rule for comments. There is no trailer section in the final version but one was located after the rules section while testing for producing scanner output.

Parser stype is again made to confirm to the OCamlYacc tool. That is it has distinct sections for declarations, grammar rules, and additional OCaml code.

Now, the interpreter module is developed in OCaml with the following Sytle. Each function declaration nis indented as to create the visual flow of control representation to ease the reading of source code. Each pattern-matcing case is indented to facilitate distinction between different cases, and as much as possible of related code is kept on a single line.

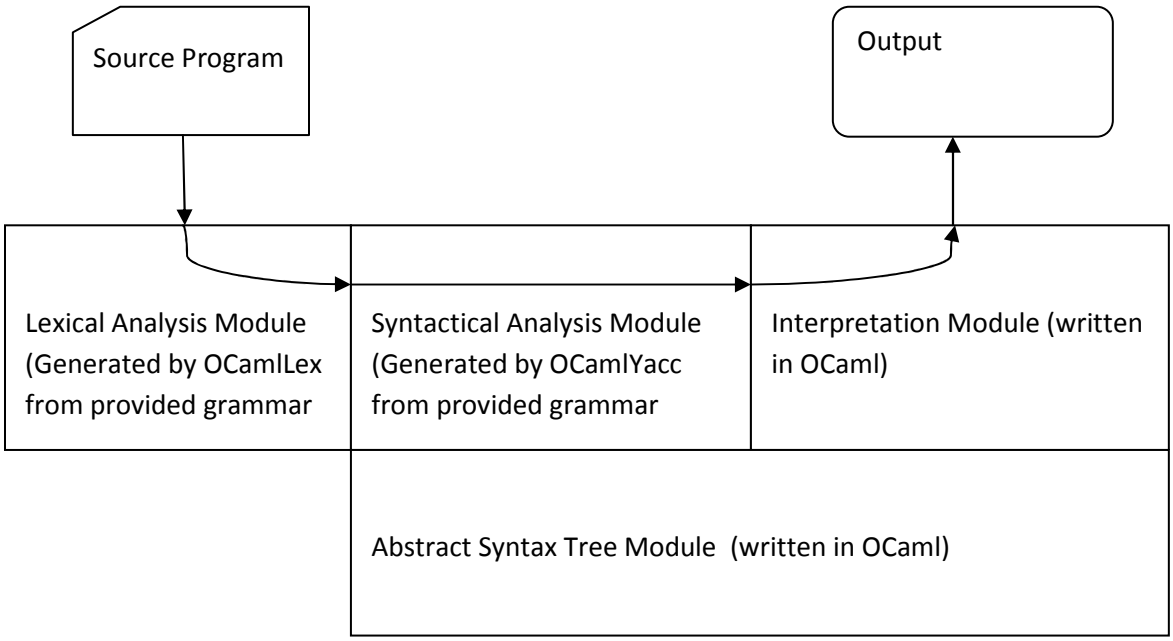
Software Development Environment

The project was developed on an Intel x86 machine with the following tools. Cygwin as operating system, Notepad++ as source text editor, UNIX simulation tools, specifically make as compilation aids.

And The Objective Caml toplevel, version 3.11.0 as programming language.

Architectural Design

The project's architecture is dictated by the Ocaml tools such as ocamllex, ocamlyacc and the fact that language is interpreted rather that compiled. Therefore there are modules for ocamllex, ocamlyacc, and interpreter. Following diagram depicts the modules and their interactions.



All the inter-module interfaces are defined by the set of tools that were used to generate the compiler, such as OCamlLex and OCamlYack. The general logical from is as follows.

Interpreter is invoked by the user with an input program. The interpreter calls the Syntactical Analyzer to parse the program, while it calls Lexical Analyzer to scan the input.

Test Plan

Due to shortage of time tests were developed on the fly. As each new feature was introduced to the language I have wrote small test program and run it to see the desired result. Later the tests were compiled into a test library.

Each language feature is embedded in to one of the test cases and the test cases are run using a shell script.

Many of the test cases were adapted after the test casef provided by Dr. Edwards in the microc reference implementation.

Example Program

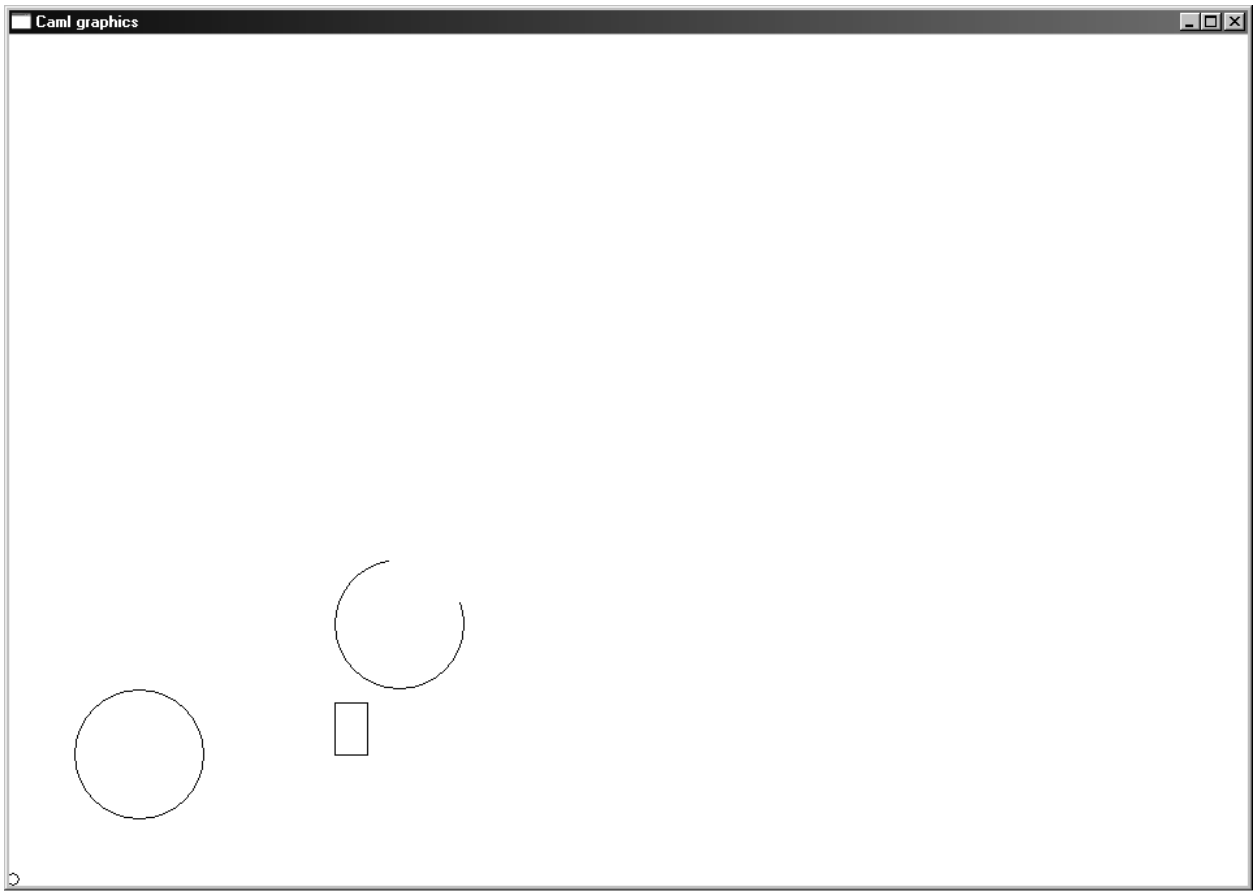
```
drawRect ()
{
      rectangle 250 100 25 40;
}

drawArc(x, y, rx, ry, a1, a2)
{
      /* arc x y rx ry a1 a2; */
      arc 300 200 50 50 100 20;

}


main ()
{
      int a;
      int b;
      a = 1;
      b = 2;
      print( a + b);
      circle 3 4 5;
      circle 100 100 50;
      drawRect ();
      drawArc(300, 200, 50, 50, 20, 20);
}
```

Output:



Lessons Learned

Designing and implementing a language is a challenging and interesting process. To be successful goals have to be defined long ahead of time that is always difficult because some of the issues do not come up until the implementation phase.

For example, I wanted to implement some higher order primitives like “circle”, “square” and others that proved much more difficult that I have anticipated. The reason being that a circle needs three parameters x, y coordinates and radius. This is fine as long as all parameters are specified either as literal values or as identifiers, but if a combination is required then a decision must be made while interpreting, to ether check the symbol tables or use the literal values. I was not able to implement this feature, and in the end decided to use only literal values.

On the positive side, I have learned a lot about mathematics and the underling structure of compilers. I think that OCaml is great not only for implementing compilers but for other general programming tasks. Many features provided by OCaml simplify the tasks at hand, but I have to admit that the language is not easy to learn.

One particular feature of OCaml that proved useful to me was the pattern-matching warnings that identified the code which does not account for all the cases. This was very helpful while developing compiler and interpreter.

Other great OCaml feature is its ability to compute types at compile time and notify me, the programmer, if there is a type mismatch. I found it particularly different from Java, C, and other languages where much less care is giver to the type system, and where it is much easier to produce type mismatch errors at runtime.

I would like to express special gratitude to our instructor for provided a working implementation of a programming language that I used as a reference and the base for my project.

Advice

My advice would be to learn OCaml as soon as possible, even simple programming tasks are much more different from iterative programming languages, but become more familiar with a lot of practice.

Other advise would be to start working on compiler in the first few weeks of the class and continue without any breaks until it is done.

Code

Interpreter.ml

```
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




(* 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 non-zero 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))
      | 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 actuals, 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 actuals 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
        | CircleLiteral(x, y, r) -> Graphics.draw_circle x y r; Unix.sleep 1; env
        | CircleId(x, y, r) -> env
        | RectangleLiteral (x, y, w, h) -> Graphics.draw_rect x y w h; Unix.sleep 1; env
        | RectangleId (x, y, w, h) -> env
        | ArcLiteral (x, y, rx, ry, a1, a2) -> Graphics.draw_arc x y rx ry a1 a2; Unix.sleep 1;
env
```

```
          | ArcId (x, y, rx, ry, a1, a2) -> env
          | EllipseLiteral (x, y, rx, ry) -> Graphics.draw_ellipse x y rx ry ; Unix.sleep 1; env
          | EllipseId (x, y, rx, ry) -> env
      | Return(e) ->
                      let v, (locals, globals) = eval env e in
                      raise (ReturnException(v, globals))
    in

    (* Enter the function: bind actual values to formal arguments *)
    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 passed to " ^ fdecl.fname))
    in
    (* Initialize local variables to 0 *)
    let locals = List.fold_left
        (fun locals local -> NameMap.add local 0 locals) locals fdecl.locals
    in
    (* Execute each statement in sequence, return updated global symbol table *)
    snd (List.fold_left exec (locals, globals) fdecl.body)

  (* Run a program: initialize 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
              Graphics.open_graph "EZDraw"; call (NameMap.find "main" func_decls) [] globals;
Unix.sleep 15;
  with
       Not_found -> raise (Failure ("Could not locate the main() function"))
```

Printer.ml

```
open Ast

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
  | CircleLiteral(x, y, r)              -> "circle with literal values("  ^ string_of_int x ^ ",
" ^ string_of_int y ^ ", " ^ string_of_int r ^ ")"
  | CircleId(x, y, r)                   -> "circle with id values(" ^ x ^ ", " ^ y ^ ", " ^ r ^
")"
  | RectangleLiteral(x, y, w, h)        -> "rectangle literal(" ^ string_of_int x ^ ", " ^
string_of_int y ^ ", " ^ string_of_int w ^ ", " ^ string_of_int h ^ ")"
  | RectangleId(x, y, w, h)             -> "rectangle literal(" ^ x ^ ", " ^ y ^ ", " ^ w ^ ", "
^ h ^ ")"
  | ArcLiteral(x, y, rx, ry, a1, a2)    -> "arc(" ^ string_of_int x ^ ", " ^ string_of_int y ^ ",
" ^ string_of_int rx ^ ", " ^ string_of_int ry ^ ", " ^ string_of_int a1 ^ ", " ^ string_of_int
a2 ^ ")"
  | ArcId(x, y, rx, ry, a1, a2)         -> "arc(" ^ x ^ ", " ^ y ^ ", " ^ rx ^ ", " ^ ry ^ ", " ^
a1 ^ ", " ^ a2 ^ ")"
  | EllipseLiteral(x, y, rx, ry)        -> "ellipse(" ^ string_of_int x ^ ", " ^ string_of_int y
^ ", " ^ string_of_int rx ^ ", " ^ string_of_int ry ^ ")"
  | EllipseId(x, y, rx, ry)             -> "ellipse(" ^ x ^ ", " ^ y ^ ", " ^ rx ^ ", " ^ ry ^
")"


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)
```

Ast.mli

```
type op =
          Add
        | Sub
        | Mult
        | Div
        | Equal
        | Neq
        | Less
        | Leq
        | Greater
        | Geq
        | CircleLiteral
        | CircleId
        | RectangleLiteral
        | RectangleId
        | ArcLiteral
        | ArcId
        | EllipseLiteral
        | EllipseId

type expr =
    Literal of int
  | Id of string
  | Binop of expr * op * expr
  | Assign of string * expr
  | Call of string * expr list
  | Noexpr


type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt
  | CircleLiteral of int * int * int
  | CircleId of string * string * string
  | RectangleLiteral of int * int * int * int
  | RectangleId of string * string * string * string
  | ArcLiteral of int * int * int * int * int * int
  | ArcId of string * string * string * string * string * string
  | EllipseLiteral of int * int * int * int
  | EllipseId of string * string * string * string

type func_decl = {
    fname : string;
    formals : string list;
    locals : string list;
    body : stmt list;
  }

type program = string list * func_decl list
```

Parser.mly

```
%{ open Ast %}

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

%nonassoc NOELSE
%nonassoc ELSE

%left 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) }
  | CIRCLE LITERAL LITERAL LITERAL SEMI                                { CircleLiteral($2, $3, $4) }
  | CIRCLE ID ID ID SEMI                                               { CircleId($2, $3, $4) }
  | RECTANGLE LITERAL LITERAL LITERAL LITERAL SEMI                     { RectangleLiteral($2, $3,
$4, $5) }
```

```
  | RECTANGLE ID ID ID ID SEMI                                          { RectangleId($2, $3, $4, $5)
}
  | ARC LITERAL LITERAL LITERAL LITERAL LITERAL LITERAL SEMI           { ArcLiteral($2, $3, $4, $5,
$6, $7) }
  | ARC ID ID ID ID ID SEMI                                           { ArcId($2, $3, $4, $5, $6,
$7) }
  | ELLIPSE LITERAL LITERAL LITERAL LITERAL SEMI                      { EllipseLiteral($2, $3, $4,
$5)}
  | ELLIPSE ID ID ID ID SEMI                                          { EllipseId($2, $3, $4, $5)}

expr_opt:
    /* empty */ { Noexpr }
  | expr          { $1 }

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 }

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

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

Scanner.mll

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n']   { token lexbuf }             (* Whitespace *)
| "/*"                   { comment lexbuf }           (* Comments *)
| '('                    { LPAREN }
| ')'                    { RPAREN }
| '{'                    { LBRACE }
| '}'                    { RBRACE }
| ';'                    { SEMI }
| ','                    { COMMA }
| '+'                    { PLUS }
| '-'                    { MINUS }
| '*'                    { TIMES }
| '/'                    { DIVIDE }
| '='                    { ASSIGN }
| "=="                   { EQ }
| "!="                   { NEQ }
| '<'                    { LT }
| "<="                   { LEQ }
| ">"                    { GT }
| ">="                   { GEQ }
| "if"                   { IF }
| "else"                 { ELSE }
| "for"                  { FOR }
| "while"                { WHILE }
| "return"               { RETURN }
| "int"                  { INT }
| "circle"               { CIRCLE }
| "rectangle"            { RECTANGLE }
| "arc"                  { ARC }
| "ellipse"              { ELLIPSE }
| ['0'-'9']+ as lxm      { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof                    { EOF }
| _ as char              { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  "*/" { token lexbuf }
| _    { comment lexbuf }
```

Makefile

```
OBJS = parser.cmo scanner.cmo printer.cmo interpret.cmo microc.cmo

TESTS = \
arith1 \
arith2 \
fib \
for1 \
func1 \
func2 \
gcd \
global1 \
hello \
if1 \
if2 \
if3 \
if4 \
ops1 \
var1 \
while1

TARFILES = Makefile testall.sh scanner.mll parser.mly \
	ast.mli interpret.ml printer.ml microc.ml \
	$(TESTS:%=tests/test-%.mc) \
	$(TESTS:%=tests/test-%.out)

microc : $(OBJS)
	ocamlc -o microc graphics.cma unix.cma $(OBJS)

.PHONY : test
test : microc testall.sh
	./testall.sh

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

%.cmo : %.ml
	ocamlc -c $<

%.cmi : %.mli
	ocamlc -c $<

microc.tar.gz : $(TARFILES)
	cd .. && tar czf microc/microc.tar.gz $(TARFILES:%=microc/%)

.PHONY : clean
clean :
	rm -f microc parser.ml parser.mli scanner.ml testall.log *.cmo *.cmi

# Generated by ocamldep *.ml *.mli
interpret.cmo: ast.cmi
interpret.cmx: ast.cmi
microc.cmo: scanner.cmo parser.cmi interpret.cmo
microc.cmx: scanner.cmx parser.cmx interpret.cmx
parser.cmo: ast.cmi parser.cmi
parser.cmx: ast.cmi parser.cmi
printer.cmo: ast.cmi
printer.cmx: ast.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmi
```

References

Language Proposal For EasyDraw Programming Language
MicroC programming language