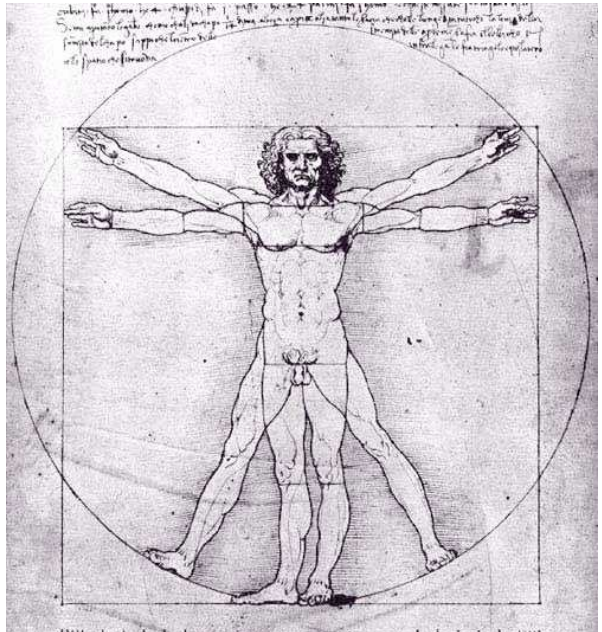
The Anatomy of Two Small Interpreters

Stephen A. Edwards

Columbia University

Fall 2008



Part I

MicroC

The MicroC Language

Interpreter for a very stripped-down version of C

Functions, global variables, and most expressions and statements, but only integer variables.

```
/* The GCD algorithm in MicroC */

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

main()
{
  print(gcd(2,14));
  print(gcd(3,15));
  print(gcd(99,121));
}
```

The Scanner (scanner.mll)

```
{ 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 *)
```

The AST (ast.mli)

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

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 *)
```

The Parser (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
%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) }

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 }
```

The Interpreter (interpret.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
  | 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"))
```

The Top-Level (microc.ml)

```
let print = false

let _ =
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  if print then
    let listing = Printer.string_of_program program in
    print_string listing
  else
    ignore (Interpret.run program)
```

Source Code Statistics

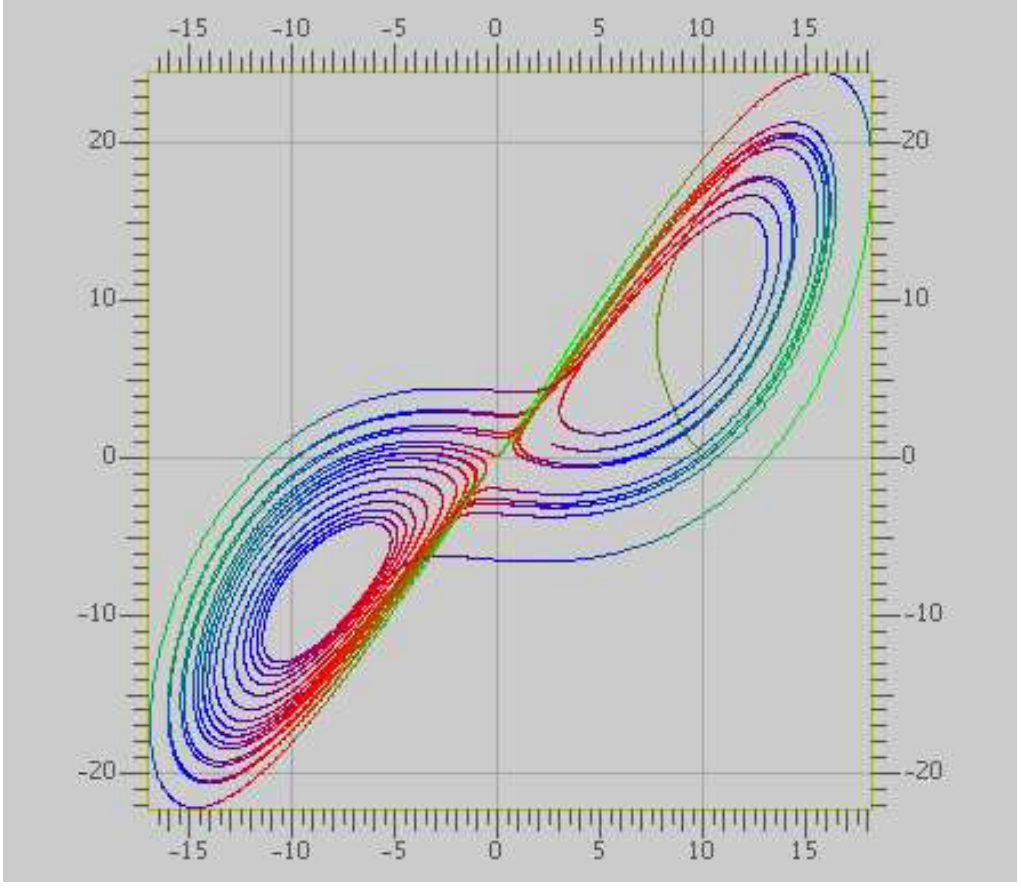
File	Lines	Role
scanner.mll	36	Token rules
parser.mly	93	Context-free grammar
ast.mli	26	Abstract syntax tree type
interpret.ml	122	AST interpreter
microc.ml	10	Top-level
Total	287	

Test Case Statistics

File	Lines	File	Lines	Role
test-arith1.mc	4	test-arith1.out	1	basic arithmetic
test-arith2.mc	4	test-arith2.out	1	precedence, associativity
test-fib.mc	15	test-fib.out	6	recursion
test-for1.mc	8	test-for1.out	6	for loop
test-func1.mc	11	test-func1.out	1	user-defined function
test-gcd.mc	14	test-gcd.out	3	greatest common divisor
test-global1.mc	29	test-global1.out	4	global variables
test-hello.mc	6	test-hello.out	3	printing
test-if1.mc	5	test-if1.out	2	if statements
test-if2.mc	5	test-if2.out	2	else
test-if3.mc	5	test-if3.out	1	false predicate
test-if4.mc	5	test-if4.out	2	false else
test-ops1.mc	27	test-ops1.out	24	all binary operators
test-var1.mc	6	test-var1.out	1	local variables
test-while1.mc	10	test-while1.out	6	while loop
Total	154		63	

Part II

The Mx Language



Mx

A Programming Langauge for Scientific Computation

Resembles Matlab, Octave, Mathematica, etc.

Project from Spring 2003

Authors:

Tiantian Zhou

Hanhua Feng

Yong Man Ra

Chang Woo Lee

Example

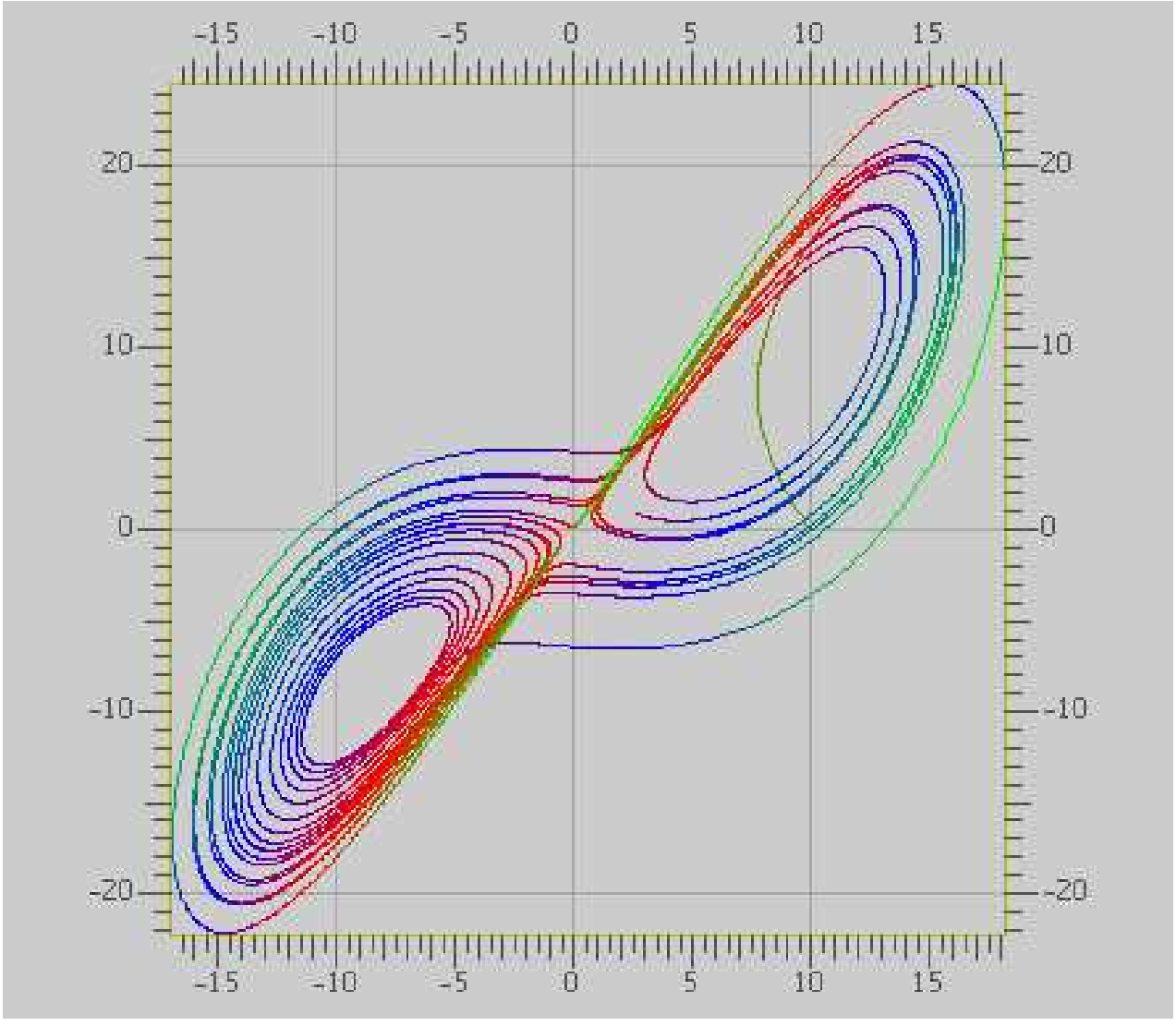
Plotting the Lorenz equations

$$\begin{aligned}
\frac{dy_0}{dt} &= \alpha(y_1 - y_0) \\
\frac{dy_1}{dt} &= y_0(r - y_2) - y_1 \\
\frac{dy_2}{dt} &= y_0 y_1 - b y_2
\end{aligned}$$

Mx source for Lorentz

```
/* Lorenz equation parameters */
a = 10;  b = 8/3.0;  r = 28;
/* Two-argument function returning a vector (body an expression) */
func Lorenz ( y, t ) =
 [ a*(y[1]-y[0]); -y[0]*y[2] + r*y[0] - y[1]; y[0]*y[1] - b*y[2] ];
/* Runge-Kutta numerical integration procedure (body is statements) */
func RungeKutta( f, y, t, h ) {
    k1 = h * f( y, t );
    k2 = h * f( y+0.5*k1, t+0.5*h );
    k3 = h * f( y+0.5*k2, t+0.5*h );
    k4 = h * f( y+k3, t+h );
    return y + (k1+k4)/6.0 + (k2+k3)/3.0;
}
/* Parameters for the procedure */
N = 20000;  p = zeros(N+1,3);  t = 0.0;  h = 0.001;
x = [ 10; 0; 10 ];  p[0,:] = x'; /* matrix transpose */
/* Perform the integration */
for ( i = 1:N ) {
    x = RungeKutta( Lorenz, x, t, h );
    p[i,:] = x'; t += h;
}
/* Plot the results */
colormap(3); plot(p); return 0;
```

Result



Architecture

Standard interpreter structure:

- Scanner identifies tokens
- Parser builds an AST
- Interpreter walks the AST
  - A Java class for each type (int, Boolean, matrix, etc.)
  - Classes have methods such as “equals” and “plus”
  - Dynamic symbol table holds variables and their values
  - Interpreter locates objects for variables, calls appropriate methods

file	lines	role
Scanner and Parser: Builds the tree		
grammar.g	314	Lexer/Parser (ANTLR source)
Interpreter: Walks the tree, invokes objects' methods		
walker.g	170	Tree Walker (ANTLR source)
MxInterpreter.java	359	Function invocation, etc.
MxSymbolTable.java	109	Name-to-object mapping
Top-level: Invokes the interpreter		
MxMain.java	153	Command-line interface
MxException.java	13	Error reporting
Runtime system: Represents data, performs operations		
MxDataType.java	169	Base class
MxBool.java	63	Booleans
MxInt.java	152	Integers
MxDouble.java	142	Floating-point
MxString.java	47	String
MxVariable.java	26	Undefined variable
MxFunction.java	81	User-defined functions
MxInternalFunction.m4	410	sin, cos, etc. (macro processed)
jamaica/Matrix.java	1387	Matrices
MxMatrix.java	354	Wrapper
jamaica/Range.java	163	e.g., 1:10
MxRange.java	67	Wrapper
jamaica/BitArray.java	226	Matrix masks
MxBitArray.java	47	Wrapper
jamaica/Painter.java	339	Bitmaps
jamaica/Plotter.java	580	2-D plotting
total	5371	