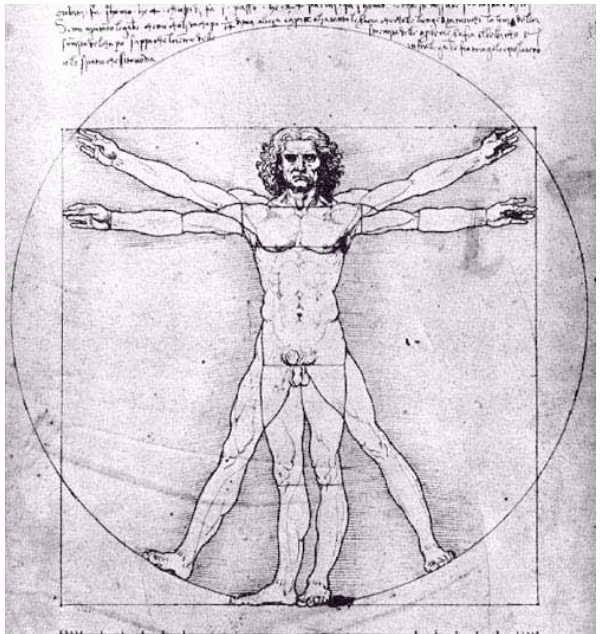
The MicroC Compiler

Stephen A. Edwards

Columbia University

Fall 2010



The MicroC Language

A very stripped-down dialect 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.ml)

```
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
%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 }
```

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 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"))
```

Bytecode

```
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 non-zero *)
  | Bra of int     (* Branch relative *)
  | Hlt            (* Terminate *)

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

Bytecode in Action

```
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));
}
```

```
 0 Jsr 2  #main()
 1 Hlt

 2 Ent 0  #main() func
 3 Lit 14
 4 Lit 2
 5 Jsr 20 #gcd(2,14)
 6 Jsr -1 #print()
 7 Drp

 8 Lit 15
 9 Lit 3
10 Jsr 20 #gcd(3,15)
11 Jsr -1 #print()
12 Drp

13 Lit 121
14 Lit 99
15 Jsr 20 #gcd(99,121)
16 Jsr -1 #print()
17 Drp

18 Lit 0
19 Rts 0
```

```
20 Ent 0   # gcd() func
21 Bra 16  # goto 37

22 Lfp -2  # a > b?
23 Lfp -3
24 Gt
25 Beq 7   # else 32

26 Lfp -2  # a = a - b
27 Lfp -3
28 Sub
29 Sfp -2
30 Drp
31 Bra 6   # goto 37

32 Lfp -3  # b = b - a
33 Lfp -2
34 Sub
35 Sfp -3
36 Drp

37 Lfp -2  # a != b?
38 Lfp -3
39 Neq
40 Bne -18 # 22

41 Lfp -2  # return a
42 Rts 2
43 Lit 0
44 Rts 2
```

The Compiler (compile.ml)

```
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 : int -> 'a list -> (int * 'a) list *)
let rec enum stride n = function
    [] -> []
  | hd::tl -> (n, hd) :: enum stride (n+stride) tl

(* string_map_pairs:StringMap 'a -> (int * 'a) list -> StringMap 'a *)
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" for each global variable *)
  let global_indexes =
    string_map_pairs StringMap.empty (enum 1 0 globals) in

  (* Assign indexes to function names; built-in "print" is special *)
  let built_in_functions =
    StringMap.add "print" (-1) StringMap.empty 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
    and formal_offsets = enum (-1) (-2) fdecl.formals in
    let env = { env with local_index = string_map_pairs
                  StringMap.empty (local_offsets @ formal_offsets) } in
```

```
(* Translate an expression *)
let rec expr = function
    Literal i -> [Lit i]

  | Id s ->
      (try [Lfp (StringMap.find s env.local_index)]
      with Not_found -> try
        [Lod (StringMap.find s env.global_index)]
      with Not_found ->
        raise (Failure ("undeclared variable " ^ s)))

  | Binop (e1, op, e2) -> expr e1 @ expr e2 @ [Bin op]

  | Assign (s, e) -> expr e @
      (try [Sfp (StringMap.find s env.local_index)]
      with Not_found -> try
        [Str (StringMap.find s env.global_index)]
      with Not_found ->
        raise (Failure ("undeclared variable " ^ s)))

  | Call (fname, actuals) -> (try
      (List.concat (List.map expr (List.rev actuals))) @
      [Jsr (StringMap.find fname env.function_index) ]
    with Not_found ->
      raise (Failure ("undefined function " ^ fname)))

  | Noexpr -> []
```

```
  (* Translate a statement *)
  in let rec stmt = function
      Block sl      -> List.concat (List.map stmt sl)

    | Expr e        -> expr e @ [Drp] (* Discard result *)

    | Return e      -> expr e @ [Rts num_formals]

    | If (p, t, f) -> let t' = stmt t and f' = stmt f in
      expr p @ [Beq(2 + List.length t')] @
      t' @ [Bra(1 + List.length f')] @ f'

    | For (e1, e2, e3, b) -> (* Rewrite into a while statement *)
        stmt (Block([Expr(e1); While(e2, Block([b; Expr(e3)]))]))

    | While (e, b) ->
        let b' = stmt b and e' = expr e in
        [Bra (1+ List.length b')] @ b' @ e' @
        [Bne (-(List.length b' + List.length e'))]

  (* Translate a whole function *)
  in [Ent num_locals] @       (* Entry: allocate space for locals *)
  stmt (Block fdecl.body) @   (* Body *)
  [Lit 0; Rts num_formals]    (* Default = return 0 *)
```

```
in 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
    [Jsr (StringMap.find "main" function_indexes); Hlt]
  with Not_found -> raise (Failure ("no \"main\" function"))
in

(* Compile the functions *)
let func_bodies = entry_function ::
                  List.map (translate env) functions in

(* Calculate function entry points by adding their lengths *)
let (fun_offset_list, _) = List.fold_left
    (fun (l,i) f -> (i :: l, (i + List.length f))) ([],0)
    func_bodies in
let func_offset = Array.of_list (List.rev fun_offset_list) in

{ num_globals = List.length globals;
  (* Concatenate the compiled functions and replace the function
     indexes in Jsr statements with PC values *)
  text = Array.of_list (List.map (function
      Jsr i when i > 0 -> Jsr func_offset.(i)
    | _ as s -> s) (List.concat func_bodies))
}
```

The Bytecode Interpreter (execute.ml)

```
open Ast
open Bytecode

let execute_prog prog =
  let stack = Array.make 1024 0
  and globals = Array.make prog.num_globals 0 in

  let rec exec fp sp pc = match prog.text.(pc) with
    Lit i  -> stack.(sp) <- i ; exec fp (sp+1) (pc+1)
  | Drp    -> exec fp (sp-1) (pc+1)
  | Bin op -> 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)
```

The Bytecode Interpreter (execute.ml)

```
  | Lod i    -> stack.(sp)   <- globals.(i)  ; exec fp (sp+1) (pc+1)
  | Str i    -> globals.(i)  <- stack.(sp-1) ; exec fp sp     (pc+1)
  | Lfp i    -> stack.(sp)   <- stack.(fp+i) ; exec fp (sp+1) (pc+1)
  | Sfp i    -> stack.(fp+i) <- stack.(sp-1) ; exec fp sp     (pc+1)
  | Jsr(-1) -> print_endline (string_of_int stack.(sp-1)) ;
               exec fp sp (pc+1)
  | Jsr i    -> stack.(sp)   <- pc + 1       ; exec fp (sp+1) i
  | Ent i    -> stack.(sp)   <- fp           ; exec sp (sp+i+1) (pc+1)
  | Rts 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    -> exec fp (sp-1)
                (pc + if stack.(sp-1) =  0 then i else 1)
  | Bne i    -> exec fp (sp-1)
                (pc + if stack.(sp-1) != 0 then i else 1)
  | Bra i    -> exec fp sp (pc+i)
  | Hlt      -> ()

  in exec 0 0 0
```

The Top Level (microc.ml)

```
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)
```

Source Code Statistics

File	Lines	Role
scanner.mll	36	Token rules
parser.mly	93	Context-free grammar
ast.ml	66	Abstract syntax tree type and pretty printer
interpret.ml	123	AST interpreter
bytecode.ml	51	Bytecode type and pretty printer
compile.ml	104	AST-to-bytecode compiler
execute.ml	51	Bytecode interpreter
microc.ml	20	Top-level
Total	544	

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-func2.mc	18	test-func2.out	1	argument eval. order
test-func3.mc	12	test-func3.out	4	argument eval. order
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	184		68	