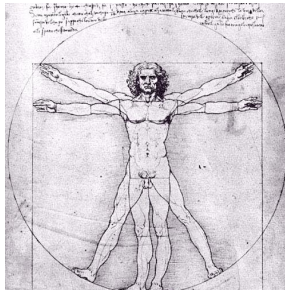


The MicroC Compiler

Stephen A. Edwards

Columbia University

Spring 2016



The Front-End

Static Semantic Checking

Code Generation

The Top-Level

The MicroC Language

A very stripped-down dialect of C

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

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

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

```
int main()  
{  
    print(gcd(2,14));  
    print(gcd(3,15));  
    print(gcd(99,121));  
    return 0;  
}
```

The Front-End

Tokenize and parse to produce
an Abstract Syntax Tree

The first part of any compiler or interpreter

The Scanner (scanner.mll)

```
{ open Parser }

rule token = parse
  [ ' ' '\t' '\r' '\n' ] { token lexbuf } (* Whitespace *)
| "/*" { comment lexbuf } (* Comments *)
| '(' { LPAREN } | '=' { ASSIGN } | "if" { IF }
| ')' { RPAREN } | "==" { EQ } | "else" { ELSE }
| '{' { LBRACE } | "!=" { NEQ } | "for" { FOR }
| '}' { RBRACE } | '<' { LT } | "while" { WHILE }
| ';' { SEMI } | "<=" { LEQ } | "return" { RETURN }
| ',' { COMMA } | ">" { GT } | "int" { INT }
| '+' { PLUS } | ">=" { GEQ } | "bool" { BOOL }
| '-' { MINUS } | "&&" { AND } | "void" { VOID }
| '*' { TIMES } | "||" { OR } | "true" { TRUE }
| '/' { DIVIDE } | "!" { NOT } | "false" { FALSE }
| ['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 }
```

The AST (ast.ml)

```
type op = Add | Sub | Mult | Div | Equal | Neg |  
        Less | Leq | Greater | Geq | And | Or  
type uop = Neg | Not  
type typ = Int | Bool | Void  
type bind = typ * string  
  
type expr = Literal of int | BoolLit of bool  
          | Id of string | Noexpr  
          | Binop of expr * op * expr | Unop of uop * expr  
          | Assign of string * expr | Call of string * expr list  
  
type stmt = Block of stmt list | Expr of expr  
          | If of expr * stmt * stmt  
          | For of expr * expr * expr * stmt  
          | While of expr * stmt | Return of expr  
  
type func_decl = {  
    typ      : typ;  
    fname    : string;  
    formals  : bind list;  
    locals   : bind list;  
    body     : stmt list;  
}  
  
type program = bind list * func_decl list
```

The Parser (parser.mly)

```
%{ open Ast %}
```

```
%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
```

```
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT
```

```
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
```

```
%token RETURN IF ELSE FOR WHILE INT BOOL VOID
```

```
%token <int> LITERAL
```

```
%token <string> ID
```

```
%token EOF
```

```
%nonassoc NOELSE
```

```
%nonassoc ELSE
```

```
%right ASSIGN
```

```
%left OR
```

```
%left AND
```

```
%left EQ NEQ
```

```
%left LT GT LEQ GEQ
```

```
%left PLUS MINUS
```

```
%left TIMES DIVIDE
```

```
%right NOT NEG
```

```
%start program
```

```
%type <Ast.program> program
```

```
%%
```

Declarations

```
program:  decls EOF { $1 }

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

fdecl:
  typ ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
  { { typ = $1; fname = $2; formals = $4;
    locals = List.rev $7; body = List.rev $8 } }

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

formal_list:  typ ID { [($1,$2)] }
            | formal_list COMMA typ ID { ($3,$4) :: $1 }

typ:  INT { Int }
     | BOOL { Bool }
     | VOID { Void }

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

vdecl:  typ ID SEMI { ($1, $2) }
```


Statements

stmt_list:

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

stmt:

```
    expr SEMI { Expr $1 }  
| RETURN SEMI { Return Noexpr }  
| 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 SEMI expr_opt RPAREN stmt  
    { For($3, $5, $7, $9) }  
| WHILE LPAREN expr RPAREN stmt { While($3, $5) }
```

Expressions

expr:

<i>LITERAL</i>	{ Literal(\$1) }
<i>TRUE</i>	{ BoolLit(true) }
<i>FALSE</i>	{ BoolLit(false) }
<i>ID</i>	{ Id(\$1) }
<i>expr PLUS expr</i>	{ Binop(\$1, Add, \$3) }
<i>expr MINUS expr</i>	{ Binop(\$1, Sub, \$3) }
<i>expr TIMES expr</i>	{ Binop(\$1, Mult, \$3) }
<i>expr DIVIDE expr</i>	{ Binop(\$1, Div, \$3) }
<i>expr EQ expr</i>	{ Binop(\$1, Equal, \$3) }
<i>expr NEQ expr</i>	{ Binop(\$1, Neq, \$3) }
<i>expr LT expr</i>	{ Binop(\$1, Less, \$3) }
<i>expr LEQ expr</i>	{ Binop(\$1, Leq, \$3) }
<i>expr GT expr</i>	{ Binop(\$1, Greater, \$3) }
<i>expr GEQ expr</i>	{ Binop(\$1, Geq, \$3) }
<i>expr AND expr</i>	{ Binop(\$1, And, \$3) }
<i>expr OR expr</i>	{ Binop(\$1, Or, \$3) }
<i>MINUS expr %prec NEG</i>	{ Unop(Neg, \$2) }
<i>NOT expr</i>	{ Unop(Not, \$2) }
<i>ID ASSIGN expr</i>	{ Assign(\$1, \$3) }
<i>LPAREN expr RPAREN</i>	{ \$2 }
<i>ID LPAREN actuals_opt RPAREN</i>	{ Call(\$1, \$3) }

Expressions concluded

expr_opt:

```
    /* nothing */ { Noexpr }  
  | expr          { $1 }
```

actuals_opt:

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

actuals_list:

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

Testing with menhir

```
$ menhir --interpret --interpret-show-cst parser.mly
INT ID LPAREN RPAREN LBRACE ID LPAREN LITERAL RPAREN SEMI RBRACE EOF
ACCEPT
```

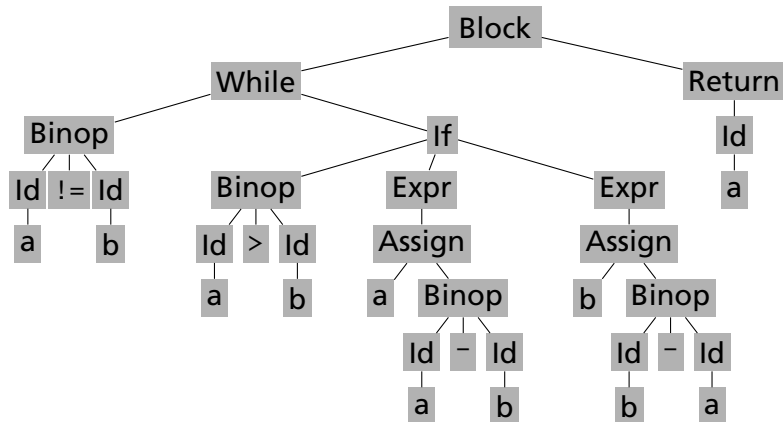
```
int main() {
  print(42);
}
```

```
[program:
  [decls:
    [decls:]
    [fdecl:
      [typ: INT]
      ID
      LPAREN
      [formals_opt:]
      RPAREN
      LBRACE
      [vdecl_list:]
      [stmt_list:
        [stmt_list:]
        [stmt:
          [expr:
            ID
            LPAREN
            [actuals_opt: [actuals_list: [expr: LITERAL]]]
            RPAREN
          ]
          SEMI
        ]
      ]
    ]
  ]
  RBRACE
]
]
EOF
]
```

AST for the GCD Example

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

typ = Int
fname = gcd
formals = [Int a; Int b]
locals = []
body =



AST for the GCD Example

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

```
typ = Int  
fname = gcd  
formals = [Int a; Int b]  
locals = []  
body =
```

```
[While (Binop (Id a) Neq (Id b))  
  (Block [(If (Binop (Id a) Greater (Id b))  
    (Expr (Assign a  
      (Binop (Id a) Sub (Id b))))  
    (Expr (Assign b  
      (Binop (Id b) Sub (Id a))))  
  ]),  
Return (Id a)]
```

Static Semantic Checking

- Walk over the AST

- Verify each node

- Establish existence of each identifier

- Establish type of each expression

- Validate statements in functions

The Semantic Checker (semant.ml)

```
open Ast

module StringMap = Map.Make(String)

(* Semantic checking of a program. Returns void if successful,
   throws an exception if something is wrong.
   Check each global variable, then check each function *)

let check (globals, functions) =
```

Used to check for identically named globals, functions, formal arguments, and local variables

```
(* Raise an exception if the given list has a duplicate *)
let report_duplicate exceptf list =
  let rec helper = function
    | n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
    | _ :: t -> helper t
    | [] -> ()
  in helper (List.sort compare list)
in
```


Helper functions

Used to check for *void* globals, formal arguments, and locals

```
(* Raise an exception if a given binding is to a void type *)  
let check_not_void exceptf = function  
    (Void, n) -> raise (Failure (exceptf n))  
  | _ -> ()  
in
```

In the assignment *lvalue = rvalue*,
can the type of *rvalue* be assigned to *lvalue*?

Used to test assignments and function calls

```
(* Raise an exception of the given rvalue type cannot be assigned  
   to the given lvalue type or return the type of the assignment *)  
let check_assign lvaluet rvaluet err =  
    if lvaluet == rvaluet then lvaluet else raise err  
in
```

Global Variables, Function Names

(** Checking Global Variables ****)**

```
List.iter (check_not_void (fun n -> "illegal void global " ^ n))  
          globals;  
  
report_duplicate (fun n -> "duplicate global " ^ n)  
                 (List.map snd globals);
```

(** Checking Functions ****)**

```
if List.mem "print" (List.map (fun fd -> fd.fname) functions)  
then raise (Failure ("function print may not be defined")) else ();  
  
report_duplicate (fun n -> "duplicate function " ^ n)  
                 (List.map (fun fd -> fd.fname) functions);
```

Function Symbol Table

```
(* Function declaration for a named function *)
```

```
let built_in_decls = StringMap.add "print"  
  { typ = Void; fname = "print"; formals = [(Int, "x")];  
    locals = []; body = [] } (StringMap.singleton "printb"  
  { typ = Void; fname = "printb"; formals = [(Bool, "x")];  
    locals = []; body = [] })  
in
```

MicroC has two built-in functions, *print* and *printb*; this is an easy way to implement it. Your compiler should have very few exceptions like this.

```
let function_decls =  
  List.fold_left (fun m fd -> StringMap.add fd.fname fd m)  
    built_in_decls functions  
in  
  
let function_decl s = try StringMap.find s function_decls  
  with Not_found -> raise (Failure ("unrecognized function " ^ s))  
in  
  
(* Ensure "main" is defined *)  
let _ = function_decl "main" in
```

Check a Function

```
let check_function func =  
  
  List.iter (check_not_void (fun n ->  
    "illegal void formal " ^ n ^ " in " ^ func.fname))  
    func.formals;  
  
  report_duplicate (fun n ->  
    "duplicate formal " ^ n ^ " in " ^ func.fname)  
    (List.map snd func.formals);  
  
  List.iter (check_not_void (fun n ->  
    "illegal void local " ^ n ^ " in " ^ func.fname))  
    func.locals;  
  
  report_duplicate (fun n ->  
    "duplicate local " ^ n ^ " in " ^ func.fname)  
    (List.map snd func.locals);
```

Variable Symbol Table

What can happen when you refer to a variable?

What are MicroC's *scoping rules*?

```
int a;      /* Global variable */
int c;

void foo(int a) { /* Formal arg. */
  int b; /* Local variable */
  ... a = ... a ...
  ... b = ... b ...
  ... c = ... c ...
  ... d = ... d ...
}
```

```
(* Variable symbol table: type of each global, formal, local *)
let symbols = List.fold_left
  (fun m (t, n) -> StringMap.add n t m)
  StringMap.empty
  ( globals @ func.formals @ func.locals )

in

let type_of_identifier s =
  try StringMap.find s symbols
  with Not_found ->
    raise (Failure ("undeclared identifier " ^ s))

in
```

Expressions

The key semantic-checking operation: establish the type of each subexpression.

```
(* Return the type of an expression or throw an exception *)  
let rec expr = function  
  Literal _ -> Int  
  | BoolLit _ -> Bool  
  | Noexpr -> Void
```

An identifier: does it exist? What is its type?

```
| Id s -> type_of_identifier s
```

Assignment: need to know the types of the *lvalue* and *rvalue*, and whether one can be assigned to the other.

```
| Assign(var, e) as ex -> let lt = type_of_identifier var  
                           and rt = expr e in  
  check_assign (type_of_identifier var) (expr e)  
  (Failure ("illegal assignment " ^ string_of_typ lt ^  
    " = " ^ string_of_typ rt ^ " in " ^ string_of_expr ex))
```

Operators

What types can be added? Subtracted? Compared for equality? Negated?

```
| Binop(e1, op, e2) as e -> let t1 = expr e1
                             and t2 = expr e2 in
  (match op with
    Add | Sub | Mult | Div      when t1 = Int && t2 = Int    -> Int
  | Equal | Neq                when t1 = t2                  -> Bool
  | Less | Leq | Greater | Geq when t1 = Int && t2 = Int    -> Bool
  | And | Or                   when t1 = Bool && t2 = Bool  -> Bool
  | _ -> raise (Failure ("illegal binary operator " ^
                        string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
                        string_of_typ t2 ^ " in " ^ string_of_expr e))
  )

| Unop(op, e) as ex -> let t = expr e in
  (match op with
    Neg when t = Int -> Int
  | Not when t = Bool -> Bool
  | _ -> raise (Failure ("illegal unary operator " ^
                        string_of_uop op ^ string_of_typ t ^
                        " in " ^ string_of_expr ex)))
```

Function Calls

Number and type of formals and actuals must match

```
void foo(t1 f1, t2 f2) { ... }
```

```
... = ... foo(expr1, expr2) ...
```

```
f1 = expr1;  
f2 = expr2;
```

The callsite behaves like

```
| Call(fname, actuals) as call -> let fd = function_decl fname in  
  if List.length actuals != List.length fd.formals then  
    raise (Failure ("expecting " ^ string_of_int  
      (List.length fd.formals) ^ " arguments in " ^  
      string_of_expr call))  
  else  
    List.iter2 (fun (ft, _) e -> let et = expr e in  
      ignore (check_assign ft et  
        (Failure ("illegal actual argument found " ^  
          string_of_typ et ^ " expected " ^  
          string_of_typ ft ^ " in " ^ string_of_expr e))))  
      fd.formals actuals;  
    fd.typ (* Finally, the call returns the function's type *)
```

in

Statements

Make sure an expression is Boolean: used in *if*, *for*, *while*.

```
let check_bool_expr e = if expr e != Bool
  then raise (Failure ("expected Boolean expression in " ^
                        string_of_expr e))
  else () in
```

Checking a statement: make sure it is well-formed or throw an exception

```
(* Verify a statement or throw an exception *)
let rec stmt = function
  Expr e -> ignore (expr e)

  | If(p, b1, b2) -> check_bool_expr p; stmt b1; stmt b2

  | For(e1, e2, e3, st) -> ignore (expr e1); check_bool_expr e2;
                        ignore (expr e3); stmt st

  | While(p, s) -> check_bool_expr p; stmt s
```

Statements: Return

The type of the argument to *return* must match the type of the function.

```
| Return e ->  
  let t = expr e in  
  if t = func.typ then ()  
  else raise (Failure ("return gives " ^ string_of_typ t ^  
    " expected " ^ string_of_typ func.typ ^ " in " ^  
    string_of_expr e))
```

Statements: Blocks

Checking a block of statements is almost

List.iter stmt sl, but LLVM does not like code after a return:

```
int main() {  
    return 1;  
    print(42); /* Illegal: code after a return */  
}
```

```
| Block sl -> let rec check_block = function  
    [Return _ as s] -> stmt s  
    | Return _ :: _ ->  
        raise (Failure "nothing may follow a return")  
    | Block sl :: ss -> check_block (sl @ ss)  
    | s :: ss -> stmt s ; check_block ss  
    | [] -> ()  
in check_block sl
```

Bodies of `check_function` and `check`

check_function: check the statements in the body

```
in stmt (Block func.body)  (* body of check_function *)
```

check: check each function in the program

```
in List.iter check_function functions (* body of check *)
```

Code Generation

Assumes AST is semantically correct

Translate each AST node into LLVM IR

Construct expressions bottom-up

Construct basic blocks for control-flow statements

<http://llvm.org>

<http://llvm.org/docs/tutorial>

<http://llvm.moe> Ocaml bindings documentation

The LLVM IR

Assembly-language like: list of simple operations

Static Single-Assignment: each value (e.g., %x3) assigned exactly once

```
int add(int x, int y)
{
    return x + y;
}
```

```
define i32 @add(i32 %x, i32 %y) {
entry:
    %x1 = alloca i32
    store i32 %x, i32* %x1
    %y2 = alloca i32
    store i32 %y, i32* %y2
    %x3 = load i32* %x1
    %y4 = load i32* %y2
    %tmp = add i32 %x3, %y4
    ret i32 %tmp
}
```

i32: 32-bit signed integer type

alloca: Allocate space on the stack; return a pointer

store: Write a value to an address

load: Read a value from an address

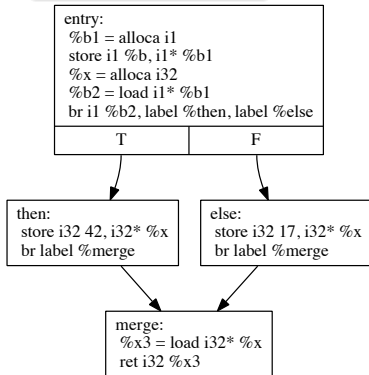
add: Add two values to produce a third

ret: Return a value to the caller

Basic Blocks

Each function consists of *basic blocks*: instruction sequence with one entry point & control-transfer at end

```
int cond(bool b) {  
    int x;  
    if (b) x = 42;  
    else   x = 17;  
    return x;  
}
```



CFG for 'cond' function

```
define i32 @cond(i1 %b) {  
entry:  
    %b1 = alloca i1  
    store i1 %b, i1* %b1  
    %x = alloca i32  
    %b2 = load i1* %b1  
    br i1 %b2, label %then, label %else
```

```
merge:        ; preds = %else, %then  
    %x3 = load i32* %x  
    ret i32 %x3
```

```
then:         ; preds = %entry  
    store i32 42, i32* %x  
    br label %merge
```

```
else:         ; preds = %entry  
    store i32 17, i32* %x  
    br label %merge  
}
```

```

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

```

```

define i32 @gcd(i32 %a, i32 %b) {
entry:
    %a1 = alloca i32
    store i32 %a, i32* %a1
    %b2 = alloca i32
    store i32 %b, i32* %b2
    br label %while

while:                                     ; preds = %merge, %entry
    %a11 = load i32* %a1
    %b12 = load i32* %b2
    %tmp13 = icmp ne i32 %a11, %b12
    br i1 %tmp13, label %while_body, label %merge14

while_body:                               ; preds = %while
    %a3 = load i32* %a1
    %b4 = load i32* %b2
    %tmp = icmp sgt i32 %a3, %b4
    br i1 %tmp, label %then, label %else

merge:                                    ; preds = %else, %then
    br label %while

then:                                    ; preds = %while_body
    %a5 = load i32* %a1
    %b6 = load i32* %b2
    %tmp7 = sub i32 %a5, %b6
    store i32 %tmp7, i32* %a1
    br label %merge

else:                                    ; preds = %while_body
    %b8 = load i32* %b2
    %a9 = load i32* %a1
    %tmp10 = sub i32 %b8, %a9
    store i32 %tmp10, i32* %b2
    br label %merge

merge14:                                 ; preds = %while
    %a15 = load i32* %a1
    ret i32 %a15
}

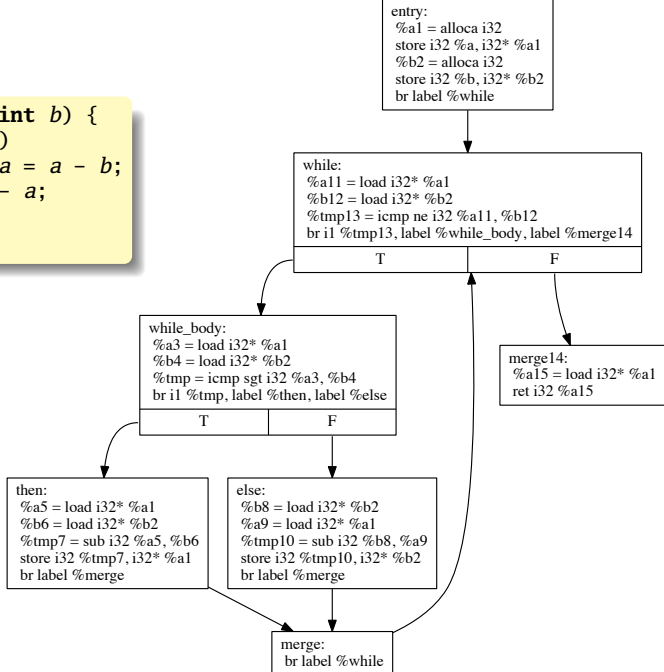
```



```

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

```



CFG for 'gcd' function

The Code Generator (codegen.ml)

Translate takes a semantically checked AST and returns an LLVM module:

```
module L = Llvm    (* LLVM VMCore interface library *)
module A = Ast      (* MicroC Abstract Syntax Tree *)

module StringMap = Map.Make(String)

let translate (globals, functions) =
  let context = L.global_context () in (* global data container *)
  let the_module = L.create_module context "MicroC" (* container *)

  and i32_t = L.i32_type context      (* int *)
  and i8_t  = L.i8_type  context      (* for printf format string *)
  and i1_t  = L.i1_type  context      (* bool *)
  and void_t = L.void_type context in (* void *)

  let ltype_of_typ = function (* LLVM type for AST type *)
    | A.Int   -> i32_t
    | A.Bool  -> i1_t
    | A.Void  -> void_t in
```

Define Global Variables

```
int i;  
bool b;  
int k;  
  
int main()  
{  
    i = 42;  
    k = 10;
```

```
@k = global i32 0  
@b = global i1 false  
@i = global i32 0  
  
define i32 @main() {  
entry:  
    store i32 42, i32* @i  
    store i32 10, i32* @k
```

```
(* Initialize each global variable; remember them in a map *)  
let global_vars =  
    let global_var m (t, n) =  
        let init = L.const_int (ltype_of_typ t) 0  
        in StringMap.add n (L.define_global n init the_module) m in  
    List.fold_left global_var StringMap.empty globals in
```

Declare external function *printf*

Declare *printf*, which we'll use to implement *print* and *printb*.

```
let printf_t =  
  L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in  
let printf_func =  
  L.declare_function "printf" printf_t the_module in
```

```
declare i32 @printf(i8*, ...)
```

Define function prototypes

```
void foo() ...
```

```
int bar(int a, bool b, int c) ...
```

```
int main() ...
```

```
define void @foo() ...
```

```
define i32 @bar(i32 %a, i1 %b, i32 %c)
```

```
define i32 @main() ...
```

Build a map from function name to (LLVM function, *fdecl*)

Construct the declarations first so we can call them when we build their bodies.

```
(* Define each function (arguments and return type) *)
```

```
let function_decls =
```

```
  let function_decl m fdecl =
```

```
    let name = fdecl.A.fname
```

```
    and formal_types = Array.of_list
```

```
      (List.map (fun (t,_) -> ltype_of_typ t) fdecl.A.formals)
```

```
  in let ftype =
```

```
    L.function_type (ltype_of_typ fdecl.A.typ) formal_types in
```

```
    StringMap.add name (L.define_function name ftype the_module,  
      fdecl) m in
```

```
  List.fold_left function_decl StringMap.empty functions in
```

build_function_body

An “Instruction Builder” is the LLVM library’s object that controls where the next instruction will be inserted.

It points to some instruction in some basic block.

This is an unfortunate artifact of LLVM being written in C++.

```
(* Fill in the body of the given function *)
```

```
let build_function_body fdecl =
```

```
    let (the_function, _) =
```

```
        StringMap.find fdecl.A.fname function_decls in
```

```
    let builder =          (* Create an instruction builder *)
```

```
        L.builder_at_end context (L.entry_block the_function) in
```

```
    let int_format_str =    (* Format string for printf calls *)
```

```
        L.build_global_stringptr "%d\n" "fmt" builder in
```

```
@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00"
```

Formals and Locals

Allocate formal arguments and local variables on the stack;
remember names in *local_vars* map

```
int foo(int a, bool b)
{
  int c;
  bool d;
```

```
define i32 @foo(i32 %a, i1 %b) {
entry:
  %a1 = alloca i32
  store i32 %a, i32* %a1
  %b2 = alloca i1
  store i1 %b, i1* %b2
  %c = alloca i32
  %d = alloca i1
```

```
let local_vars =
  let add_formal m (t, n) p = L.set_value_name n p;
    let local = L.build_alloca (ltype_of_typ t) n builder in
    ignore (L.build_store p local builder);
    StringMap.add n local m in

  let add_local m (t, n) =
    let local_var = L.build_alloca (ltype_of_typ t) n builder
    in StringMap.add n local_var m in

  let formals = List.fold_left2 add_formal StringMap.empty
    fdecl.A.formals (Array.to_list (L.params the_function)) in
  List.fold_left add_local formals fdecl.A.locals in
```

lookup

Look for a variable among the locals/formal arguments, then the globals. Semantic checking ensures one of the two is always found.

Used for both identifiers and assignments.

```
(* Return the value for a variable or formal argument *)  
let lookup n = try StringMap.find n local_vars  
               with Not_found -> StringMap.find n global_vars  
in
```


Expressions

The main expression function: build instructions in the given builder that evaluate an expression; return the expression's value

```
let rec expr builder = function
  | A.Literal i -> L.const_int i32_t i
  | A.BoolLit b -> L.const_int i1_t (if b then 1 else 0)
  | A.Noexpr -> L.const_int i32_t 0
  | A.Id s -> L.build_load (lookup s) s builder
  | A.Assign (s, e) -> let e' = expr builder e in
                       ignore (L.build_store e' (lookup s) builder); e'
```

```
int a;

void foo(int c)
{
    a = c + 42;
}
```

```
@a = global i32 0

define void @foo(i32 %c) {
entry:
    %c1 = alloca i32
    store i32 %c, i32* %c1
    %c2 = load i32* %c1           ; read c
    %tmp = add i32 %c2, 42        ; tmp = c + 42
    store i32 %tmp, i32* @a      ; a = tmp
    ret void
}
```

Binary Operators

Evaluate left and right expressions; combine results

```
| A.Binop (e1, op, e2) ->
  let e1' = expr builder e1
  and e2' = expr builder e2 in
  (match op with
    | A.Add      -> L.build_add
    | A.Sub      -> L.build_sub
    | A.Mult     -> L.build_mul
    | A.Div      -> L.build_sdiv
    | A.And      -> L.build_and
    | A.Or       -> L.build_or
    | A.Equal    -> L.build_icmp L.Icmp.Eq
    | A.Neq      -> L.build_icmp L.Icmp.Ne
    | A.Less     -> L.build_icmp L.Icmp.Slt
    | A.Leq      -> L.build_icmp L.Icmp.Sle
    | A.Greater  -> L.build_icmp L.Icmp.Sgt
    | A.Geq      -> L.build_icmp L.Icmp.Sge
  ) e1' e2' "tmp" builder
```

neg/not/print/printb

Unary operators: evaluate subexpression and compute

```
| A.Unop(op, e) ->  
  let e' = expr builder e in  
  (match op with  
    A.Neg      -> L.build_neg  
  | A.Not      -> L.build_not) e' "tmp" builder
```

print/printb: Invoke printf("%d\n", v)

```
| A.Call ("print", [e]) | A.Call ("printb", [e]) ->  
  L.build_call printf_func  
  [| int_format_str ; (expr builder e) |]  
  "printf" builder
```

Function calls

Normal calls: evaluate the actual arguments and pass them to the call. *Do not name the result of void functions.*

```
| A.Call (f, act) ->
  let (fdef, fdecl) = StringMap.find f function_decls in
  let actuals =
    List.rev (List.map (expr builder) (List.rev act)) in
  let result = (match fdecl.A.typ with A.Void -> ""
                | _ -> f ^ "_result") in
  L.build_call fdef (Array.of_list actuals) result builder
```

```
void foo(int a)
{
    print(a + 3);
}

int main()
{
    foo(40);
    return 0;
}
```

```
define void @foo(i32 %a) {
entry:
    %a1 = alloca i32
    store i32 %a, i32* %a1
    %a2 = load i32* %a1
    %tmp = add i32 %a2, 3
    %printf = call i32 @__printf(i8*, ...) @__printf(i8* getelementptr
        @__printf, i32 0, i32 0), i32 %tmp)
    ret void
}

define i32 @main() {
entry:
    call void @foo(i32 40)
    ret i32 0
}
```

Statements

Used to add a branch instruction to a basic block only if one doesn't already exist. Used by *if* and *while*

```
let add_terminal builder f =  
  match L.block_terminator (L.insertion_block builder) with  
  | Some _ -> ()  
  | None -> ignore (f builder) in
```

The main statement function: build instructions in the given builder for the statement; return the builder for where the next instruction should be placed. *Return* has an interesting expression only in non-void functions, enforced by semantic checking.

```
let rec stmt builder = function  
  A.Block sl -> List.fold_left stmt builder sl  
  
  | A.Expr e -> ignore (expr builder e); builder  
  
  | A.Return e -> ignore (match fdecl.A.typ with  
    A.Void -> L.build_ret_void builder  
    | _ -> L.build_ret (expr builder e) builder); builder
```

If Statements

Build basic blocks for *then*, *else*, and *merge*—where the next statement will be placed.

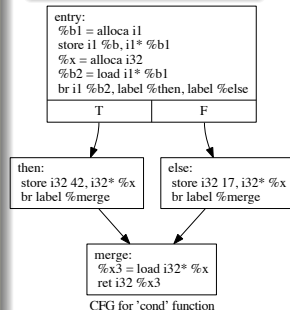
```
| A.If (predicate, then_stmt, else_stmt) ->
  let bool_val = expr_builder predicate in
  let merge_bb = L.append_block context
    "merge" the_function in

  let then_bb = L.append_block context
    "then" the_function in
  add_terminal
    (stmt (L.builder_at_end context then_bb)
      then_stmt)
  (L.build_br merge_bb);

  let else_bb = L.append_block context
    "else" the_function in
  add_terminal
    (stmt (L.builder_at_end context else_bb)
      else_stmt)
  (L.build_br merge_bb);

  ignore (L.build_cond_br bool_val
    then_bb else_bb builder);
  L.builder_at_end context merge_bb
```

```
int cond(bool b) {
  int x;
  if (b) x = 42;
  else   x = 17;
  return x;
}
```



While Statements

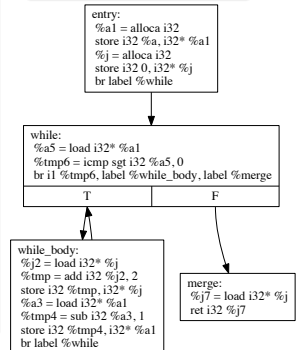
```
| A.While (predicate, body) ->
  let pred_bb = L.append_block context
    "while" the_function in
  ignore (L.build_br pred_bb builder);

  let body_bb = L.append_block context
    "while_body" the_function in
  add_terminal (stmt (L.builder_at_end
    context body_bb)
    body)
    (L.build_br pred_bb);

  let pred_builder =
    L.builder_at_end context pred_bb in
  let bool_val =
    expr pred_builder predicate in

  let merge_bb = L.append_block context
    "merge" the_function in
  ignore (L.build_cond_br bool_val
    body_bb merge_bb pred_builder);
  L.builder_at_end context merge_bb
```

```
int foo(int a)
{
  int j;
  j = 0;
  while (a > 0) {
    j = j + 2;
    a = a - 1;
  }
  return j;
}
```



CFG for 'foo' function

For Statements

```
for ( expr1 ; expr2 ; expr3 ) {  
    body;  
}
```

```
expr1;  
while ( expr2 ) {  
    body;  
    expr3;  
}
```

```
| A.For (e1, e2, e3, body) -> stmt builder  
    ( A.Block [A.Expr e1 ;  
              A.While (e2, A.Block [body ;  
                                     A.Expr e3]) ] )  
  
in
```


The End

The remainder of *build_function_body*: build the body of the function; add a *return* if control fell off the end

```
(* Build the code for each statement in the function *)  
let builder = stmt builder (A.Block fdecl.A.body) in  
  
(* Add a return if the last block falls off the end *)  
add_terminal builder (match fdecl.A.typ with  
  A.Void -> L.build_ret_void  
  | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))  
in
```

The remainder of *translate*: build the body of each function

```
List.iter build_function_body functions;  
the_module
```

The Top-Level

microc.ml

Top-level of the MicroC compiler: scan & parse, check the AST, generate LLVM IR, dump the module

```
type action = Ast | LLVM_IR | Compile
let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1)
      [ ("-a", Ast); ("-l", LLVM_IR); ("-c", Compile) ]
  else Compile in

  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in

  Semant.check ast;

  match action with
    Ast -> print_string (Ast.string_of_program ast)

  | LLVM_IR -> print_string (Llvm.string_of_llmodule
                           (Codegen.translate ast))

  | Compile -> let m = Codegen.translate ast in
    Llvm_analysis.assert_valid_module m; (* Useful built-in check *)
    print_string (Llvm.string_of_llmodule m)
```

Source Code Statistics

File	Lines	Role
scanner.mll	45	Token rules
parser.mly	115	Context-free grammar
ast.ml	103	Abstract syntax tree & pretty printer
semant.ml	158	Semantic checking
codegen.ml	183	LLVM IR generation
microc.ml	20	Top-level
Total	624	

Type	Files	Total lines
Working .mc	34	413
Working outputs	34	105
Failing .mc	31	298
Error messages	31	31
Total	130	847