The MicroC Compiler

Only difference with NanoC is that here we support function calls.

example.mc

```
/* The GCD algorithm in MicroC */
int a;
int b;

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

int main() {
  int x;
  int y;
  a = 18;
  b = 9;
  x = 2;
  y = 14;
  print(gcd(x,y));
  print(gcd(3,15));
  print(gcd(99,121));
  print(gcd(a,b));
  return 0;
}
```

ast.ml

```
// Extends nanoc with additional microc stuff
(* Abstract Syntax Tree and functions for printing it *)

type op = Add | Sub | Equal | Neq | Less | And | Or

type typ = Int | Bool

type expr =
    Literal of int
  | BoolLit of bool
  | Id of string
  | Binop of expr * op * expr
  | Assign of string * expr
  (* function call *)                // an expression can now be a function call
  | Call of string * expr list       // string is name of func, expr list is the arguments

type stmt =
    Block of stmt list
  | Expr of expr
  | If of expr * stmt * stmt
  | While of expr * stmt
  (* return *)                       // return statement from a function
  | Return of expr                   // that accepts an expression to return

(* int x: name binding *)
type bind = typ * string

(* func_def: ret_typ fname formals locals body *)
type func_def = {                    // dunction definition
  rtyp: typ;                         // return type
  fname: string;                     // function name
  formals: bind list;                // formal parameters
  locals: bind list;                 // local variables
  body: stmt list;                   // body (list of statements)
}
```

```
type program = bind list * func_def list     // program is global variable list and list of function definitions

(* Pretty-printing functions *)
let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | And -> "&&"
  | Or -> "||"

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | Id(s) -> s
  | Binop(e1, o, e2) ->
    string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"

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, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
                      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let string_of_typ = function
    Int -> "int"
  | Bool -> "bool"

let string_of_vdecl (t, id) = string_of_typ t ^ " " ^ id ^ ";\n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.rtyp ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map snd 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) =
  "\n\nParsed program: \n\n" ^
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

scanner.mll

```
(* Ocamllex scanner for MicroC *)

{ open Microcparse }

let digit = ['0'-'9']
let letter = ['a'-'z' 'A'-'Z']

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
(* COMMA *)
| ','      { COMMA }  // comma (to separate arguments), microcparse.mly was modified to add COMMA.
| '+'      { PLUS }
| '-'      { MINUS }
| '='      { ASSIGN }
```

```
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "&&"     { AND }
| "||"     { OR }
| "if"     { IF }
| "else"   { ELSE }
| "while"  { WHILE }
(* RETURN *)
| "return" { RETURN }  // return (from function), microcparse.mly was modified to add RETURN.
| "int"    { INT }
| "bool"   { BOOL }
| "true"   { BLIT(true)  }
| "false"  { BLIT(false) }
| digit+ as lem  { LITERAL(int_of_string lem) }
| letter (digit | letter | '_')* as lem { ID(lem) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

microcparse.mly

```
/* Ocamlyacc parser for MicroC */

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE PLUS MINUS ASSIGN
%token EQ NEQ LT AND OR
%token IF ELSE WHILE INT BOOL
/* return, COMMA token */
%token RETURN COMMA    // new tokens for return and comma in MicroC
%token <int> LITERAL
%token <bool> BLIT
%token <string> ID
%token EOF

%start program
%type <Ast.program> program

%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT
%left PLUS MINUS

%%

/* add function declarations*/
program:                      // program is a list of declarations
  decls EOF { $1}             // just returns the list

decls:                        // declaration list of variables or functions
   /* nothing */ { ([], [])               }
 | vdecl SEMI decls { (($1 :: fst $3), snd $3) }     // add vdecl to first list
 | fdecl decls { (fst $2, ($1 :: snd $2)) }          // add fdecl to second list

vdecl_list:
  /*nothing*/ { [] }
  | vdecl SEMI vdecl_list  {  $1 :: $3 }

/* int x */
vdecl:
  typ ID { ($1, $2) }
// ALTERNATIVELY (but that causes shift/reduce error, as described further):
// program:
//   vdecl_list fdecl_list EOF { ($1, $2) }
```

```
typ:
    INT   { Int   }
  | BOOL  { Bool  }

/* fdecl */
// like: int gcd(int a, int b) { }
fdecl:
  vdecl LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
  {
    {
      rtyp=fst $1;
      fname=snd $1;
      formals=$3;
      locals=$6;
      body=$7
    }
  }

/* formals_opt */
formals_opt:
  /*nothing*/ { [] }
  | formals_list { $1 }

formals_list:
  vdecl { [$1] }
  | vdecl COMMA formals_list { $1::$3 }

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

stmt:
    expr SEMI                               { Expr $1      }
  | LBRACE stmt_list RBRACE                 { Block $2 }
  /* if (condition) { block1} else {block2} */
  /* if (condition) stmt else stmt */
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7) }
  | WHILE LPAREN expr RPAREN stmt           { While ($3, $5)  }
  /* return */
  | RETURN expr SEMI                        { Return $2      }      // new for return statement

expr:
    LITERAL          { Literal($1)            }
  | BLIT             { BoolLit($1)            }
  | ID               { Id($1)                 }
  | expr PLUS   expr { Binop($1, Add,   $3)   }
  | expr MINUS  expr { Binop($1, Sub,   $3)   }
  | expr EQ     expr { Binop($1, Equal, $3)   }
  | expr NEQ    expr { Binop($1, Neq, $3)     }
  | expr LT     expr { Binop($1, Less,  $3)   }
  | expr AND    expr { Binop($1, And,   $3)   }
  | expr OR     expr { Binop($1, Or,    $3)   }
  | ID ASSIGN expr   { Assign($1, $3)         }
  | LPAREN expr RPAREN { $2                   }
  /* call */
  | ID LPAREN args_opt RPAREN { Call ($1, $3)  }     // function call

/* args_opt*/
args_opt:              // actual arguments (no type, unlike formals_opt)
  /*nothing*/ { [] }
  | args { $1 }

args:
  expr  { [$1] }       // NOTE: we return as a list
  | expr COMMA args { $1::$3 }


// Here we can compile the above with:
//   ocamlyacc microcparse.mly
```

Gets **"4 shift/reduce conflicts"** - Meaning that during state transition automata, there are some states that have two or more choices for an incoming token to reduce the top of the stack to a non-terminal or shift the incoming token to the stack.
Run **ocamlyacc –v microcparse.mly**

```
Open microcparse.output
Search for "conflict", we find in "state 1" (with empty stack top) that we have 2 choices:
  1: shift/reduce conflict (shift 3, reduce 2) on INT
  1: shift/reduce conflict (shift 4, reduce 2) on BOOL
  state 1
    %entry% : '\001' . program (38)
    vdecl_list: . (2)  // to apply rule 2 to reduce to a vdecl_list
    INT shift 3        // to shift an incoming INT to the stack
    BOOL shift 4       // to shift an incoming BOOL to the stack
This is because of the grammar below (that was corrected in the microcparse.mly above):
  program:
    vdecl_list fdecl_list EOF { ($1, $2) }
  vdecl_list:
    /* nothing */ { [] }
    | vdecl SEMI vdecl_list { $1 :: $3 }
  vdecl:
    typ ID { ($1, $2) }
  typ:
    INT { Int }
    | BOOL { Bool }
  fdecl_list:
    /* nothing */ { [] }
    | fdecl fdecl_list { $1 :: $2 }
  fdecl:
    vdecl LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE {
      ... }
At the beginning the stack is empty and say the next incoming token is INT or BOOL, you have choices to:
1) apply vdecl_list to turn top of stack into an empty list of variable declarations (i.e. there are no global
variables) and then use fdecl_list to accept the INT or BOOL (since fdecl can also take an INT or BOOL
    vdecl_list: . (2)  // means stack empty and via rule 2 reduce to a vdecl_list (no more var decls after!)
2) shift INT to top of stack and later try to parse it into variable declaration (vdecl cia vdecl_list) or
function declaration (fdecl via fdecl_list)
The problem is we define both variable and function declarations as (potentially empty) lists:
  program:
    vdecl_list fdecl_list EOF { ($1, $2) }
and both vdecl and fdecl can begin with INT or BOOL, so when we receive INT or BOOL we don't know if it is a
variable or function declaration.
An empty top of stack can definitely be turned into an empty list, say vdecl_list, but then:
  vdecl_list:
    /* nothing */ { [] }
    | vdecl SEMI vdecl_list { ($1 :: $3 )
would prevent us from appending any more variable declarations, once we get a variable declaration list – we
must append function declarations, since above says a variable declaration can only appear before variable
declaration list.
We can try to change it to:
  program:
    vdecl_list fdecl_list EOF { (List.rev $1, $2) }
  vdecl_list:
    /* nothing */ { [] }
    | vdecl_list vdecl SEMI { $2 :: $1 }
  fdecl:
    vdecl LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE {
      ...
      locals = List.rev $6
      ... }
to try to group the precious variable declarations into a list and simply append new variable declarations to
the previously constructed list (resulting in a reverse list) and add a semicolon. After the above correction,
there are no more shift/reduce conflicts – initially we have an empty stack, which is reduced to vdecl_list,
then if we get INT we shift to stack, then if we get ID we reduce to vdecl, then 1) if we get SEMI, we do shift
and reduce top of stack to vdecl and connect it to vdecl_list to form a new vdecl_list; 2) if we get LPAREN, we
know this is the end of vdecl_list and we try to parse function declaration.
.
More efficient way that also allows us to mix definitions of functions and global variables and to make grammar
non-ambiguous is to do (this is in the microcparse.mly above):
  program:                    // program is a list of declarations
    decls EOF { $1}           // just returns the list
  decls:                              // declaration list of variables or functions
     /* nothing */ { ([], [])}
   | vdecl SEMI decls { (($1 :: fst $3), snd $3) }   // add vdecl to first list
   | fdecl decls { (fst $2, ($1 :: snd $2)) }        // add fdecl to second list
and we can leave the vdecl_list order as previously:
  vdecl_list:
    /*nothing*/ { [] }
```

```
    | vdecl SEMI vdecl_list  {  $1 :: $3 }
```

because we don’t have the reduce/shift conflict now – before we could not decide if, looking ahead one token, this is the end of the variable declaration list, now this is not necessary because the list is mixed, we don’t need to decide where the end of the variable declaration list is, we just keep appending variable and function declarations.
We can also remove fdecl_list, because it is no longer needed.
We can compile w/o any shift/reduce conflict.
To test:
  ocamlbuild test1.native
first time it will tell you to remove the **microcparse.ml** and **microcparse.mly** files that were generated from **ocamlyacc microcparse.mly** (just delete them).
Then run **./test1.native** and paste some program to **stdin**, which will be output in parsed form to **stdout**.

sast.ml

```
(* Semantically-checked Abstract Syntax Tree and functions for printing it *)

open Ast

type sexpr = typ * sx
and sx =
    SLiteral of int
  | SBoolLit of bool
  | SId of string
  | SBinop of sexpr * op * sexpr
  | SAssign of string * sexpr
  (* call *)
  | SCall of string * sexpr list      // function call

type sstmt =
    SBlock of sstmt list
  | SExpr of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SWhile of sexpr * sstmt
  (* return *)
  | SReturn of sexpr                  // return with a value (expression)

(* func_def: ret_typ fname formals locals body *)
type sfunc_def = {                    // this was just sprogram before w/ slocals and sbody
  srtyp: typ;                         // now it has return type, name, and formals
  sfname: string;
  sformals: bind list;
  slocals: bind list;
  sbody: sstmt list;
}

type sprogram = bind list * sfunc_def list // now this is a pair of variable and function declarations

(* Pretty-printing functions *)       // also modified for microc
let rec string_of_sexpr (t, e) =
  "(" ^ string_of_typ t ^ " : " ^ (match e with
        SLiteral(l) -> string_of_int l
      | SBoolLit(true) -> "true"
      | SBoolLit(false) -> "false"
      | SId(s) -> s
      | SBinop(e1, o, e2) ->
        string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
      | SAssign(v, e) -> v ^ " = " ^ string_of_sexpr e
      | SCall(f, el) ->
          f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
    ) ^ ")"

let rec string_of_sstmt = function
    SBlock(stmts) ->
    "{\n" ^ String.concat "" (List.map string_of_sstmt stmts) ^ "}\n"
  | SExpr(expr) -> string_of_sexpr expr ^ ";\n"
  | SReturn(expr) -> "return " ^ string_of_sexpr expr ^ ";\n"
  | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
                       string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt s

let string_of_sfdecl fdecl =
  string_of_typ fdecl.srtyp ^ " " ^
```

```
  fdecl.sfname ^ "(" ^ String.concat ", " (List.map snd fdecl.sformals) ^
  ")\n{\n" ^
  String.concat "" (List.map string_of_vdecl fdecl.slocals) ^
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
  "}\n"

let string_of_sprogram (vars, funcs) =
  "\n\nSementically checked program: \n\n" ^
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_sfdecl funcs)
```

semant.ml

```
(* Semantic checking for the MicroC compiler *)

open Ast
open Sast

module StringMap = Map.Make(String)

(* Semantic checking of the AST. Returns an SAST if successful,
   throws an exception if something is wrong.

   Check each global variable, then check each function *)

let check (globals, functions) =      // before that was "let check program ="
                                      // which we now rename to "let check_func func =" (below)
                                      // and create "let check (globals, functions)" at one level above here

  (* Verify a list of bindings has no duplicate names *)
  let check_binds (kind : string) (binds : (typ * string) list) =
    let rec dups = function
        [] -> ()
      | ((_,n1) :: (_,n2) :: _) when n1 = n2 ->
        raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
      | _ :: t -> dups t
    in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)
  in

  (* Make sure no globals duplicate *)
  check_binds "global" globals;       // no duplicate global names

  (* Collect function declarations for built-in functions: no bodies *)
  let built_in_decls =        // built-in function declarations!!!
    StringMap.add "print" {
      rtyp = Int;
      fname = "print";
      formals = [(Int, "x")];
      locals = []; body = [] } StringMap.empty
  in

  (* Add function name to symbol table *)
  let add_func map fd =       // function to construct a map for func name -> func declarations
    let built_in_err = "function " ^ fd.fname ^ " may not be defined"
    and dup_err = "duplicate function " ^ fd.fname
    and make_err er = raise (Failure er)
    and n = fd.fname (* Name of the function *)
    in match fd with (* No duplicate functions or redefinitions of built-ins *)
      _ when StringMap.mem n built_in_decls -> make_err built_in_err
    | _ when StringMap.mem n map -> make_err dup_err
    | _ ->  StringMap.add n fd map
  in

  (* Collect all function names into one symbol table *)
  // just apply add_func to all built-in and declared functions
  // and aggregate the result into function_decls map
  let function_decls = List.fold_left add_func built_in_decls functions
  in

  (* Return a function from our symbol table *)
  let find_func s =           // simply looks up in function_decls map
    try StringMap.find s function_decls
    with Not_found -> raise (Failure ("unrecognized function " ^ s))
```

```
in

let _ = find_func "main" in (* Ensure "main" is defined *)        // check main func has been defined

let check_func func =
  (* Make sure no formals or locals are void or duplicates *)
  check_binds "formal" func.formals;       // no duplicates in formal arg declarations
  check_binds "local" func.locals;         // no duplicates in local var declarations

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

  (* Build local symbol table of variables for this function *)
  // We modify to include the global var declarations and the function arguments
  // by simply concatenating the separate lists of variable declarations
  // NOTE: if you have a local var with same name as global var, it will be shadowed, same for args
  // because List.fold_left works head (left) to tail (right), processing local AFTER (overwriting) globals
  let symbols = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
      StringMap.empty (globals @ func.formals @ func.locals)
  in

  (* Return a variable from our local symbol table *)
  let type_of_identifier s =               // this remains the same
    try StringMap.find s symbols
    with Not_found -> raise (Failure ("undeclared identifier " ^ s))
  in

  (* Return a semantically-checked expression, i.e., with a type *)
  let rec check_expr = function
      Literal l -> (Int, SLiteral l)
    | BoolLit l -> (Bool, SBoolLit l)
    | Id var -> (type_of_identifier var, SId var)
    | Assign(var, e) as ex ->
      let lt = type_of_identifier var
      and (rt, e') = check_expr e in
      let err = "illegal assignment " ^ string_of_typ lt ^ " = " ^
                string_of_typ rt ^ " in " ^ string_of_expr ex
      in
      (check_assign lt rt err, SAssign(var, (rt, e')))

    | Binop(e1, op, e2) as e ->
      let (t1, e1') = check_expr e1
      and (t2, e2') = check_expr e2 in
      let err = "illegal binary operator " ^
                string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
                string_of_typ t2 ^ " in " ^ string_of_expr e
      in
      (* All binary operators require operands of the same type*)
      if t1 = t2 then
        (* Determine expression type based on operator and operand types *)
        let t = match op with
            Add | Sub when t1 = Int -> Int
          | Equal | Neq -> Bool
          | Less when t1 = Int -> Bool
          | And | Or when t1 = Bool -> Bool
          | _ -> raise (Failure err)
        in
        (t, SBinop((t1, e1'), op, (t2, e2')))
      else raise (Failure err)

    | Call(fname, args) as call -> // new for function call (fname and list of args)
      let fd = find_func fname in                 // find the function by name
      let param_length = List.length fd.formals in
      if List.length args != param_length then    // check number of parameters
        raise (Failure ("expecting " ^ string_of_int param_length ^
                        " arguments in " ^ string_of_expr call))
      else let check_call (ft, _) e =             // check type and expression of parameter
             let (et, e') = check_expr e in
             let err = "illegal argument found " ^ string_of_typ et ^
                       " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e
```

```
                in (check_assign ft et err, e')
          in
          let args' = List.map2 check_call fd.formals args  // check types of all parameters
          in
          (fd.rtyp, SCall(fname, args'))      // creates a semantically-checked expression (w/ a type)
    in

    let check_bool_expr e =
      let (t, e') = check_expr e in
      match t with
      | Bool -> (t, e')
      |  _ -> raise (Failure ("expected Boolean expression in " ^ string_of_expr e))
    in

    let rec check_stmt_list =function
        [] -> []
      | Block sl :: sl'  -> check_stmt_list (sl @ sl') (* Flatten blocks *)
      | s :: sl -> check_stmt s :: check_stmt_list sl
    (* Return a semantically-checked statement i.e. containing sexprs *)

    and check_stmt =function  // NOTE: we inherit func from check_func above!!!
      (* A block is correct if each statement is correct and nothing
         follows any Return statement.  Nested blocks are flattened. *)
        Block sl -> SBlock (check_stmt_list sl)
      | Expr e -> SExpr (check_expr e)
      | If(e, st1, st2) ->
        SIf(check_bool_expr e, check_stmt st1, check_stmt st2)
      | While(e, st) ->
        SWhile(check_bool_expr e, check_stmt st)
      | Return e ->                  // this is added to previous definition
        let (t, e') = check_expr e in
        if t = func.rtyp then SReturn (t, e')        // check return expression, if type matches func
        else raise (                                 // else error, NOTE: could use check_assign for check
            Failure ("return gives " ^ string_of_typ t ^ " expected " ^
                     string_of_typ func.rtyp ^ " in " ^ string_of_expr e))

    in (* body of check_func *)       // this now constructs a func
    { srtyp = func.rtyp;              // return type remains the same
      sfname = func.fname;            // function name remains the same
      sformals = func.formals;        // formals remain the same
      slocals  = func.locals;         // locals remain the same
      sbody = check_stmt_list func.body      // function body is checked
    }

  in
  (globals, List.map check_func functions)   // return for check (globals, functions) is Sast.sprogram
                                             // since globals are not changed, they just remain globals
                                             // the second part is applying the check_func to all functions


// To build the semantic checker:
//  ocamlbuild test2.native
// to run:
//  ./test2.native
// and input the test program to stdin to get the typed semantic tree to stdout


// You can run some buggy examples to check if the semantic checker can catch errors:
// undeclared main
int x;
// undeclared variable
int x;
int main() {
  x = y + 1;
}
// duplicate globals
int x;
int x;
// duplicate arguments
int main(int a, int a) {
}
// duplicate locals
int main() {
```

```
  int a;
  int a;
}
// undeclared function
int main() {
  f();
}
// duplicate func
int f() {}
int f() {}
// wrong ret type
int f() {
  return true;
}
int main() {}
// wrong arg type
int f(int x) {
  return x;
}
int main() {
  return f(true);
}
// wrong arg number
int f(int x) {
  return x;
}
int main() {
  return f(5, 7);
}
// wring function type
bool f(bool x) {
  return false;
}
int main() {
  int x;
  x = f(true);
}
```

irgen.ml

```
(* IR generation: translate takes a semantically checked AST and
   produces LLVM IR
   LLVM tutorial: Make sure to read the OCaml version of the tutorial
   http://llvm.org/docs/tutorial/index.html
   Detailed documentation on the OCaml LLVM library:
   http://llvm.moe/
   http://llvm.moe/ocaml/
*)

module L = Llvm
module A = Ast
open Sast

module StringMap = Map.Make(String)

(* translate : Sast.program -> Llvm.module *)
let translate (globals, functions) =  // parameters changed to be globals + list of functions
  let context    = L.global_context () in

  (* Create the LLVM compilation module into which
     we will generate code *)
  let the_module = L.create_module context "MicroC" in

  (* Get types from the context *)
  let i32_t      = L.i32_type    context
  and i8_t       = L.i8_type     context
  and i1_t       = L.i1_type     context in

  (* Return the LLVM type for a MicroC type *)
  let ltype_of_typ = function
      A.Int   -> i32_t
    | A.Bool  -> i1_t
  in
```

```
(* Create a map of global variables after creating each *)
let global_vars : L.llvalue StringMap.t =
  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

let printf_t : L.lltype =   // function declaration for built-in print
  L.var_arg_function_type i32_t [| L.pointer_type i8_t |] in
let printf_func : L.llvalue =
  L.declare_function "printf" printf_t the_module in

(* Define each function (arguments and return type) so we can
   call it even before we've created its body *)
let function_decls : (L.llvalue * sfunc_def) StringMap.t =       // map to lookup func decl by name
  let function_decl m fdecl =
    let name = fdecl.sfname // get name
    and formal_types =      // get llvm types for formal args
      Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.sformals)
    in let ftype = L.function_type (ltype_of_typ fdecl.srtyp) formal_types in    // gen llvm func decl
    StringMap.add name (L.define_function name ftype the_module, fdecl) m in     // insert in func decl map
  List.fold_left function_decl StringMap.empty functions in      // aggregate function list into decl map

(* Fill in the body of the given function *)
// used to build all the functions in microc
let build_function_body fdecl =

  // gets the func location where we insert the body (i.e. the function declaration)
  let (the_function, _) = StringMap.find fdecl.sfname function_decls in

  // the builder is just at the end of this block (which is empty at the beginning)
  let builder = L.builder_at_end context (L.entry_block the_function) in

  let int_format_str = L.build_global_stringptr "%d\n" "fmt" builder in  // helper, for built-in print

  (* Construct the function's "locals": formal arguments and locally
     declared variables.  Allocate each on the stack, initialize their
     value, if appropriate, and remember their values in the "locals" map *)
  // builds global variables
  let local_vars =

    let add_formal m (t, n) p =
      L.set_value_name n p; // name of the formal
      let local = L.build_alloca (ltype_of_typ t) n builder in    // allocate the data on the stack
      ignore (L.build_store p local builder);      // store the value from caller stack frame to callee frame
                                                   // this way we can change it
      StringMap.add n local m       // add to local variable map
    (* Allocate space for any locally declared variables and add the
     * resulting registers to our map *)
    and add_local m (t, n) =        // for local vars...
      let local_var = L.build_alloca (ltype_of_typ t) n builder   // we simply allocate space
      in StringMap.add n local_var m       // and add to local variable map
    in

    let formals = List.fold_left2 add_formal StringMap.empty fdecl.sformals      // aggregate to single map
        (Array.to_list (L.params the_function)) in
    List.fold_left add_local formals fdecl.slocals
  in

  (* Return the value for a variable or formal argument.
     Check local names first, then global names *)
  let lookup n = try StringMap.find n local_vars  // given a var name, query local table, then global table
    with Not_found -> StringMap.find n global_vars
  in

  (* Construct code for an expression; return its value *)
  let rec build_expr builder ((_, e) : sexpr) = match e with
      SLiteral i  -> L.const_int i32_t i
    | SBoolLit b  -> L.const_int i1_t (if b then 1 else 0)
    | SId s       -> L.build_load (lookup s) s builder
    | SAssign (s, e) -> let e' = build_expr builder e in
      ignore(L.build_store e' (lookup s) builder); e'
```

```
    | SBinop (e1, op, e2) ->
      let e1' = build_expr builder e1
      and e2' = build_expr builder e2 in
      (match op with
         A.Add     -> L.build_add
       | A.Sub     -> L.build_sub
       | 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
      ) e1' e2' "tmp" builder

    // new for built-in print, just call printf_func that we defined above
    | SCall ("print", [e]) ->
      L.build_call printf_func [| int_format_str ; (build_expr builder e) |]
        "printf" builder

    // new for general function call
    | SCall (f, args) ->
      let (fdef, fdecl) = StringMap.find f function_decls in      // get func info from table
     // generate code to calculate the expressions for each argument, in reverse order
     // for each argument in reverse, generate a list of locations for its expression
     // llags is a list of addresses for the results of all arguments
     // reverse order because we can access them as stack ptr - 1, 2, 3, etc. (since stack grows downward),
     // instead of the more unintuitive stack ptr - # args + 1, 2, 3, ...
     // similarly, we can access local vars as stack ptr + 1, 2, 3, ...
      let llargs = List.rev (List.map (build_expr builder) (List.rev args)) in
      let result = f ^ "_result" in // name of return value
      L.build_call fdef (Array.of_list llargs) result builder     // build call from args, func, and retval
  in

  (* LLVM insists each basic block end with exactly one "terminator"
     instruction that transfers control.  This function runs "instr builder"
     if the current block does not already have a terminator.  Used,
     e.g., to handle the "fall off the end of the function" case. *)
  let add_terminal builder instr =
    match L.block_terminator (L.insertion_block builder) with
      Some _ -> ()
    | None -> ignore (instr builder) in

  (* Build the code for the given statement; return the builder for
     the statement's successor (i.e., the next instruction will be built
     after the one generated by this call) *)
  let rec build_stmt builder = function    // removed the_function param here since we set it above

      SBlock sl -> List.fold_left build_stmt builder sl
    | SExpr e -> ignore(build_expr builder e); builder

    | SReturn e -> ignore(L.build_ret (build_expr builder e) builder); builder // new for return
             // ignore return value from build_ret but return the changed builder

    | SIf (predicate, then_stmt, else_stmt) ->
      let bool_val = build_expr builder predicate in

      let then_bb = L.append_block context "then" the_function in
      ignore (build_stmt (L.builder_at_end context then_bb) then_stmt);
      let else_bb = L.append_block context "else" the_function in
      ignore (build_stmt (L.builder_at_end context else_bb) else_stmt);

      let end_bb = L.append_block context "if_end" the_function in
      let build_br_end = L.build_br end_bb in (* partial function *)
      add_terminal (L.builder_at_end context then_bb) build_br_end;
      add_terminal (L.builder_at_end context else_bb) build_br_end;

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

    | SWhile (predicate, body) ->
      let while_bb = L.append_block context "while" the_function in
      let build_br_while = L.build_br while_bb in (* partial function *)
      ignore (build_br_while builder);
      let while_builder = L.builder_at_end context while_bb in
```

```
        let bool_val = build_expr while_builder predicate in

        let body_bb = L.append_block context "while_body" the_function in
        add_terminal (build_stmt (L.builder_at_end context body_bb) body) build_br_while;

        let end_bb = L.append_block context "while_end" the_function in

        ignore(L.build_cond_br bool_val body_bb end_bb while_builder);
        L.builder_at_end context end_bb

    in
    (* Build the code for each statement in the function *)
    let func_builder = build_stmt builder (SBlock fdecl.sbody) in

    (* Add a return if the last block falls off the end *)
    add_terminal func_builder (L.build_ret (L.const_int i32_t 0))

  in    // body for let translate (globals, functions)

  List.iter build_function_body functions;   // build the function bodies for all functions, iter returns unit
                                              // build_function_body manipulates builder, which is mutable
  the_module
```

microc.ml

```
// modified to support microc testing

(* Top-level of the MicroC compiler: scan & parse the input,
   check the resulting AST and generate an SAST from it, generate LLVM IR,
   and dump the module *)

type action = Ast | Sast | LLVM_IR

let () =
  let action = ref LLVM_IR in
  let set_action a () = action := a in
  let speclist = [
    ("-a", Arg.Unit (set_action Ast), "Print the AST");
    ("-s", Arg.Unit (set_action Sast), "Print the SAST");
    ("-l", Arg.Unit (set_action LLVM_IR), "Print the generated LLVM IR");
  ] in
  let usage_msg = "usage: ./microc.native [-a|-s|-l] [file.mc]" in
  let channel = ref stdin in
  Arg.parse speclist (fun filename -> channel := open_in filename) usage_msg;

  let lexbuf = Lexing.from_channel !channel in

  let ast = Microcparse.program Scanner.token lexbuf in
  match !action with
    Ast -> print_string (Ast.string_of_program ast)
  | _ -> let sast = Semant.check ast in
    match !action with
      Ast     -> ()
    | Sast    -> print_string (Sast.string_of_sprogram sast)
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Irgen.translate sast))


// To test the whole program
//   ocamlbuild -pkgs llvm microc.native
// Then:
//   ./microc.native -l example.mc
// to show the generated LLVM code
// NOTE: it is much simpler than nanoc because we have more complex control flow and avoid blocks
// To run:
//   ./microc.native -l example.mc > example.out
//   lli example.out
```