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) }
 | fdecl decls { (fst $2, ($1 :: snd $2)) }

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

/* int x */
vdecl:
  typ ID { ($1, $2) }
```

```
// ALTERNATIVELY:
// 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
```