The NanoC Compiler

example.mc

```
// Example of a program written in NanoC, a subset of C:
// * no func calls, etc
// * bool and int built-in types
// * control flows: while and if
// * assignment, comparison, subtraction
// GCD example

int a;
int b;
a = 18;
b = 9;

while (a != b) {
  if (b < a) a = a - b;
  else b = b - a;
}

a = a;
```

example.out

```
// LLVM code for example.mc
// Run with: lli example.out

; ModuleID = 'NanoC'
source_filename = "NanoC"

@fmt = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.1 = private unnamed_addr constant [4 x i8] c"%g\0A\00", align 1
@fmt.2 = private unnamed_addr constant [4 x i8] c"%d\0A\00", align 1
@fmt.3 = private unnamed_addr constant [4 x i8] c"%g\0A\00", align 1


@a = global i32 0              // int a;
@b = global i32 0              // int b;

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

define i32 @main() {
entry:
  store i32 18, i32* @a        // a = 18;
  store i32 9, i32* @b         // b = 9;
  br label %while

while:                                            ; preds = %merge, %entry
  %a7 = load i32, i32* @a
  %b8 = load i32, i32* @b
  %tmp9 = icmp ne i32 %a7, %b8
  br i1 %tmp9, label %while_body, label %merge10

while_body:                                       ; preds = %while
  %b = load i32, i32* @b
  %a = load i32, i32* @a
  %tmp = icmp slt i32 %b, %a
  br i1 %tmp, label %then, label %else

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

then:                                             ; preds = %while_body
  %a1 = load i32, i32* @a
  %b2 = load i32, i32* @b
  %tmp3 = sub i32 %a1, %b2
  store i32 %tmp3, i32* @a
  br label %merge
```

```
else:                                               ; preds = %while_body
  %b4 = load i32, i32* @b
  %a5 = load i32, i32* @a
  %tmp6 = sub i32 %b4, %a5
  store i32 %tmp6, i32* @b
  br label %merge

merge10:                                            ; preds = %while
  %a11 = load i32, i32* @a
  store i32 %a11, i32* @a
  %printf = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([4 x i8], [4 x i8]* @fmt, i32 0, i32 0),
i32 %a11)
  ret i32 0
}
```

ast.ml

```
// Abstract syntax tree:
//      1. This is the scanner and parser combined to produce an AST
//      2. Then semantic analysis parses the AST and does checks and adds type information
//      3. Then IR generation converts the decorated AST into IR code

(* Abstract Syntax Tree and functions for printing it *)

type op = Add | Sub | Equal | Neq | Less | And | Or         // binary operators

type typ = Int | Bool                                       // types

type expr =                                                 // AST for expression
    Literal of int                                          // number literal
  | BoolLit of bool                                         // boolean literal (true or false)
  | Id of string                                            // identifier
  | Binop of expr * op * expr                               // binary oper over 2 smaller exp's
  | Assign of string * expr                                 // assignment, in C it has ret val

type stmt =                                                 // AST for statement
    Block of stmt list                                      // list of statements
  | Expr of expr                                            // simple expression (ex. Assignment)
  | If of expr * stmt * stmt                                // if statement
  | While of expr * stmt                                    // while statement

type program = {                                            // AST for program (NOTE: a record)
  locals: (typ * string) list;                              // declarations
  body: stmt list;                                          // statements
}

// ALTERNATIVELY:
        type bind = typ * string
        type program = {
          locals: bind list;
          body: stmt list;
        }

// AST is done above, below is just functions to print the original program for debugging

(* 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
```

```
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";
  | 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_program fdecl =
  "\nParsed program: \n" ^
  String.concat "" (List.map string_of_vdecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "\n"
```

nanocparse.mly

```
// Ocaml yacc file
// Tokens for AST and parser to construct the AST

/* Ocamlyacc parser for NanoC */

%{
open Ast                                                // definition of the AST
%}

// All acceptable tokens for language

%token SEMI LPAREN RPAREN LBRACE RBRACE PLUS MINUS ASSIGN
%token EQ NEQ LT AND OR
%token IF ELSE WHILE INT BOOL
%token <int> LITERAL                                    // this token exports an int
%token <bool> BLIT                                      // this token exports a bool
%token <string> ID                                      // this token exports a string (var)
%token EOF                                              // special token for EOF

%start program                                          // entry rule for the parser
%type <Ast.program> program                             // this rule exports an object of type Ast.program
// NOTE: from ast.ml:
//      type program = {
//        locals: (typ * string) list;
//        body: stmt list;
//      }

// Associativity and precedence (in order of increasing precedence!)
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT
%left PLUS MINUS

// See the scanner in scanner.mll before proceeding

%%

// Rules to parse the input (program)
program:                                                // progam is:
  vdecl_list stmt_list EOF { {locals=$1; body=$2} }     // declarations followed by statements and EOF
                                                        // assigns vdecl_list to local and stmt_list to body
vdecl_list:
  /*nothing*/ { [] }                                    // empty list!
  | vdecl vdecl_list { $1 :: $2 }                       // list of head (declaration) + tail (may be empty)

vdecl:
  typ ID SEMI { ($1, $2) }                              // variable declaration: type, identifier, semicolon
                                                        // produces a tuple of (type, identifier)
```

```
typ:                                                    // NOTE: from ast.ml: type bind = typ * string
    INT   { Int   }
  | BOOL  { Bool  }

stmt_list:                                              // similar to list of declarations
  /*nothing*/ { [] }
  | stmt stmt_list { $1 :: $2 }

stmt:
    expr SEMI                                 { Expr $1               }      // from ast.ml: Expr of expr
// NOTE: if stml_list was defined as: stmt_list stmt { $2 :: $1 } above
// then the below should have { Block (List.rev $2) } for the correct order
  | LBRACE stmt_list RBRACE                   { Block $2    }              // from ast.ml: Block of stmt list
  | IF LPAREN expr RPAREN stmt ELSE stmt      { If ($3, $5, $7)        }     // If of expr * stmt * stmt
  | WHILE LPAREN expr RPAREN stmt             { While ($3, $5) }             // While of expr * stmt

expr:
    LITERAL          { Literal($1)            }      // $1 here is the object exported by the LITERAL token
                                                     // from %token <int> LITERAL we know it exports an int
  | 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)         }      // ASSIGN is the "=", we turn it into Assign object
  | LPAREN expr RPAREN { $2                   }
```

scanner.mll

```
// Scanner for language - Ocaml lex file
// Maps patterns to tokens (with attributes) that are generated from them

(* Ocamllex scanner for NanoC *)

{ open Nanocparse }                   // Header to add OCaml native code
                                      // opens nanocparse module w/ token definitions
let digit = ['0'-'9']  in             // pattern for digit
let letter = ['a'-'z' 'A'-'Z'] in     // pattern for letter

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)// "token lexbuf" means apply token rule to rest of buf
| "/*"     { comment lexbuf }           (* Comments *)// apply the comment rule (below) to rest of input buffer
| '('      { LPAREN }                 // chars below are simply parsed in the respective token
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| '+'      { PLUS }
| '-'      { MINUS }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "&&"     { AND }
| "||"     { OR }
| "if"     { IF }
| "else"   { ELSE }
| "while"  { WHILE }
| "int"    { INT }
| "bool"   { BOOL }
// NOTE: from nanocparse.mly:
//      <bool> BLIT
//      …
//      | BLIT             { BoolLit($1)           }
// and from ast.ml:
//      | BoolLit of bool
| "true"   { BLIT(true)  }
| "false"  { BLIT(false) }
// NOTE: from nanocparse.mly:
```

```
//      %token <int> LITERAL
//      …
//      LITERAL          { Literal($1)           }
// and from ast.ml:
//      Literal of int
| digit+ as lxm  { LITERAL(int_of_string lxm)}       // construct LITERAL of int value of lxm (digit+)
// NOTE: from nanocparse.mly:
//      %token <string> ID
//      …
//      | ID               { Id($1)                 }
// and from ast.ml:
//      | Id of string
| letter (letter | digit | '_')* as lxm { ID(lxm) }
| eof { EOF }                                         // token for EOF
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }  // error for any other case

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

We can compile and try the above phases before we move on.

```
With something like this (say, in test1.ml):
open Ast
let _ =
  let lexbuf = Lexing.from_channel stdin in                // Lexing module turns input into lexical buffer
  let program = Nanocparse.program Scanner.token lexbuf in  // this assigns the AST in program
  print_endline (string_of_program program)                 // converts the AST back into program instructions


ocamlbuild test1.native               // test1.native is the target we are building
./test1.native
<input program text, say from example.mc>
^D
```

sast.ml

```
// Semantic analysis
// Adds type info to AST and checks types and scopes to produce a decorated AST
// The types here describe the decorated AST that results from semantic analysis

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

open Ast

// NOTE: here we know (or can infer) the type, so we want to decorate the AST with that information
// this is because nanoc is statically typed
// This is a new type for expressions that adds that type information
type sexpr = typ * sx                 // sexpr consists of type information + a typed expression (sx)
and sx =                              // a typed expr. (sx) is either a literal or operation on typed expr.'s
    SLiteral of int                   // NOTE: Can't reuse Literal here, has to be different in OCaml
  | SBoolLit of bool
  | SId of string
  | SBinop of sexpr * op * sexpr      // we must use sexpr here to have type information
  | SAssign of string * sexpr

// similar for statement
// This is a new type for statements that adds type information
type sstmt =
    SBlock of sstmt list
  | SExpr of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SWhile of sexpr * sstmt

// similar for program
type sprogram = {
  slocals : (typ * string) list;
  sbody : sstmt list;
}

// this concludes the semantically-decorated AST, below are printing functions

(* Pretty-printing functions *)
```

```
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
    ) ^ ")"

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";
  | 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_sprogram fdecl =
  "\nSementically checked program: \n" ^
  String.concat "" (List.map string_of_vdecl fdecl.slocals) ^
  String.concat "" (List.map string_of_sstmt fdecl.sbody) ^
  "\n"
```

semant.ml

```
// Semantic analysis module

(* Semantic checking for the NanoC 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 *)

// This is the semantic check function that maps AST to sAST
let check program =                           // function that takes a program of type Ast.program as input
                                              // and produces output of type Sast.sprogram

  // Given a list of variable declarations, check if there are any duplicates
  (* 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 ->      // then, check if the front two in list have same name
        raise (Failure ("duplicate " ^ kind ^ " " ^ n1))
      | _ :: t -> dups t                             // finally, check list tail
    in dups (List.sort (fun (_,a) (_,b) -> compare a b) binds)      // first, sort the declaration list
  in

  (* Make sure no locals duplicate *)
  check_binds "local" program.locals;

  // Builds the symbol table from variable declarations (no nesting)
  // MicroC will use multiple level
  (* Build local symbol table of variables for this function *)
  let symbols = List.fold_left (fun m (ty, name) -> StringMap.add name ty m)
      StringMap.empty program.locals
  in

  (* Return a variable from our local symbol table *)
  let type_of_identifier s =                 // looks up the type of an identifier in symbol table
    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              // takes expr and generates sexpr
      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                // LHS type
      and (rt, e') = check_expr e in                 // RHS type
      if lt = rt then (lt, SAssign(var, (rt, e')))   // NOTE: SAssign of string * sexpr
      else raise (Failure ("illegal assignment " ^ string_of_typ lt ^ " = " ^
                          string_of_typ rt ^ " in " ^ string_of_expr ex))

    | 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
      if t1 = t2 then
        let ty = 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 (ty, SBinop((t1, e1'), op, (t2, e2')))
      else raise (Failure err)
  in

  let check_bool_expr e =            // first, parse e (expr) into an sexpr, then check its type
    let (t, e') = check_expr e in
    if t = Bool then (t, e') else raise (Failure ("expected Boolean expression in " ^ string_of_expr e))
    // Above can be done as:
    //  match t with
    //  | Bool -> (t, e')
    //  | _ -> raise …
  in

// NOTE: use and keyword below because the definitions of check_stmt_list and check_stmt depend on one another
  let rec check_stmt_list = function         // takes stmt list and generates sstmt list
      [] -> []
    // Optimization to flatten blocks by concatenating to a single statement list
    | Block sl :: sl' -> check_stmt_list (sl @ sl')  // sl' is just another variable name, diff than sl
    | s :: sl -> check_stmt s :: check_stmt_list sl  // check_stmt s to sstmt then check_stmt_list the rest
  and check_stmt = function                  // takes stmt and converts it to sstmt
    | Expr e -> SExpr(check_expr e)
    | If(e, st1, st2) -> SIf(check_bool_expr e, check_stmt st1, check_stmt st2) // check_bool_expr forces bool
    | While(e, st) -> SWhile(check_bool_expr e, check_stmt st)             // check_bool_expr forces bool
    | Block sl -> SBlock(check_stmt_list sl)
  in
  {
    slocals = program.locals;        // since declarations in AST and decorated AST same: (typ * string) list
    sbody = check_stmt_list program.body // check_stmt_list is func to parse/decorate program.body to sstmt list
  }
```

<u>We can compile and try the above phases before we move on.</u>

With something like this (say, in test2.ml):

```
open Sast

let _ =
  let lexbuf = Lexing.from_channel stdin in                // Lexing module turns input into lexical buffer
  let program = Nanocparse.program Scanner.token lexbuf in  // this assigns the AST in program
  let sprogram = Semant.check program in                   // this checks the AST and converts it to sAST
  print_endline (string_of_program sprogram)               // converts the sAST back into program instructions
```

```
ocamlbuild test2.native              // test2.native is the target we are building
./test2.native
<input program text, say from example.mc>
^D
```

irgen.ml

```
// Intermediate code generator

(* 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        // open Llvm module and rename it to L
module A = Ast         // open Ast module and rename it to Ast
open Sast

module StringMap = Map.Make(String)

// translate function that takes Sast.program and returns Llvm.module
// this does all of the translation work
(* translate : Sast.program -> Llvm.module *)
let translate program =
  let context    = L.global_context () in    // define global context to remember types, functions, etc.

  (* Create the LLVM compilation module into which
     we will generate code *)
  let the_module = L.create_module context "NanoC" in       // create module for compiled code w/ context & name

  (* Get types from the context *)
  let i32_t      = L.i32_type    context     // get 32-bit int type from context (for integers)
  and i1_t       = L.i1_type     context in  // get 1-bit int type from content (for Booleans)

  (* Return the LLVM type for a NanoC type *)        // maps AST type to llvm type
  let ltype_of_typ = function
      A.Int   -> i32_t
    | A.Bool  -> i1_t
  in

  // map of global variables
  (* Create a map of global variables after creating each *)
  let global_vars : L.llvalue StringMap.t =
    // given a variable declaration w/ type t and name n, and a map m
    let global_var m (t, n) =
      // initial value
      let init = L.const_int (ltype_of_typ t) 0
      // define global var w/ name n, initial value init, and in the_module
      // i.e. given a map, global_var will add the name n and the global definition into the map
      in StringMap.add n (L.define_global n init the_module) m in
    // fold left all variable declarations in program.slocals
    List.fold_left global_var StringMap.empty program.slocals in

  // lookup a name to get the llvm global var definition for it
  (* Return the value for a variable or formal argument.
     Check global names *)
  let lookup n = StringMap.find n global_vars in

  // turns an expression into a list of llvm instructions
  // builder argument is the position for the next generated instruction, pointer to position in module
  //
  (* Construct code for an expression; return its value *)
  // returns the temporal that is equal to value of entire expression (in llvm each expr is new temporal)
  let rec build_expr builder ((_, e) : sexpr) = match e with
      SLiteral i  -> L.const_int i32_t i     // for number, return its const value (no code, just addr)
    | SBoolLit b  -> L.const_int i1_t (if b then 1 else 0)  // return 1 or 0 for bool (no code, just addr)
    // lookup var and load it (from static mem region for globals)
    // this generates code at the current position in builder and mutates builder to move to next position
    | SId s       -> L.build_load (lookup s) s builder
    // generate instruction to calculate e, return it in e' (a temporal) and then store it in memory store
    // and return the value of the expression e'
```

```
    | SAssign (s, e) -> let e' = build_expr builder e in
      ignore(L.build_store e' (lookup s) builder); e'
    // for binary ops, evalueate e1 and e2, then generate the instruction to perform the operation
    | 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 // generates new temporal name & inserts it into builder (to be used for result)
    // NOTE: the above is a 3-address code: the operator followed by 3 addresses (two source and one dest)
  in

  // we reviewed the concept of basic blocks, either:
  //   1. beginning of program, or
  //   2. starting after a label, or
  //   3. starting after some conditional jump
  // and is a sequence of instructions w/o any jump, until we hit a label or conditional jump
  // i.e. the only entry point in block is the first instruction and guarantees that, once entered, it completes
  // execution of all instructions in the block and the only exit is the end of the block
  // LLVM insists every basic block ends w/ jump or conditional jump, but in our language we do not enforce that
  // (ex. the “else” block from “Basic Blocks and Control-Flow Graphs” of ir.pdf slide does not end w/ any jump)
  // the benefit of always having jump at the end (as LLVM insists) is that you don’t care about the sequence of
  // the resulting blocks, because every block jumps to next block explicitly.
  (* 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. *)
  // so, this function terminates the basic block in builder with instr, if necessary
  let add_terminal builder instr =
    match L.block_terminator (L.insertion_block builder) with
      Some _ -> ()     // if block has a terminator, do nothing
    | None -> ignore (instr builder) in      // otherwise, add instr to the builder (as terminator)

  (* 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) *)
  // this is how we generate llvm instructions for a statement:
  // the_function is the statement you want to insert the instruction(s) for
  // builder is the builder to use when inserting the instruction(s)
  // and returns a builder (i.e. what is the next position to insert instructions)
  let rec build_stmt the_function builder = function
      // starting w/ provided builder, apply build_stmt function to the list of statements
      // each time returning a new builder that is forwarded to next statement in list
      // the result is an aggregated builder
      SBlock sl -> List.fold_left (build_stmt the_function) builder sl
    | SExpr e -> ignore(build_expr builder e); builder // for expr, simply generate list of instructions for it
    // ignore returned temporal value for expression above, since we want to return builder (already mutated)

    | SIf (predicate, then_stmt, else_stmt) ->
      let bool_val = build_expr builder predicate in // generate code for expression and remember its address

      let then_bb = L.append_block context "then" the_function in // generate label for then
      ignore (build_stmt the_function (L.builder_at_end context then_bb) then_stmt); // gen code for then branch
      // right after the then label that was just generated

      let else_bb = L.append_block context "else" the_function in // generate label for else
      ignore (build_stmt the_function (L.builder_at_end context else_bb) else_stmt); // gen code for else branch
      // right after the else label that was just generated

      let end_bb = L.append_block context "if_end" the_function in // generate label for end of if
      // because both then and else will jump to end of if

      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 jump to end of if to end of then
      add_terminal (L.builder_at_end context else_bb) build_br_end; // add jump to end of if to end of else

      ignore(L.build_cond_br bool_val then_bb else_bb builder); // build conditional jump to then_bb or else_bb
```

```
    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 the_function (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

(* Fill in the body of the given function *)
let build_program fdecl =
  (* Define each function (arguments and return type) so we can
     call it even before we've created its body *)
  let main_decl : L.llvalue =
    let ftype = L.function_type i32_t [||]
    in L.define_function "main" ftype the_module in
  let the_function = main_decl in

  let builder = L.builder_at_end context (L.entry_block the_function) in

  (* Build the code for each statement in the function *)
  let builder_end = List.fold_left (build_stmt the_function) builder fdecl.sbody in

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

build_program program;
the_module
```