# 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**