Compati Final Report:

Introduction

Compati is a toy version of C with all the feature of MicroC with the addition of

1. Fully functional structs, with functional member access and assignment
2. Stack allocated arrays
3. String type
4. Char type

As a toy version of C, compati has limited utility, but could potentially be utilized as a learning tool for others. Compati could be a good educational tool for new programmers, as it has nearly all the universal types most modern languages have and is syntactically similar to C.

Requirements

opam
llvm
install ocamlfind if necessary

Tutorial

Setup for compati is pretty simple as it uses Dune as it's build system, and the required dune files are already in my repository. Just clone or download, and then run

```
dune build
```

This will create an executable called **compati.exe** within the src file. After writing a compati program, saved with the .compati extension, the **program can be compiled** by running

```
dune exec -- ./compati.exe {INSERT PATH TO FILE}
```

This will also print the generated LLVM IR to console. Pass in -a after the file path to pretty print the ast, or -s for the pretty printed sast.

The **program can be run** by the following:

```
dune exec -- ./compati.exe {INSERT PATH TO FILE} >  {NAME OF FILE}.ll &&
/usr/local/opt/llvm/bin/llc -relocation-model=pic {NAME OF FILE}.ll >
{NAME OF FILE}.s && gcc -o  {NAME OF FILE}.exe  {NAME OF FILE}.s && ./
{NAME OF FILE}t.exe >  {NAME OF FILE}.out
```

Note: files ending with .mc can also be compiled and run by compati.exe

LRM (Language Reference Manual)

1. Lexical Elements
    a. Identifiers
        i. Identifiers can be any combination of letters, numbers and underscores. Compati leaves it up to the programmer to decide what convention to follow, only requiring identifiers start with a letter.
    b. Key words
        i. if
        ii. else
        iii. elseif
        iv. bool
        v. int
        vi. float
        vii. string
        viii. char
        ix. struct
        x. while
        xi. void
        xii. true
        xiii. false
        xiv. return
        xv. for
    c. Constants
        i. Constants can be considered immutable data types.
            1. Strings - string instance enclosed by double quotation marks. Strings are currently considered constants as in Compati strings are immutable.
            2. Integers - [1-9]+[0-9]*
            3. Floats - following the C standard
2. Data Types
    a. Primitive Types
        i. Int
            1. Integers, 32-bit
        ii. Float
            1. 8 byte double precision floating point numbers
        iii. Boolean
            1. true or false
        iv. String
            1. A sequence of ascii characters.
        v. Char
            1. represents 1 ascii character

b. Compound types
    i. Array []
        1. Arrays are sequences of same type elements, max length 2^31. There is currently no support for array literals (though an attempt was made and can be found by running *grep ArrayLit ** or some equivalent in the repository. Currently an array of a user specified size can only be declared, and then have elements assigned to it by index. Unfortunately anything beyond initialization of the array is faulty. The capacity of arrays in Compati is demonstrated below (as in this is really all you can do with them) :

```
int a;
int i[3];
i[0] = 1;
i[1] = 2;
i[2] = 3;
```

There was also an attempt on imposing semantic checking on arrays and is still present in the code, but the effectiveness is dubious.

    ii. Struct
        1. A struct is a composite data structure with member variables. Currently structs can only be local variables. Furthermore, structs must be defined at the top of any program, prior to any function or global variable declarations. Nested structs are not supported (but an attempt was made)

```
struct cat {
   int age;
   float weight;
}

int main ()
{
   struct cat meow;
   meow.age = 2;
   meow.weight = 22.2;
   print(meow.age);
   print(meow.weight);
}
```

3. Functions

    a. Functions in Compati are static structures consisting of a list of formal arguments and a list of statements. Functions must be declared and defined together (no hanging function signatures)
4. Expressions and Operators
    a. Expressions are composed of at least one operand, If there is an operator present then there must be two operands on either side of the operator (with the exception of negation and decrementation). Operands can be constants or variables accessed through an identifier.
5. Operator Precedence
    a. A general overview of operator precedence (in order of ascending order):
        i. else
        ii. equals (‘=’)
        iii. or
        iv. and
        v. ==, !=
        vi. comparison operators
        vii. plus, minus
        viii. times, divide
6. Expressions
    a. Identifier expressions: Because of the introduction of structs and arrays, an identifier could either be for a ‘simple’ case, i.e the case of a regular variable, or an identifier into an array or struct. Therefore an identifier itself is an expression, which can be broken down into the following cases (which will resolve into the correct value)
        i. Simple
            1. Just a regular variable, this will just retrieve the variable value
        ii. Struct
            1. This is when there is a dot operator following the identifier, indicating that this is a member access
                a. The design of the parser was to make the implementation of nested member access possible but there wasn’t enough time.
        iii. Index
            1. Accessing an array element at index i
            2. could be of form:
                a. Identifier expression [i] where i is an int
    b. Assignment
        i. Stores values to variables or struct members. Currently it seems possible to store literals into arrays via index, but that’s as far as it goes for array assignment.
    c. Logical AND expressions
    d. Logical OR expressions
    e. Arithmetic (in order of ascending precedence)
        i. left associative plus, minus

ii. left associative times, divide

f. Comparison operators

i. <, >, <= , >=, ==, !=

```
struct owner {
   int age;
}

struct cat {
   int age;
}

int main ()
{
   struct cat meow;
   struct owner dad;

   dad.age = 50;
   meow.age = 2;

   if ((dad.age) > (meow.age)) print(1);

}
```

7. Member Access
    a. Currently member access is nonassociative, via the dot operator (.). Making the dot operator left associative would be the first step in making nested member access possible.

```
struct x {
   int a;
}

int main ()
{
   struct x m;
    m.a = 2;
   return 0;
}
```

8. Statements
    a. if-else statements : must be of the form
        i. if (statement) statement else statement
        ii. if (statement) statement

```
if (true) print(42); else print(8);
if (true) print(42);
```

b. while-statements: must be of form
   i. while (expression) statement

```
while (i > 0) {
  print(i);
  i = i - 1;
 }
```

c. for statements: must be of the form
   i. for (expression ; expression ; expression) { statement}

```
for (i = 0 ; i < 5 ; i = i + 1) {
  print(i);
 }
```

   ii. for (; expression ;) { statement}

```
for ( ; i < 5; ) {
  print(i);
  i = i + 1;
 }
```

d. return: must be of the form
   i. return expression;
   ii. return;
e. Block: must be of the form
   i. { list of statements }
   ii. The block statement is mainly used to group together statements for functions, but has potential to be used in other ways.

Testing

I tested as I went along. All of the microc tests work. I have

1. Struct declaration test
2. Struct definition test
3. Struct assignment and access test
4. String test
5. Char test
6. Array test
7. Struct fail test (test semantic checking)
8. Struct fail test 2

9. Struct fail void test
10. Struct expr test
11. Struct comparison test

Please see the readme of https://github.com/shannonjin/compati for full file information and the bash commands for running these tests.

Timeline

Cstar

1. October 31st
    a. LRM and Parser submitted for C star
2. November 12
    a. Our group misses the hello world deadline. Language has no grammar.
        i. At this moment in time the group decided to use an alternative build system, from the makefile present in MicroC.
            1. This alternative build system ends up taking up too much time to set up
                a. Involved one of our group members trying to set up a vm for it etc. etc.
3. December 4
    a. I attempt to write a parser for C star (and in the process a grammar), efforts made at https://github.com/kkysen/cstar/tree/shannon and https://github.com/kkysen/cstar/tree/shannon2
        i. The approach we decided to take was divide and conquer, in retrospect not a good approach
4. December 17:
    a. I realize that trying to fit a parser into the pre-existing repo, with many features fleshed out in scanner, ast, sast, and with a ambiguous build system only known to the person who devised it
        i. I create an 'interpreter' of hello world cstar, basically the grammar of cstar https://github.com/shannonjin/cstar
            1. The hope is with this extremely pared down base, we could add features one by one
5. December 19:
    a. Group dissolves

Compati

1. Worked on adding structs to microc (with some level of semantic checking, wrote tests as I went along):
    a. December 25 - January 4 1:15 PM
2. January 4 1:15 PM thereafter
        i. Tried to add strings, chars, and arrays in one go

Lessons Learned

Objectively speaking, this project is one big failure wrapped around a few small successes. The amount of work at which I was able to accomplish was largely limited by the scarce amount of time I had after my group dissolved, considering that it occured right before finals week. The implosion of a group is never a beautiful thing, but I can only speak for myself in terms of what I think I could've done better.

Our group definitely started with too big of scope, and from this project I think I've learned the valuable lesson of learning to say no. The short explanation (one which pinpoints the shared culpability rather than any individual fault) is as follows. There was too much of a disconnect between our language architect and the rest of the group, and we all made the mistake of deciding on a language proposal centered largely around a heavily rust inspired language (a language which none but our language architect knew). A lot of time was lost trying to understand the language proposed, which ultimately led to the dissolution of our group.

As I was able to implement structs in about a week's time, I think that had I been more assertive and confident in proposing a directional shift to the group, perhaps we could've salvaged the project sooner. Ultimately I am very grateful for this experience. I think situational awareness is gained through experience. Even though this was very painful, I would rather go through this experience again rather than take a class I would easily sail through. Thank you so much Professor Edwards, for making experience possible for me.

Code Files in src directory

scanner.ml

```
(* Ocamllex scanner for Compati *)

{ open Compatiparse }

let digit = ['0' - '9']
let digits = digit+

rule token = parse
```

```
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '['      { LBRACKET }
| ']'      { RBRACKET }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| '.'      { DOT }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "bool"   { BOOL }
| "float"  { FLOAT }
| "void"   { VOID }
| "true"   { BLIT(true)  }
| "false"  { BLIT(false) }
| "struct" { STRUCT }
| "string" { STRING }
| "char"   { CHAR }
| digits as lxm { LITERAL(int_of_string lxm) }
| digits '.'  digit* ( ['e' 'E'] ['+' '-']? digits )? as lxm { FLIT(lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']*     as lxm { ID(lxm) }
| ''' (_ as ch) ''' { CHAR_LITERAL(ch) }
```

```
| '"' ([^ '"']* as str) '"' { STRING_LITERAL(str) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

compatiparse.mly

```
/* Ocamlyacc parser for Compati */

%{
open Ast
%}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA PLUS MINUS TIMES DIVIDE
ASSIGN
%token NOT EQ NEQ LT LEQ GT GEQ AND OR DOT LBRACKET RBRACKET CHAR STRING
%token RETURN IF ELSE FOR WHILE INT BOOL FLOAT VOID STRUCT
%token <int> LITERAL
%token <bool> BLIT
%token <char> CHAR_LITERAL
%token <string> STRING_LITERAL
%token <string> ID FLIT
%token EOF

%start program
%type <Ast.program> program

%nonassoc NOELSE
%nonassoc ELSE
%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE
%right NOT

%%
```

```
program:
  struct_decls decls EOF { ($1, fst $2, snd $2) }

struct_decls:
                              { [] }
  | struct_decls struct_defn { $2 :: $1 }


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

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

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

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

typ:
    INT   { Int   }
  | BOOL  { Bool  }
  | FLOAT { Float }
  | VOID  { Void  }
  | STRUCT ID { Struct($2) }
  | CHAR      { Char   }
  | STRING    { String }

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

```
vdecl:
   typ ID SEMI { ($1, $2) }
  | typ ID LBRACKET LITERAL RBRACKET SEMI {(Array($1, $4), $2)}


struct_defn:
  STRUCT ID LBRACE vdecl_list RBRACE
  { { sname = $2;
      members = List.rev $4 } }


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

stmt:
    expr SEMI                               { Expr $1               }
  | RETURN expr_opt SEMI                    { Return $2             }
  | LBRACE stmt_list RBRACE                 { Block(List.rev $2)    }
  | IF LPAREN expr RPAREN stmt %prec NOELSE { If($3, $5, Block([])) }
  | IF LPAREN expr RPAREN stmt ELSE stmt    { If($3, $5, $7)        }
  | FOR LPAREN expr_opt SEMI expr SEMI expr_opt RPAREN  stmt
                                            { For($3, $5, $7, $9)   }
  | WHILE LPAREN expr RPAREN stmt           { While($3, $5)         }

expr_opt:
    /* nothing */ { Noexpr }
  | expr          { $1 }

id_expr:
    ID                 { SimpleId($1) }
  | id_expr  DOT  ID   { MemberId($1, $3) }
  | id_expr  LBRACKET LITERAL RBRACKET { IndexId($1, $3) }

expr:
    LITERAL          { Literal($1)           }
  | FLIT               { Fliteral($1)          }
  | BLIT             { BoolLit($1)           }
  | CHAR_LITERAL     { CharLit($1)           }
  | STRING_LITERAL   { StringLit($1)         }
  | id_expr              { Id($1)                }
  | expr PLUS   expr { Binop($1, Add,   $3)   }
  | expr MINUS  expr { Binop($1, Sub,   $3)   }
```

```
  | expr TIMES  expr { Binop($1, Mult,  $3)   }
  | expr DIVIDE expr { Binop($1, Div,   $3)   }
  | expr EQ     expr { Binop($1, Equal, $3)   }
  | expr NEQ    expr { Binop($1, Neq,   $3)   }
  | expr LT     expr { Binop($1, Less,  $3)   }
  | expr LEQ    expr { Binop($1, Leq,   $3)   }
  | expr GT     expr { Binop($1, Greater, $3) }
  | expr GEQ    expr { Binop($1, Geq,   $3)   }
  | expr AND    expr { Binop($1, And,   $3)   }
  | expr OR     expr { Binop($1, Or,    $3)   }
  | MINUS expr %prec NOT { Unop(Neg, $2)      }
  | NOT expr             { Unop(Not, $2)          }
  | id_expr ASSIGN expr       { Assign($1, $3)    }
  | id_expr ASSIGN array_lit              { Assign($1, $3)              }
  | ID LPAREN args_opt RPAREN { Call($1, $3)  }
  | LPAREN expr RPAREN { $2                   }

array_lit:
  LBRACE args_opt RBRACE { ArrayLit($2) }

args_opt:
    /* nothing */ { [] }
  | args_list  { List.rev $1 }

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

ast.ml

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

type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq
|
          And | Or

type uop = Neg | Not

type typ = Int | Bool | Float | Void | Char | String | Struct of string |
Array of typ * int
(*struct that contains an identifier (the string)*)

type bind = typ * string

type id =
    SimpleId of string
  | MemberId of id * string
  | IndexId of id * int
  (*| ArrayId of string *)

and expr =
    Literal of int
```

```
  | Fliteral of string
  | CharLit of char
  | StringLit of string
  | ArrayLit of expr list
  | BoolLit of bool
  | Id of id
  | Binop of expr * op * expr
  | Unop of uop * expr
  | Assign of id * expr
  | Call of string * expr list
  | Noexpr

type stmt =
    Block of stmt list
  | Expr of expr
  | Return of expr
  | If of expr * stmt * stmt
  | For of expr * expr * expr * stmt
  | While of expr * stmt

type func_decl = {
    typ : typ;
    fname : string;
    formals : bind list;
    locals : bind list;
    body : stmt list;
  }

  type struct_defn = {
    sname : string;
    members : bind list;
  }

type program = struct_defn list * bind list * func_decl list

(* Pretty-printing functions *)

let string_of_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Equal -> "=="
```

```
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
  | Geq -> ">="
  | And -> "&&"
  | Or -> "||"

let string_of_uop = function
    Neg -> "-"
  | Not -> "!"

let rec string_of_id = function
  SimpleId(s) -> s
| MemberId(id, s) -> "Member(" ^ string_of_id id ^ ", " ^ s ^ ")"
| _ -> "Index id"

let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | Fliteral(l) -> l
  | CharLit(l) -> String.make 1 l
  | StringLit(l) -> l
  | ArrayLit(arr) -> "[" ^ (List.fold_left (fun lst elem -> lst ^ " " ^
string_of_expr elem ^ ",") "" arr) ^ "]"
  | BoolLit(true) -> "true"
  | BoolLit(false) -> "false"
  | Id(s) -> "Id(" ^ string_of_id s ^ ")"
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_expr e2
  | Unop(o, e) -> string_of_uop o ^ string_of_expr e
  | Assign(v, e) -> string_of_id v ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""

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, s, Block([])) -> "if (" ^ string_of_expr e ^ ")\n" ^
string_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ string_of_expr e ^ ")\n" ^
```

```
      string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ string_of_expr e1  ^ " ; " ^ string_of_expr e2 ^ " ; " ^
      string_of_expr e3  ^ ") " ^ string_of_stmt s
  | While(e, s) -> "while (" ^ string_of_expr e ^ ") " ^ string_of_stmt s

let rec string_of_typ = function
    Int -> "int"
  | Bool -> "bool"
  | Float -> "float"
  | Void -> "void"
  | Char -> "char"
  | Struct(name) -> name
  | String -> "String"
  | Array(t, _) -> (string_of_typ t) ^ "[]"

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

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  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_struct st =
  st.sname ^ " {" ^
  String.concat "" (List.map string_of_vdecl st.members) ^ "}\n"


let string_of_program (structs, vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_struct structs) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

semant.ml

```
(* Semantic checking for the Compati 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 (structs, globals, functions) =

  (* Verify a list of bindings has no void types or duplicate names or uses
an undefined struct *)
  let check_binds (kind : string) (binds : bind list) map =
    List.iter (function
      (Void, b) -> raise (Failure ("illegal void " ^ kind ^ " " ^ b))
  | (Struct(b), _) -> if not (StringMap.mem b map) then raise (Failure
("undefined struct type member " ^ b))
      | _ -> ()) binds;
```

```
    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

  (**** Check global variables ****)

  check_binds "global" globals StringMap.empty;

  (* Add struct name to symbol table *)
  let add_struct map st =
      let dup_err = "duplicate struct" ^ st.sname
      and make_err er = raise (Failure er)
      and n = st.sname
      in match st with
        _ when StringMap.mem n map -> make_err dup_err
      | _  -> check_binds n st.members map;
      StringMap.add n st.members map
in

let struct_defn_map = List.fold_left add_struct StringMap.empty structs in

  (**** Check functions ****)

  (* Collect function declarations for built-in functions: no bodies *)
  let built_in_decls =
    let add_bind map (name, ty) = StringMap.add name {
      typ = Void;
      fname = name;
      formals = [(ty, "x")];
      locals = []; body = [] } map
    in List.fold_left add_bind StringMap.empty [ ("print", Int);
                                                 ("printb", Bool);
                                                 ("printf", Float);
                                                 ("printbig", Int) ]
  in

  (* Add function name to symbol table *)
  let add_func map fd =
    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 *)
  let function_decls = List.fold_left add_func built_in_decls functions
  in

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

  let _ = find_func "main" in (* Ensure "main" is defined *)

  let check_function func =
    (* Make sure no formals or locals are void or duplicates *)
    check_binds "formal" func.formals struct_defn_map;
    check_binds "local" func.locals struct_defn_map;

    (* 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) *)
      match lvaluet with
       Array(t, _) ->
          (match rvaluet with
              Array(t', _) -> if (t == t') then lvaluet else raise (Failure
"f2")
            | _ -> if rvaluet == t then lvaluet else raise (Failure
("f1")))
     | _ ->
      if (String.contains (string_of_typ rvaluet) '[' ) then lvaluet
      else if lvaluet == rvaluet then lvaluet else raise (Failure err)
```

```
    in

    (* 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 (globals @ func.formals @ func.locals
)
    in
    (* Return a variable from our local symbol table *)
    let rec type_of_identifier = function
      SimpleId(s) ->
        let t  =
          (try StringMap.find s symbols
          with Not_found -> raise (Failure ("undeclared identifier " ^ s)))
        in t
      | MemberId(id, s) ->
        let st_type = type_of_identifier id in
        (match st_type with
           Struct(_)  ->  let t =
            (try List.find (fun (_, n') -> n' = s) (StringMap.find
(string_of_typ st_type) struct_defn_map)
           with Not_found -> raise (Failure("undeclared identifier " ^
(string_of_typ st_type) ^ "." ^ s)))
          in fst t
         | _ -> raise (Failure (s ^ " is not a struct type")))
      | IndexId(id, _) ->
        let container_type = type_of_identifier id in
        match container_type with
           Array(_) -> container_type (*To DO more types*)
         | _ -> raise (Failure "Not an array")


    (* Return a semantically-checked expression, i.e., with a type *)
    and expr  = function
        Literal  l -> (Int, SLiteral l)
      | Fliteral l -> (Float, SFliteral l)
      | CharLit l   -> (Char, SCharLit l)
      | StringLit l -> (String, SStringLit l)
      | ArrayLit elist  ->
        let tlist = List.map (fun a -> expr a) elist in
        if (List.length tlist) = 0
        then raise (Failure "Empty arrays not ok")
        else
```

```
          let typmatch t (ty, _) =
            if t == ty then
              ty
            else
              raise (Failure ("array elements must be same type"))
          and slist = List.map (fun e -> expr e) elist in
          (match slist with
              [] -> raise (Failure "can't have 0 element array literal")
            | _ ->
              let ty = List.fold_left typmatch (fst (List.hd slist)) slist
in
              (Array(ty, List.length elist), SArrayLit(slist)) )

         (* let allMatch = List.hd tlist in
          if List.for_all (fun t -> t = allMatch) tlist
          then (Array(allMatch, List.length tlist), SArrayLit(tlist)))
          else raise (Failure ("inconsistent types in array literal "
                               ^ string_of_expr ex))  *)
      | BoolLit l  -> (Bool, SBoolLit l)
      | Noexpr     -> (Void, SNoexpr)
      | Id s       -> (type_of_identifier s, SId s)
      | Assign(var, e) as ex ->
          let lt = type_of_identifier var
          and (rt, e') = 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')))
      | Unop(op, e) as ex ->
          let (t, e') = expr e in
          let ty = match op with
            Neg when t = Int || t = Float -> t
          | Not when t = Bool -> Bool
          | _ -> raise (Failure ("illegal unary operator " ^
                                 string_of_uop op ^ string_of_typ t ^
                                 " in " ^ string_of_expr ex))
          in (ty, SUnop(op, (t, e')))
      | Binop(e1, op, e2) as e ->
          let (t1, e1') = expr e1
          and (t2, e2') = expr e2 in
          (* All binary operators require operands of the same type *)
          let same = t1 = t2 in
          (* Determine expression type based on operator and operand types
*)
```

```
          let ty = match op with
            Add | Sub | Mult | Div when same && t1 = Int   -> Int
          | Add | Sub | Mult | Div when same && t1 = Float -> Float
          | Equal | Neq            when same               -> Bool
          | Less | Leq | Greater | Geq
                       when same && (t1 = Int || t1 = Float) -> Bool
          | And | Or when same && t1 = Bool -> Bool
          | _ -> raise (
            Failure ("illegal binary operator " ^
                     string_of_typ t1 ^ " " ^ string_of_op op ^ " " ^
                     string_of_typ t2 ^ " in " ^ string_of_expr e))
          in (ty, SBinop((t1, e1'), op, (t2, e2')))
      | Call(fname, args) as call ->
          let fd = find_func fname in
          let param_length = List.length fd.formals in
          if List.length args != param_length then
            raise (Failure ("expecting " ^ string_of_int param_length ^
                            " arguments in " ^ string_of_expr call))
          else let check_call (ft, _) e =
            let (et, e') = 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
          in (fd.typ, SCall(fname, args'))
    in

    let check_bool_expr e =
      let (t', e') = expr e
      and err = "expected Boolean expression in " ^ string_of_expr e
      in if t' != Bool then raise (Failure err) else (t', e')
    in

    (* Return a semantically-checked statement i.e. containing sexprs *)
    let rec check_stmt = function
        Expr e -> SExpr (expr e)
      | If(p, b1, b2) -> SIf(check_bool_expr p, check_stmt b1, check_stmt
b2)
      | For(e1, e2, e3, st) ->
        SFor(expr e1, check_bool_expr e2, expr e3, check_stmt st)
      | While(p, s) -> SWhile(check_bool_expr p, check_stmt s)
      | Return e -> let (t, e') = expr e in
```

```
        if t = func.typ then SReturn (t, e')
        else raise (
        Failure ("return gives " ^ string_of_typ t ^ " expected " ^
               string_of_typ func.typ ^ " in " ^ string_of_expr e))

          (* A block is correct if each statement is correct and nothing
             follows any Return statement.  Nested blocks are flattened. *)
      | Block sl ->
          let rec check_stmt_list = function
              [Return _ as s] -> [check_stmt s]
            | Return _ :: _   -> raise (Failure "nothing may follow a
return")
            | Block sl :: ss  -> check_stmt_list (sl @ ss) (* Flatten
blocks *)
            | s :: ss         -> check_stmt s :: check_stmt_list ss
            | []              -> []
          in SBlock(check_stmt_list sl)

    in (* body of check_function *)
    { styp = func.typ;
      sfname = func.fname;
      sformals = func.formals;
      slocals  = func.locals;
      sbody = match check_stmt (Block func.body) with
      SBlock(sl) -> sl
      | _ -> raise (Failure ("internal error: block didn't become a
block?"))
    }
in (struct_defn_map, globals, List.map check_function functions)
```

sast.ml

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

open Ast
module StringMap = Map.Make(String)

type sexpr = typ * sx
and sx =
    SLiteral of int
  | SFliteral of string
  | SBoolLit of bool
  | SCharLit of char
  | SStringLit of string
  | SId of id
  | SBinop of sexpr * op * sexpr
  | SUnop of uop * sexpr
  | SAssign of id * sexpr
  | SArrayLit of sexpr list
  | SCall of string * sexpr list
  | SNoexpr

type sstmt =
```

```
    SBlock of sstmt list
  | SExpr of sexpr
  | SReturn of sexpr
  | SIf of sexpr * sstmt * sstmt
  | SFor of sexpr * sexpr * sexpr * sstmt
  | SWhile of sexpr * sstmt

type sfunc_decl = {
    styp : typ;
    sfname : string;
    sformals : bind list;
    slocals : bind list;
    sbody : sstmt list;
  }

type sprogram = bind list StringMap.t * bind list * sfunc_decl list

(* 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"
  | SFliteral(l) -> l
  | SCharLit(l) -> String.make 1 l
  | SStringLit(l) -> l
  | SId(s) -> "Id(" ^ string_of_id s ^ ")"
  | SBinop(e1, o, e2) ->
      string_of_sexpr e1 ^ " " ^ string_of_op o ^ " " ^ string_of_sexpr e2
  | SUnop(o, e) -> string_of_uop o ^ string_of_sexpr e
  | SAssign(v, e) -> string_of_id v ^ " = " ^ string_of_sexpr e
  | SArrayLit(arr) -> "[" ^ (List.fold_left (fun lst elem -> lst ^ " " ^
string_of_sexpr elem ^ ",") "" arr) ^ "]"
  | SCall(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_sexpr el) ^ ")"
  | SNoexpr -> ""
                          ) ^ ")"

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, s, SBlock([])) ->
      "if (" ^ string_of_sexpr e ^ ")\n" ^ string_of_sstmt s
  | SIf(e, s1, s2) ->  "if (" ^ string_of_sexpr e ^ ")\n" ^
      string_of_sstmt s1 ^ "else\n" ^ string_of_sstmt s2
  | SFor(e1, e2, e3, s) ->
      "for (" ^ string_of_sexpr e1  ^ " ; " ^ string_of_sexpr e2 ^ " ; " ^
      string_of_sexpr e3  ^ ") " ^ string_of_sstmt s
  | SWhile(e, s) -> "while (" ^ string_of_sexpr e ^ ") " ^ string_of_sstmt
s

let string_of_sfdecl fdecl =
  string_of_typ fdecl.styp ^ " " ^
  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_struct st =
  fst st ^ " {" ^
  String.concat "" (List.map string_of_vdecl (snd st)) ^ "}\n"

let string_of_sprogram (structs, vars, funcs) =
  String.concat "" (List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "\n" (List.map string_of_struct (StringMap.bindings
structs)) ^ "\n" ^
  String.concat "\n" (List.map string_of_sfdecl funcs)
```

codegen.ml

```
(* Code 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)

let debug message =
  if false
  then prerr_endline message
```

```
  else ()

(* translate : Sast.program -> Llvm.module *)
let translate (structs, globals, 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 "Compati" 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
  and float_t    = L.double_type context
  and void_t     = L.void_type   context
  and string_t   = L.pointer_type (L.i8_type context)
  in

  let struct_cache = Hashtbl.create 10 in

  (* Return the LLVM type for a MicroC type *)
  let rec ltype_of_typ = function
      A.Int   -> i32_t
    | A.Bool  -> i1_t
    | A.Float -> float_t
    | A.Void  -> void_t
    | A.String -> string_t
    | A.Char -> i8_t
    | A.Array(t, s) -> L.array_type (ltype_of_typ t) s
    | A.Struct(name) ->
      let t =
        (try Hashtbl.find struct_cache name
         with Not_found ->
           let struct_t = L.named_struct_type context name in
           Hashtbl.add struct_cache name struct_t;
           L.struct_set_body struct_t (Array.of_list (List.map ltype_of_typ
(List.map fst (StringMap.find name structs)
           ))) false;
           struct_t) in
      L.pointer_type 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 = match t with
          A.Float -> L.const_float (ltype_of_typ t) 0.0
        | _ -> 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 =
      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

  let printbig_t : L.lltype =
      L.function_type i32_t [| i32_t |] in
  let printbig_func : L.llvalue =
      L.declare_function "printbig" printbig_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_decl) StringMap.t =
    let function_decl m fdecl =
      let name = fdecl.sfname
      and formal_types =
      Array.of_list (List.map (fun (t,_) -> ltype_of_typ t) fdecl.sformals)
      in let ftype = L.function_type (ltype_of_typ fdecl.styp) formal_types
in
      StringMap.add name (L.define_function name ftype the_module, fdecl) m
in
    List.fold_left function_decl StringMap.empty functions in

  (* Fill in the body of the given function *)
  let build_function_body fdecl =
    let (the_function, _) = StringMap.find fdecl.sfname function_decls in
    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
    (*and str_format_str = L.build_global_stringptr "%s\n" "fmt" builder *)
    and float_format_str = L.build_global_stringptr "%g\n" "fmt" builder in
```

```
    let llstore lval laddr builder =
      let ptr = L.build_pointercast laddr (L.pointer_type (L.type_of lval))
"" builder in
      let store_inst = (L.build_store lval ptr builder) in
      debug ((L.string_of_llvalue store_inst));
      ()
    in

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

      (* Allocate space for any locally declared variables and add the
       * resulting registers to our map*)
      and add_local m (t, n) =
      let local_var = match t with
      A.Struct(n) ->
        let struct_ptr_t = ltype_of_typ t in
        let struct_t = L.element_type struct_ptr_t in
        let struct_ptr = L.build_alloca struct_ptr_t n builder in
        let struct_val = L.build_malloc struct_t n builder in
        ignore (llstore struct_val struct_ptr builder);
        struct_ptr
     |_ -> L.build_alloca (ltype_of_typ t) n builder

    in StringMap.add n local_var m
      in

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

   (* let find_index exp = match exp
    with A.Literal(i) -> L.const_int i32_t i
```

```
    | _ -> raise(Failure("must be numerical index"))
  in
     Return the value for a variable or formal argument.
       Check local names first, then global names *)
    let rec lookup x =
      begin match x with
      A.SimpleId(s) ->
        let return =
          (try StringMap.find s local_vars
                   with Not_found -> StringMap.find s global_vars)
        in return
    | A.IndexId(id, i) ->
      let indices = [|L.const_int i32_t 0; L.const_int i32_t i|] in
      let ref = L.build_gep (lookup id) indices "" builder in
      (L.build_load ref "" builder)
    | A.MemberId(id, field_id) ->
      let rec find_index_of field_id l =
        match l with
        [] -> raise (Failure ("undeclared field " ^ field_id))
      | hd :: tl -> if field_id = (snd hd)
                    then 0
                    else 1 + find_index_of field_id tl
      in
      let struct_pp = lookup id in
      let struct_addr = L.build_load struct_pp "" builder in
      let the_struct = L.build_load struct_addr "" builder in
      let struct_tname_opt = L.struct_name (L.type_of the_struct) in
      (match struct_tname_opt with
         None -> raise (Failure ("expected struct, found tuple"))
       | Some(struct_tname) ->
          let fields = StringMap.find struct_tname structs in
          let idx = find_index_of field_id fields in
          let addr = L.build_struct_gep struct_addr idx ("."^field_id)
builder in
          addr)
  end in
  let rec 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)
    | SFliteral l -> L.const_float_of_string float_t l
    | SNoexpr     -> L.const_int i32_t 0
    | SId id       -> L.build_load (lookup id) "" builder
    | SStringLit l -> L.build_global_stringptr l "str" builder
```

```
      | SCharLit l -> L.const_int i8_t (Char.code l)
      | SArrayLit elist ->
      if List.length elist == 0
        then raise (Failure "Array cannot be empty")
      else
        let (elements) = List.map (fun a -> expr builder a) elist in
        let ty =  ltype_of_typ (fst (List.hd elist)) in
        let ptr = L.build_array_malloc
        ty
        (L.const_int i32_t 1)
        ""
        builder in
        ignore (List.fold_left
                (fun i elem ->
                  let ind = L.const_int i32_t i in
                  let eptr = L.build_gep ptr [|ind|] "" builder in
                  llstore elem eptr builder;
                  i+1
                ) 0 elements); (ptr)
      | SAssign (s, e) ->
        let e' = expr builder e in
        (match s with
        A.IndexId(id, index) ->
          let i' = L.const_int i32_t index in
          let ex' = expr builder e in
          let indices = [|L.const_int i32_t 0; i'|] in
          let ref = L.build_gep (lookup id) indices "" builder in
          ignore(L.build_store ex' ref builder); ex'
        | _ ->  ignore(L.build_store e' (lookup s) builder); e' )
      | SBinop ((A.Float,_ ) as e1, op, e2) ->
        let e1' = expr builder e1
        and e2' = expr builder e2 in
        (match op with
          A.Add     -> L.build_fadd
        | A.Sub     -> L.build_fsub
        | A.Mult    -> L.build_fmul
        | A.Div     -> L.build_fdiv
        | A.Equal   -> L.build_fcmp L.Fcmp.Oeq
        | A.Neq     -> L.build_fcmp L.Fcmp.One
        | A.Less    -> L.build_fcmp L.Fcmp.Olt
        | A.Leq     -> L.build_fcmp L.Fcmp.Ole
        | A.Greater -> L.build_fcmp L.Fcmp.Ogt
        | A.Geq     -> L.build_fcmp L.Fcmp.Oge
```

```
        | A.And | A.Or ->
            raise (Failure "internal error: semant should have rejected
and/or on float")
        ) e1' e2' "tmp" builder
      | SBinop (e1, op, e2) ->
        let e1' = expr builder e1
        and e2' = expr builder e2 in
        (match op with
          A.Add     -> L.build_add
        | A.Sub     -> L.build_sub
        | A.Mult    -> L.build_mul
          | A.Div     -> L.build_sdiv
        | A.And     -> L.build_and
        | A.Or      -> L.build_or
        | A.Equal   -> L.build_icmp L.Icmp.Eq
        | A.Neq     -> L.build_icmp L.Icmp.Ne
        | A.Less    -> L.build_icmp L.Icmp.Slt
        | A.Leq     -> L.build_icmp L.Icmp.Sle
        | A.Greater -> L.build_icmp L.Icmp.Sgt
        | A.Geq     -> L.build_icmp L.Icmp.Sge
        ) e1' e2' "tmp" builder
      | SUnop(op, ((t, _) as e)) ->
          let e' = expr builder e in
        (match op with
          A.Neg when t = A.Float -> L.build_fneg
        | A.Neg                  -> L.build_neg
          | A.Not                  -> L.build_not) e' "tmp" builder
      | SCall ("print", [e]) | SCall ("printb", [e]) ->
        L.build_call printf_func [| int_format_str ; (expr builder e) |]
          "printf" builder
      | SCall ("printbig", [e]) ->
        L.build_call printbig_func [| (expr builder e) |] "printbig"
builder
      | SCall ("printf", [e]) ->
        L.build_call printf_func [| float_format_str ; (expr builder e) |]
          "printf" builder
      | SCall (f, args) ->
         let (fdef, fdecl) = StringMap.find f function_decls in
       let llargs = List.rev (List.map (expr builder) (List.rev args)) in
       let result = (match fdecl.styp with
                        A.Void -> ""
                      | _ -> f ^ "_result") in
         L.build_call fdef (Array.of_list llargs) result builder
```

```
    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 stmt builder = function
      SBlock sl -> List.fold_left stmt builder sl
      | SExpr e -> ignore(expr builder e); builder
      | SReturn e -> ignore(match fdecl.styp with
                              (* Special "return nothing" instr *)
                              A.Void -> L.build_ret_void builder
                              (* Build return statement *)
                            | _ -> L.build_ret (expr builder e) builder );
                     builder
      | SIf (predicate, then_stmt, else_stmt) ->
        let bool_val = expr builder predicate in
       let merge_bb = L.append_block context "merge" the_function in
        let build_br_merge = L.build_br merge_bb in (* partial function *)

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

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

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

      | SWhile (predicate, body) ->
        let pred_bb = L.append_block context "while" the_function in
```

```
      ignore(L.build_br pred_bb builder);

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

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

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

    (* Implement for loops as while loops *)
    | SFor (e1, e2, e3, body) -> stmt builder
        ( SBlock [SExpr e1 ; SWhile (e2, SBlock [body ; SExpr e3]) ] )
  in

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

  (* Add a return if the last block falls off the end *)
  add_terminal builder (match fdecl.styp with
      A.Void -> L.build_ret_void
    | A.Float -> L.build_ret (L.const_float float_t 0.0)
    | t -> L.build_ret (L.const_int (ltype_of_typ t) 0))
in

List.iter build_function_body functions;
the_module
```

dune

```
(ocamllex scanner)      ; name of scanner .mll file
(ocamlyacc compatiparse) ; name of parser .mly file
(executable
 (name compati)          ; build target is name.exe
 (libraries
  llvm
  llvm.analysis
  ; add any other libraries you might need here
 )
)
```

My test files

string-test.compati

```
int main ()
{
    string hello;
    hello = "Hello world!";
    return 0;
}
```

```
1  struct test {
2      int i;
3      int zig;
4  }
5
6  int main ()
7  {
8      struct test wow;
9      struct test cow;
10     wow.i = 18;
11     wow.zig = 1;
12
13     cow.i = wow.i;
14     cow.zig = 1;
15
16     print(wow.i);
17     print(cow.i);
18     print(wow.zig);
19     print(cow.zig);
20     return 0;
21 }
```

struct-assign-test.compati

struct-comp.compati

```c
struct owner {
    int age;
}

struct cat {
    int age;
}

int main ()
{
    struct cat meow;
    struct owner dad;

    dad.age = 50;
    meow.age = 2;

    if ((dad.age) > (meow.age)) print(1);

    return 0;

}
```

struct-decl-test.compati

```
1   struct test {
2       int i;
3   }
4
5   int main ()
6   {
7       struct test wow;
8       print(17);
9       return 0;
10  }
```

struct-defn-test.compati

```
1   struct test {
2       int i;
3   }
4
5   int main ()
6   {
7       struct test wow;
8       wow.i = 1;
9       print(wow.i);
10      return 0;
11  }
```

struct-test-fail-void.compati

```
1  struct dog {
2      int i;
3      int z;
4      void k;
5  }
6  int main ()
7  {
8      struct dog w;
9  }
```

struct-test-fail2.compati

9 lines (9 sloc) | 83 Bytes

```
1  struct dog {
2      int i;
3      int z;
4      int z;
5  }
6  int main ()
7  {
8      struct dog w;
9  }
```

struct-test-void.compati

```
1  struct dog {
2      int i;
3      int z;
4      void k;
5  }
6  int main ()
7  {
8      struct dog w;
9  }
```

char-test.compati

6 lines (6 sloc) | 55 Bytes

```
1  int main ()
2  {
3      char h;
4      h = 'h';
5      return 0;
6  }
```

array-test1.compati

12 lines (11 sloc) | 126 Bytes

```
1   int main ()
2   {
3       int a;
4       int i[3];
5       i[0] = 1;
6       i[1] = 2;
7       i[2] = 3;
8       a = i[0];
9
10      print(a);
11      return 0;
12  }
```