TABLE PROGRAMMING
LANGUAGE
By Hamza Jazmati
Introduction:

Table Programming Language, or TPL, is based on the C programming language, with the main difference that TPL supports an extra data type called *Table*. On the contrary, TPL does not include all the features supported by C programming language, only a limited subset described in the rest of this document.

Sample program:

Note: The following program does not work in the current version:

In this program, I developed an inner join with the limited set of features that the language has for tables.
Language Reference Manual:

1. Lexical Conventions

Just like C, which TPL is based on, there are six types of tokens:

- Identifiers
- Keywords
- Constants
• Strings
• Expression operators
• Other separators.

White space including tabs, newlines, blanks, and comments only purpose is to separate tokens. Otherwise, they are ignored by the compiler.

1.1 Comments:
Two types of comments will be supported:
• The characters /* introduce a comment, which terminates with the characters */
  This type is borrowed from C.
• The characters // introduce a comment, which terminates at the end of the line. This is borrowed from C++.

1.2 Identifiers:
Identifiers can be described as sequence of characters that start with a letter (lower or upper case) and the rest of it can be a combination of letters and numbers. Identifiers are case sensitive.

1.3 Keywords:
The following list includes all the keywords in TPL. These keywords cannot be used as identifiers.

array
bool
else
float
int
if
string
table
while

1.3 Constants:
These constants include:
1.3.1 Integer Constants:
An integer constant is a sequence of digits. An integer is always taken to be decimal.

1.3.2 Floating Constants:
A floating constant consists of an integer part, a decimal point, and a fraction part. The integer and fraction parts both consist of a sequence of digits. The fraction or the integer part can be missing.
1.3.3 String
A string is a sequence of characters surrounded by double quotes “”.

2. Conversions:
There are no supported implicit conversion in TPL is between any two types.

3. Expressions:
3.1 Primary Expressions:
3.1.1 Identifier:
An identifier is a primary expression, provided it has been suitably declared. Its type is specified by its declaration.

3.1.2 constant:
A decimal that can be represented with int, or a floating point constant represented as float. String is also another type of constant.

3.1.3 ( expression )
A parenthesized expression is a primary expression whose type and value are identical to those of the unadorned expression.

3.2 Unary operators:
Expressions with unary operators group right-to-left.

3.2.1 – Expression:
The result is the negative of the expression, and has the same type. The type of the expression must be int or float.

3.2.2 ! expression:
The result of the logical negation operator ! is 1 if the value of the expression is 0, 0 if the value of the expression is 1. The type of the result is bool. This operator is applicable only to bool.

3.3 Multiplicative Operators:
The multiplicative operators *, /, and % group left-to-right.

3.3.1 expression * expression:
The binary * operator indicates multiplication. If both operands are int, the result is int; if one is int and one is float, the former is converted to float, and the result is float; if both are float, the result is float. No other combinations are allowed.

3.3.2 expression / expression:
The binary / operator indicates division. The same type considerations as for multiplication apply.
3.3.3 expression % expression:
The binary % operator yields the remainder from the division of the first expression by the second. Both operands must be int, and the result is int. In the current implementation, the remainder has the same sign as the dividend.

3.4 Additives Operators:
The additive operators + and – group left-to-right.

3.4.1 expression + expression:
The result is the sum of the expressions. If both operands are int, the result is int. If both are float, the result is float. If one is int and one is float, the former is converted to float and the result is float. No other type combinations are allowed.

3.4.2 expression - expression:
The result is the difference of the operands. If both operands are int, or float the same type considerations as for + apply.

3.5 Rational Operators:
The relational operators group left-to-right, but this fact is not very useful; “a<b<c” does not mean what it seems to.

3.5.1 expression < expression
3.5.2 expression <= expression
3.5.3 expression > expression
3.5.4 expression >= expression
The operators < (less than), > (greater than), <= (less than or equal to) and >= (greater than or equal to) all yield 0 if the specified relation is false and 1 if it is true.

3.6 Equality Operators:
3.6.1 expression == expression
3.6.2 expression != expression
The == (equal to) and the != (not equal to) operators are exactly analogous to the relational operators except for their lower precedence. (Thus “a<b == c<d” is 1 whenever a<b and c<d have the same truth-value).

3.7 expression || expression:
3.8 expression && expression:
3.9 Assignment Operators:
Ivalue = expression.
The value of the expression replaces that of the object referred to by the lvalue. The operands need not have the same type, but both must be int, or float.
4. Statements:

4.1 Expression Statement:
Most statements are expression statements, which have the form

expression ;

4.2 Compound Statement:
So that several statements can be used where one is expected, the compound statement is provided:

\[
\text{compound-statement}: \{ \text{statement-list} \}
\]

4.3 Conditional Statement:
The two forms of the conditional statement are

if ( expression ) statement

if ( expression ) statement else statement

In both cases the expression is evaluated and if it is 1, the first substatement is executed. In the second case the second substatement is executed if the expression is 0. As usual the “else” ambiguity is resolved by connecting an else with the last encountered elseless if.

4.4 While Statement:
The while statement has the form

while ( expression ) statement  The substatement is executed repeatedly so long as the value of the expression remains 1. The test takes place before each execution of the statement.

5. Lists and Tables:
The two main differences between C/C++ and TPL are the ways lists and tables are handled.

5.1 Lists:
lists are a collection of objects that do not have a fixed length. In TPL, a list can have only one type. Lists can be of any of the types: int, float, or string.

5.1.1 Declaring Lists:
Declaring a list can be as follows:

list string header = {“Hamza”, “Jazmati”, “Edward”, “Snowden”}
list int primes = {2,3,5,7,11}

5.1.2 Getters/Setters:
To obtain a value from the list, we use the following syntax:

primes[3]

This will get us the fourth element of the list, which is 7.
primes[4] = 13
This will set the value of the fourth element of the list to 13.

5.2.3 Appending:
To append a new value to the end of the list, we do as follows:
primes.append(17)
This statement should return the index of the newly added item, in this case, 5.

5.2 Tables:
Tables are two dimensional lists where the header is a string list and each column is a list of a specific type.

5.2.1 Declaring a Table:
table mytable = {“First_Name”, “Last_Name”, “Grade”} {string, string,int}
Column names cannot be the used more than once in the same table.

5.2.2 Getters/Setters:
To get the value of a specific cell in the table, we do as follows:
mytable[“First_Name”][0]
To get a specific column, we can use:
mytable. “First_Name”
To set a value of a specific cell in the table, we do as follows:
mytable[“First_Name”][0] = “John”

5.2.3 Appending Rows:
To append a row to the table, we use the following syntax:
mytable.appendrow (“Hamza”, “Jazmati”, 100)

5.2.3 Generate a column:
To generate a new column from other columns in the table, we use the following syntax:
mytable.newcolumn (“Grade_Out_Of_Ten” , “Grade”/10 )

5.2.4 Sorting:
To sort a column according to a column, we use the following syntax:
mytable.sort( “Grade”, asc)
mytable.sort( “Grade”, desc)
asc indicates that the sorting is ascending. dsec indicates descending.

5.2.5 Getting Row Count:
To get row count of the table, mytable.rowcount.

Architectural Design:
The main modules are:
The scanner: takes the program text and converts it into tokens.
The parser:
Test Plan:
For testing, I used the MicroC testing suite. Test cases were added before developing a feature to guide the development and eliminating any development biases. The following test cases were added:

<table>
<thead>
<tr>
<th>Test Case Name</th>
<th>Test Case Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>test-float-print-1</td>
<td>Tests printing multiple float numbers</td>
</tr>
<tr>
<td>test-float-arth-1</td>
<td>Tests simple operations on float numbers</td>
</tr>
<tr>
<td>test-float-declaration</td>
<td>Tests declaring float numbers</td>
</tr>
</tbody>
</table>

Lessons Learned:
Although this project has been a major failure for me, I did learn a lot of lessons:
- If you work full time, switch to CVN.
- Doing the project alone is not viable unless you have a light course load.
- Being intimidated by how hard the project is, is the best way to waste weeks.
- Start small and use MicroC.
- Be conservative with the requirements as much as you can.

Current Status:
Many features were not in the final build due to time constrains. The following features are currently supported:
- SAST
- Float Support: float literals, float printing
- Beginnings of String Support.
However, it is not compiling now due to some issues that I could not fix in a timely manner.

Code:
(*) Ocamllex scanner for TPL *)
{ open Parser }

(* Add float and good string support *)
rule token = parse
[' ' 't' 'r' 'n'] { token lexbuf } (* Whitespace *)
"/*" { multilinecomment lexbuf } (* Comments *)
"//" { singlelinecomment lexbuf } (* Comments *)
""{" { STRING (str) } ""{" as strl literal }""{" STRLITERAL(strl literal) }
"([0-9]+)\*[.][0-9]+ as l } { FLOATLITERAL()}\n| ' (' { LPAREN } |
| ') ' { RPAREN } |
| '{ ' { LBRACE } |
| '}' { RBRACE } |
| '[' { LBRACKET } |
| ']' { RBRACKET } |
| ';' { SEMI } |
| '.' { DOT } |
| ',' { COMMA } |
| '+' { PLUS } |
| '-' { MINUS } |
| '*' { TIMES } |
| '/' { DIVIDE } |
| '=' { ASSIGN } |
| "=" { EQ } |
| "!=" { NEQ } |
| '<' { LT } |
| '<=' { LEQ } |
| '>=' { GT } |
| '>' { GEQ } |
| '&&' { AND } |
| '||' { OR } |
| '!' { NOT } |
| 'if' { IF } |
| "else" { ELSE } |
| "for" { FOR } |
| "while" { WHILE } |
| "return" { RETURN } |
| "array" { ARRAY } |
| "int" { INT } |
| "string" { STRING } |
| "float" { FLOAT } |
| "table" { TABLE } |
| "bool" { BOOL } |
| "void" { VOID } |
| "true" { TRUE } |
| "false" { FALSE } |
\n[0-9]+ \* as lxm { LITERAL(int_of_string lxm) }
[\a-zA-Z][\a-zA-Z\_0-9] as lxm { ID(lxm) }
\nand multilinecomment = parse
"/*" { token lexbuf }
| _ { multilinecomment lexbuf } |

and singlelinecomment = parse
\n[\r\n] { token lexbuf }
(* Top-level of the TPL compiler: scan & parse the input, check the resulting AST, generate LLVM IR, and dump the module *)

type action = Ast | LLVM_IR | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [("-a", Ast); (* Print the AST only *)
    ("-l", LLVM_IR); (* Generate LLVM, don't check *)
    ("-c", Compile)] (* Generate, check LLVM IR *)
  else Compile in

  let lexbuf = Lexing.from_channel stdin in
  let ast = Parser.program Scanner.token lexbuf in
  let sast =
    let tmp = Semant.check_vardecls (fst ast) in
    semant.check_functions tmp (fst ast) (snd ast) in

  match action with
    | Ast -> print_string (Ast.string_of_program ast)
    | LLVM_IR -> print_string (Llvm.string_of_llmodule (Codegen.translate sast))
    | Compile -> let m = Codegen.translate sast in
    Llvm_analysis.assert_valid_module m;
    print_string (Llvm.string_of_llmodule m);
Parser:

```plaintext
/* Ocaml yacc parser for TPL */

open Ast

%{

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACKET RBRACKET COMMA DOT
%token PLUS MINUS TIMES DIVIDE ASSIGN NOT
%token EQ NEQ LT LEQ GT GEQ TRUE FALSE AND OR
%token RETURN IF ELSE FOR WHILE INT BOOL VOID STRING FLOAT ARRAY TABLE
%token <int> LITERAL
%token <string> STRLITERAL
%token <string> FLOATLITERAL
%token <string> ID
%token EOF

%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 NEG

%start program
%type <Ast.program> program

%%

program:
  decls EOF { $1 }

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

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

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

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

the_type:
  basic_types { The_type ($1) }
```
basic_types:
  INT { Int }
  | BOOL { Bool }
  | VOID { Void }
  | FLOAT { Float }
  | STRING { String }
  /* ADD THE REST OF PATTERN */
  | ARRAY {Array}
  | TABLE {Table}

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

vdecl:
  the_type ID SEMI { ($1, $2) }

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

stmt:
  expr SEMI { Expr $1 }
  | RETURN SEMI { Return Noexpr }
  | RETURN expr 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 }
expr:
  ID LBRACKET LITERAL RBRACKET {IndexValue($1, $3)}
  | LITERAL {Literal($1)}
  | STRLITERAL {StrLiteral($1)}
  | FLOATLITERAL {FloatLiteral (float_of_string $1)}
  | TRUE {BoolLit(true)}
  | FALSE {BoolLit(false)}
  | ID {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 NEG {Unop(Neg, $2)}
  | NOT expr {Unop(Not, $2)}
  | ID ASSIGN expr {Assign($1, $3)}
  | ID LPAREN actuals_opt RPAREN {Call($1, $3)}
  | LPAREN expr RPAREN {$2}

actuals_opt:
  /* nothing */ {[]}
  | actuals_list {List.rev $1}

actuals_list:
  expr {[$1]}
  | actuals_list COMMA expr {$3 :: $1}
(* TPL
 * 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 basic_types = Int | Bool | Void | Table | Float | String |

Array of basic_types * int

type the_types = Thetype of basic_types

(* HJ: ADD MAIN FUNCTIONS HERE *)

(type expr =

Literal of int

| Floatliteral of float
| Strliteral of string
| BoolLit of bool
| Id of string

(* HJ: ArrayIndexValue *)
| ArrayIndexValue of string * expr
| Binop of expr * op * expr
| Unop of uop * expr
| Assign of string * 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 var_formal = VFormal of the_types * string

(type var_local = VLocal of the_types * string

(type var_decl = the_types * string

(type func_decl = {

typ : the_types;
fname : string;
formals : var_formal list;
locals : var_local list;
body : stmt list;
}

(type program = var_decl list * func_decl list


let string_of_op = function
  Add -> "+
  Sub -> "-
  Mult -> "*
  Div -> "/
  Equal -> "==
  Neq -> "!=
  Less -> "<
  Leq -> "<=
  Greater -> ">
  Geq -> ">="
  Or -> "||"

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

let rec string_of_expr = function
  Literal(l) -> string_of_int l
  FloatLiteral(f) -> string_of_float f
  StrLiteral(s) -> s
  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
  Unop(o, e) -> string_of_uop o ^ string_of_expr e
  Assign(v, e) -> v ^ " = " ^ string_of_expr e
  Call(f, e1) ->
    f ^ "(" ^ String.concat ", " (List.map string_of_expr e1) ^ ")"
  Noexpr -> ""

(* HJ: Remove Hardcoded Value *)
 ArrayIndexValue(id, e) -> 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"
  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 string_of_basic_types = function
  Int -> "int"
  Bool -> "bool"
  Void -> "void"
  Float -> "float"
  String -> "string"

(* HJ: Modify array and Table *)
 Array(_,_) -> "array"
 Table -> "table"

let string_of_typ = function
  Thetype(t) -> (string_of_basic_types t)
let string_of_var_formal = function
  VFormal (t, id) -> string_of_typ t ^ " " ^ id ^ "\n"

let string_of_var_local = function
  VLocal (t, id) -> (string_of_typ t) ^ " " ^ id ^ "\n"

let string_of_vdecl = function
  (t, id) -> string_of_typ t ^ " " ^ id ^ "\n"

let string_of_fdecl fdecl =
  string_of_typ fdecl.typ ^ " " ^
  fdecl.fname ^ "(" ^ String.concat ", " (List.map string_of_var_formal fdecl.formals) ^ "\n")\n"

String.concat "(List.map string_of_var_local fdecl.locals) ^
String.concat "(List.map string_of_stmt fdecl.body) ^
"

let string_of_program (vars, funcs) =
  String.concat "(List.map string_of_vdecl vars) ^ "\n" ^
  String.concat "(List.map string_of_fdecl funcs)"
SAST:

```plaintext
open Ast

type sast_expr =
  SAST_Literal of int
  | SAST_FloatLiteral of float
  | SAST_StrLiteral of string
  | SAST_BoolLit of bool
  | SAST_Id of string
  | (* HJ: ArrayIndexValue *)
  | SAST_ArrayIndexValue of string * sast_expr
  | SAST_Binop of sast_expr * op * sast_expr
  | SAST_Unop of uop * sast_expr
  | SAST_Assign of string * sast_expr
  | SAST_Call of string * sast_expr list
  | SAST_Noexpr

type sast_stmt =
  SAST_Block of sast_stmt list
  | SAST_Expr of sast_expr
  | SAST_Return of sast_expr
  | SAST_If of sast_expr * sast_stmt * sast_stmt
  | SAST_For of sast_expr * sast_expr * sast_expr * sast_stmt
  | SAST_While of sast_expr * sast_stmt

type func_decl = {
  sast_typ : the_types;
  sast_fname : string;
  sast_formals : var_formal list;
  sast_locals : var_local list;
  sast_body : sast_stmt list;
}

type sast_program = var_decl list * func_decl list
```

Semant:
(* Semantic checking for the TPL compiler *)
open Ast
open Sast

module StringMap = Map.Make(String)

(* Semantic checking of a program. Returns void if successful, 
throws an exception if something is wrong.

Check each global variable, then check each function *)

let check (globals, functions) =

(* Raise an exception if the given list has a duplicate *)
let report_duplicate exceptf list =
  let rec helper = function
    | n1 :: n2 :: _ when n1 = n2 -> raise (Failure (exceptf n1))
    | _ :: t -> helper t
    | [] -> ()
  in helper (List.sort compare list)

(* Raise an exception if a given binding is to a void type *)
let check_not_void exceptf = function
  | (Thetype(Void), n) -> raise (Failure (exceptf n))
  | _ -> ()

(* Raise an exception of 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 err

(** Checking Global Variables ***)
List.iter (check_not_void (fun n -> "illegal void global " ^ n)) globals;
report_duplicate (fun n -> "duplicate global " ^ n) (List.map snd globals);

(** Checking Functions ***)

let predefined_functions = [ "print" ; "printb" ; "printfloat" ] in

let check_is_pridifined x = if List.mem x (List.map (fun fd -> fd.fname) functions)
  then raise (Failure ("function " ^ x ^ " may not be defined")) else () in
List.iter check_is_pridifined predefined_functions;

report_duplicate (fun n -> "duplicate function " ^ n)
  (List.map (fun fd -> fd.fname) functions);

(* Function declaration for a named function *)
let built_in_decls = StringMap.add "print" { typ = Thetype(Void); fname = "print"; formals = [VFormal(Thetype(Int), "x")]; locals = []; body = [] } {StringMap.add "println" { typ = Thetype(Void); fname = "println"; formals = [VFormal(Thetype(Bool), "x")]; locals = []; body = [] } {StringMap.singleton "printfloat" { typ = Thetype(Void); fname = "printfloat"; formals = [VFormal(Thetype(Float), "x")]; locals = []; body = [] } })

let function_decls = List.fold_left (fun m fd -> StringMap.add fd.fname fd m) built_in_decls functions

let function_decl s = try StringMap.find s function_decls with Not_found -> raise (Failure ("unrecognized function " ^ s))

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

let check_function func =
  List.iter (check_not_void (fun n -> "illegal void formal " ^ n ^ 
  " in " ^ func.fname)) (List.map (function VFormal (ty, id) -> (ty, id)) func.formals);
  report_duplicate (fun n -> "duplicate formal " ^ n ^ 
  " in " ^ func.fname) (List.map (function VLocal (ty, id) -> (ty, id)) func.locals);
  List.iter (check_not_void (fun n -> "illegal void local " ^ n ^ 
  " in " ^ func.fname)) (List.map (function VLocal (ty, id) -> (ty, id)) func.locals);
  report_duplicate (fun n -> "duplicate local " ^ n ^ 
  " in " ^ func.fname) (List.map (function VLocal (ty, id) -> (ty, id)) func.locals);

(* Type of each variable (global, formal, or local *)
let symbols = List.fold_left (fun m (t, n) -> StringMap.add n t m) StringMap.empty (globals @ (List.map (function VFormal(ty, id) -> (ty, id)) func.formals ) @ (List.map (function VLocal(ty, id) -> (ty, id)) func.locals ))

let type_of_identifier s =
  try StringMap.find s symbols
  with Not_found -> raise (Failure ("undeclared identifier " ^ s))
let rec expr = function
| SAST_Literal _ -> Int
| SAST_FloatLiteral _ -> Float
| SAST_StrLiteral _ -> String
| SAST_BoolLit _ -> Bool
| SAST_Id s -> type_of_identifier s
| SAST_Binop(e1, op, e2) as e -> let t1 = expr e1 and t2 = expr e2 in
(match op with
| Add | Sub | Mult | Div when t1 = Int && t2 = Int -> Int
| Equal | Neq | Greater | Geq when t1 = Int && t2 = Int -> Bool
| And | Or when t1 = Bool && t2 = Bool -> Bool
| _ -> raise (Failure ("illegal binary operator " ^ string_of_typ t1 ^ " " ^ string_of_op ^ " " ^ string_of_typ t2 ^ " in " ^ string_of_expr e))
| SAST_Unop(op, e) as e -> let t = expr e in
(match op with
| Neg when t = Int -> Int
| _ -> raise (Failure ("illegal unary operator " ^ string_of_uop op ^ string_of_typ t ^ " in " ^ string_of_expr e)))
| SAST_Noexpr -> TheType(Void)
| SAST_Assign(var, e) as e -> let lt = type_of_identifier var
| and rt = expr e in
| check_assign lt rt (Failure ("illegal assignment " ^ string_of_typ lt ^ " = " ^ string_of_typ rt ^ " in " ^ string_of_expr ex))
| SAST_Call(fname, actuals) as call -> let fd = function_decl fname in
if List.length actuals = List.length fd.formals then
raise (Failure ("expecting " ^ string_of_int
| (List.length fd.formals) ^ " arguments in " ^ string_of_expr call))
else
| List.iter2 (fun (ft, _) e -> let et = expr e in
| ignore (check_assign ft et
| (Failure ("illegal actual argument found " ^ string_of_typ et ^ " expected " ^ string_of_typ ft ^ " in " ^ string_of_expr e)))
| fd.formals actuals;
| fd.typ
| in
| let check_bool_expr e = if expr e = Bool
| then raise (Failure ("expected Boolean expression in " ^ string_of_expr e))
| else () in
| in
/* Verify a statement or throw an exception */

let rec stmt = function
  | SAST_Return _ :: _ -> raise (Failure "nothing may follow a return")
  | SAST_Block s1 :: ss -> check_block (s1 @ ss)
  | s :: ss -> stmt s ; check_block ss
  | [] -> ()

in check_block s

| SAST_Expr e -> ignore (expr e)
| SAST_Return e -> let t = expr e in if t = func.typ then () else
  raise (Failure ("return gives " ^ string_of_typ t ^ " expected " ^
  string_of_typ func.typ ^ " in " ^ string_of_expr e))

| SAST_If(p, b1, b2) -> check_bool_expr p; stmt b1; stmt b2
| SAST_For(e1, e2, e3, st) -> ignore (expr e1); check_bool_expr e2;
  ignore (expr e3); stmt st
| SAST_While(p, s) -> check_bool_expr p; stmt s

in

stmt (Block func.body)

in

List.iter check_function functions
(* 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
module S = Sast

module StringMap = Map.Make(String)

let translate (globals, functions) =
  let context = L.global_context () in
  let the_module = L.create_module context "TPL"
  and i32_t = L.i32_type context
  and i16_t = L.i16_type context
  and i8_t = L.i8_type context
  and void_t = L.void_type context
  and pointer_t = L.pointer_type
  and float_t = L.double_type context in

  (* HJ: Not complete list *)
  let ltype_of_basic_types = function
    | A.Int -> i32_t
    | A.Bool -> i1_t
    | A.Void -> void_t
    | A.Float -> float_t
    | A.String -> pointer_t i8_t
  in

  let ltype_of_typ = function
    | A.T(the_type) -> ltype_of_basic_types the_type
  in

  (* Declare each global variable; remember its value in a map *)
  let global_vars =
    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

  (* Declare printf(), which the print built-in function will call *)
  let printf_t = L.var_arg_function_type i32_t [] L.pointer_type i8_t [] in
  let printf_func = L.declare_function "printf" printf_t the_module in
let function_decls = let function_decl m fdecl = let name = fdecl.s.sast_fname and formal_types = Array.of_list (list.map (function A.VFormal (t, _) -> ttype_of_typ t) fdecl.s.sast_formals) in let ftype = L.function_type (ttype_of_typ fdecl.s.sast_typ) formal_types in StringMap.add name (L.define_function_name name ttype the_module, fdecl) m in List.fold_left function_decl StringMap.empty functions in

let build_function_body fdecl = let (the_function, _) = StringMap.find fdecl.s.sast_fname function_decls in let builder = L.builder_at_end context (L.entry_block the_function) in

let fmt_format_str = L.build_global_stringptr "%d" "fmt_integer" builder in let float_format_str = L.build_global_stringptr "%f" "fmt_float" builder in let string_format_str = L.build_global_stringptr "%s" "fmt_string" builder 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.buildalloca (ttype_of_typ t) n builder in ignore (L.build_store p local builder); StringMap.add n local m in

let add_local m (t, n) = let local_var = L.buildalloca (ttype_of_typ t) n builder in StringMap.add n local_var m in

let formals = List.fold_left2 add_formal StringMap.empty (list.map (function A.VFormal(ty, id) -> (ty, id)) fdecl.s.sast_formals) (Array.to_list (L.params the_function)) in List.fold_left add_local_formals (list.map (function A.VLocal (ty, id) -> (ty, id)) fdecl.s.sast_locals) in

(* Return the value for a variable or formal argument *) let lookup n = try StringMap.find n local_vars with Not_found -> StringMap.find n global_vars in
(* Construct code for an expression; return its value *)
(* M3 Account for Sast*)

let rec expr builder =
  S.SAST_Literal i -> L.const_int i32_t i
  | S.SAST_FloatLiteral f -> L.const_float float_t f
  | S.SAST_BoolLit b -> L.const_int i1_t (if b then 1 else 0)
  | S.SAST_Noexpr -> L.const_int i32_t 0
  | S.SAST_Id s -> L.build_load (lookup s) s builder
  | S.SAST_Binop (e1, op, e2) ->
let e1' = expr builder e1
  and e2' = expr builder e2 in
(match op with
  A.Add -> if (true) then (L.build_add) else (L.build_fadd)
  | A.Sub -> if (true) then L.build_sub else L.build_fsub
  | A.Mult -> if (true) then L.build_mul else L.build_fmul
  | A.Div -> if (true) then L.build_sdiv else L.build_fdiv
  | 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
  | S.SAST_Unop (op, e) ->
let e' = expr builder e in
(match op with
  A.Neg -> L.build_neg
  | A.Not -> L.build_not)
  e' "tmp" builder
  | S.SAST_Assign (s, e) -> let e' = expr builder e in
ignore (L.build_store e' (lookup s) builder); e'
  | S.SAST_Call ("printf", [e]) | S.SAST_Call ("printf", [e]) ->
L.build_call printf_func [] (int_format_str ; (expr builder e))
"printf" builder
  | S.SAST_Call ("printf", [e]) ->
L.build_call printf_func [] (float_format_str ; (expr builder e))
"printf" builder
  | S.SAST_Call (f, act) ->
let (fdef, fdecl) = StringMap.find f function_decls in
let actuals = List.map (expr builder) (List.rev act) in
let result = (match fdecl.S.sast_typ with A.TheType(A.Void) -> ""
  | _ -> f ^ ".result") in
L.build_call fdef (Array.of_list actuals) result builder

(* Invoke "f builder" if the current block doesn't already
have a terminal (e.g., a branch). *)
let add_terminal builder f =
match L.block_terminator (L.insertion_block builder) with
  Some _ -> ()
and None -> ignore (f builder) in
let rec stmt builder = function
  | S.SAST_Block sl -> List.fold_left stmt builder sl
  | S.SAST_Expr e -> ignore (expr builder e); builder
  | S.SAST_Return e -> ignore (match fdecl.S.sast_typ with
    A.Thetype (A.Void) -> L.build_ret_void builder
    _ -> L.build_ret (expr builder e) builder)
    | S.SAST_If (predicate, then_stmt, else_stmt) ->
      let bool_val = expr builder predicate in
      let merge_bb = L.append_block context "merge" the_function in
      let then_bb = L.append_block context "then" the_function in
      let add_terminal (stmt (L.builder_at_end context then_bb) then_stmt) (L.builder_br merge_bb)
      let else_bb = L.append_block context "else" the_function in
      let add_terminal (stmt (L.builder_at_end context else_bb) else_stmt) (L.builder_br merge_bb)
      ignore (L.build_cond_br bool_val then_bb else_bb builder) in
      L.builder_at_end context merge_bb
  | S.SAST_While (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
    let add_terminal (stmt (L.builder_at_end context body_bb) body) (L.builder_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) in
    L.builder_at_end context merge_bb
  | S.SAST_For (e1, e2, e3, body) -> stmt builder
    (S.SAST_Block [S.SAST_Expr e1; S.SAST_While (e2, S.SAST_Block [body ; S.SAST_Expr e3]) ])
  in

let builder = stmt builder (S.SAST_Block fdecl.S.sast_body) in

let build_function_body_functions = List.iter build_function_body_functions;

the_module