A Whitepaper and Tutorial for the La Mesa Language

La Mesa is a language that deals extensively with tables. It is designed for people who have programming experience in an object oriented language, but find themselves having to interact a lot with tabular data. The two primary avenues that computers have for dealing with table-oriented data are relational databases and spreadsheets. Both have well known strengths and drawbacks; SQL and the relational model in particular are often used as a catch-all for many different types of solutions that wouldn't otherwise fit the relational concept. The world of databases and schemas is sometimes overkill for someone who needs a light, flexible, storage mechanism for a small problem; as the problem grows, however, one often needs to transition to a more stable solution than a spreadsheet, while still maintaining the flexibility of being able to shift the data structure as the problem requirements change, and offering the programmer a visual way to examine the problem.

La Mesa is designed to bridge these two worlds, combining the flexibility and ease of use of a spreadsheet with the programming-centric nature of the relational database, without falling victim to all the associated heft of a relational database. La Mesa is also an experiment to look at the world entirely from a tabular perspective, and see what would happen. In other words, if the primary data construct available to tackle a problem was a table, how would a language designed around this construct more readily accommodate a programmer? This language is targeted around the same family of concepts that the "NoSQL" movement seems targeted to, but as the author is not very familiar with any particular flavor of a NoSQL architecture, this will be an independent implementation.

TUTORIAL

To demonstrate the features of the language, the reader is introduced to two example tables, shown below. The tables are simple; they are the type of thing that one would create in a spreadsheet. Nevertheless, we would like to be able to manipulate the tables in a meaningful way before we decide whether a more formal relational model is appropriate. The first table describes data for standardized test questions, such as the type of question, the number of the question, the page from the book whence the question came, and the difficulty of the question. The standardized test in this case was the SAT; a larger version of this table was obtained off the web.

prb_type	prb_num	pg_num	Diff
“AbsValue”	**15**	**418**	**“Medium”**
“AbsValue”	**17**	**456**	**“Medium”**
“Average”	**18**	**835**	**“Hard”**
“Equations”	**6**	**454**	**“Easy”**

There are 10 tests in the book, and it would be nice to know from which of the 10 tests each question came from. Rather than have to go through and look up that information for each question, we've compiled the following table:

Test_number	Test_begin_page	Test_end_page
1	**389**	**431**
2	**451**	**493**
8	**823**	**865**

From this data, we'd like to construct a very simple customized test for a student, consisting of an easy, a medium, and a hard question, each of different problem types. As an additional constraint, we'd like to make sure that each question was pulled from a different test, to make sure that there is maximum variation in our test construction. In a more formal environment, we’d either have to throw this into a database or learn the VB Spreadsheet Model to lookup this data, but here we can, without too much overhead, express this just as a table. Eventually, the La Mesa standard library will come to encompass comma-separated-value files and other ways of handling tables in files, but for now we just describe the table in native language construct:

```
table question_info =
#{ string prb_type    ,int prb_num    ,int pg_num     ,string diff
#  “absvalue”         ,15              ,418            ,”medium”
#  “absvalue”         ,17              ,456            ,”medium”
#  “average”          ,18              ,835            ,”hard”
#  “equations”        ,6               ,454            ,”easy”
}#;
```

As we can see, we had to specify the signature, or sequence of data types contained in each table, in the first row. Future versions of La Mesa might take multiple passes through the data to infer the data type of each column, but this is a good way to circumvent unnecessary complexity. We declare our next table:

```
table lookup_table =
#{ int test_number, int test_begin_page,int test_end_page
#  1  ,389  ,431
#  2  ,451  ,493
#  8  ,823  ,865
}#;
```

La Mesa currently supports three different types of data: int, string, and float. While the language it generates, Javascript (and Javascript DOM), doesn't have strongly typed data, La Mesa preserves some type strength, again as a bridging mechanism between the “anything goes” world of spreadsheets and the formally specified world of relational databases. Some type strength would also help with future formatting of the data.

Now that we have our test data, to produce our customized test it would be a good idea to create a table where the number of the test is simply appended to the first table that we created. In a relational world, we’d have to execute a join and then project the columns we wanted. Since the target La Mesa audience doesn’t understand relational databases, but also doesn’t know how to write spreadsheet scripting code, we’d like to express this idea in a more standard programming language way. Before we get too far into the process, and because La Mesa is compiled, and not interpreted, we need to encapsulate the code in a function. Let's use 'main,' which is the entry point for the La Mesa compiler into every program, and hence, should be avoided for use as a different identifier. If you don't define a main function, the compiler will throw an exception that one was not found.

```
function int main() {
  table xtd_question_info =
  #{ int prb_num ,int pg_num, int test_num
  }#;
  int test_num;
  table xtd_question_row = #{ int prb_num   ,int pg_num
  #   0,         , 0
  }#;
  xtd_question_row :: test_num;
  question_info | question_row {
    test_lookup | test_lookup_row {
      if(question_row.pg_num >= test_lookup_row.test_begin_page) {
        if (question_row.pg_num <= test_lookup_row.test_end_page) {
          xtd_question_row.test_num = test_lookup_row.test_number;
          xtd_question_row.prb_num = question_info.prb_num;
          xtd_question_row.pg_num = question_info.pg_num;
          xtd_question_info << xtd_question_row;
        }
      }
    }
  }
}
```

Now, there is a lot going on here, so we'll step through each line. The first line,

```
table xtd_question_info =
  #{ int prb_num ,int pg_num, int test_num
}#;
```

is a declaration of an empty table that's going to hold our result. The next line,

```
int test_num;
```

is a declaration of a local variable that's going to be used later. Primitive types of int, float, and string are the foundations for constructing our table, but, like java, a primitive variable is a separate construct from a full-on table.   There are a couple of language features that bridge these worlds, but for now it is best to think of them as separate entities. The next line,

```
table xtd_question_row = #{ int prb_num     ,int pg_num
  #   0,         , 0
  }#;
```

declares a single table row that's going to be used to build our final table. Upon inspection, it appears we forgot to add on the final "test_num" column. There is a feature that allows us to do this, which is the Column Append operator:

```
xtd_question_row :: test_num;
```

is a declaration of a variable of type int.

```
question_info :: qi_test_num;
```

which appends the variable we created to the table that contains our data. The double colon operator takes either a primitive or a table variable, appends it to a table, and produces a table of the new type. The variable on the left side of the double colon operator must be a table, it cannot be a primitive. Also, both sides of the double colon operator must be previously declared variables; table literals cannot be used outside of their declaration. With this operator, initial tables can be expanded column by column, if necessary, allowing the table to grow as user needs dictate. This serves as a counterpoint to the relational method of having to predefine tables. Due to issues with scope and functions, however, the programmer is strongly cautioned not to use the column operator too liberally, and to be aware of the dangers of modifying the signature of a table, where the original might be expected by a function at some point in the future. Good programming style suggests not using tables that might be returned by functions with the double colon operator; keep returned tables separate from the ones being constructed, and build them up row by row. A future release of La Mesa would make explicit this notion of unexpandable global tables (which would be required as a function return value) versus expandable local tables. Future versions of La Mesa might support more operations akin to joins, but it is not currently under consideration.

The next two lines,

```
question_info | question_row {

      lookup_table | lookup_row {
```

showcase the use of the pipe operator. The pipe operator connects an existing table and a newly defined identifier that will execute the code in the curly braces once for each row in the table. The newly defined identifier can be used in the curly braces to access columns in each call to the row. Indeed, in our if statement in the next line,

```
if(question_row.pg_num >= test_lookup.test_begin_page) {

if(question_row.pg_num <= test_lookup.test_end_page) {

      question_row.qi_test_num = lookup_row.test_number;

}
}
```

we access the value in the “pg_num” column of the “question_row”, and make a boolean comparison to the “test_begin_page” of the “test_lookup” row. If the “pg_num” value satisfies both comparison, we populate the “qi_test_num” column with the value from the corresponding “test_number” column. After running this code, we now have a table that looks like:

Prb_num	**Pg_num**	**Test_num**
15	**418**	**1**
17	**456**	**2**
18	**835**	**8**
6	**454**	**2**

Chapter 3: Reference Manual

3.1 Grammar Notation

In order to facilitate understanding, grammar notation is defined as it is introduced. In maintaining a common style with other reference manuals, nonterminals are shown in italics, while terminals are shown in quoted symbols. A ‘|’ defines that one of several options may be selected.

3.2 Lexical Conventions

A program consists of one or more global variable declarations, followed by one or more function declarations. Global variable declarations will be defined in section 3.3, and functions is section 3.4

3.2.1 Whitespace

Whitespace consists of all standard whitespace, such as tabs, new lines, formfeeds, and spaces. Whitespace does not affect the compilation of La Mesa.

3.2.2 Comments

Comments are the standard C and C++ multiline comments, starting with ‘/*’ and ending with ‘*/’. Comments are not nested. Any symbols in between the start and end of a comment will be ignored by the compiler.

3.2.4 Tokens

There are four classes of tokens in La Mesa: identifiers, keywords, literals, and operators.

3.2.5 Identifiers

An identifier conforms with many programming language notions of an identifier: it must start with a letter, after which it many contain any combination of letters, digits, or underscores. No special characters are allowed in identifiers.

*Identifier ▭ letter (letter | digit | ‘_’) **

3.2.5 Keywords

The following identifiers are used as keywords, and may not be used otherwise:

`int`	`float`	`string`	`table`	`if`
`Else`	`For`	`while`	`return`	`function`

3.2.5 Literals

A literal conforms with the standard notion of a java/C++ literal, albeit somewhat simplified. A string must be quoted, and it may only contain the same characters as an identifier within the string, with the exception of also allowing spaces. An integer consists of a sequence of digits, and a floating point number consists of two sequences of digits, separated by a decimal point.

Float_Literal -> digit+'.'digit+

String_Literal -> '"'letter (letter | digit | '_' | ' ')'"'*

Literal -> digit+

3.2.6 Operators

An operator is a token that either acts as a separator to distinguish concepts in La Mesa, or as a construct that manipulates primitive values. Primitive values consist either of literals of 'string', 'int', and 'float', and identifiers referring to such literals, or tables, which are collections of those primitive variables organized into rows and columns, which every column having a name and type, and every row containing data that corresponds to that type. Operators will be discussed in terms of the grammar of global table declarations and statements.

3.3 Global Table Declarations

A global table declaration contains one of two possible declarations: a declaration of a primitive datum identifier of type 'int','float','string', followed by an optional assignment of a literal datum to that identifier (unassigned primitives are given default values), or a declaration of a table construct, containing the naming and sequence of primitive data types that compose its signature, followed by an optional assignment of literal values arranged in a table that correspond to the data types listed in that signature.

table_declaration:-
'int' Identifier ';'
| 'float' identifier ';'
| 'string' identifier ';'
| 'int' Identifier '=' Literal ';'
| 'string' Identifier '=' String_Literal ';'
| 'float' Identifier '=' Float_Literal ';'
| 'table' Identifier '=' '#{'((('Int' | 'float' | 'string') Identifier),) (('Int' | 'float' | 'string') Identifier) '}#;'*

| ‘table’ Identifier ‘=’ ‘#{‘ (((‘Int’ | ‘float’ | ‘string’) Identifier),) ((‘Int’ | ‘float’ | ‘string’) Identifier) ‘#’ ((‘Literal’ | ‘Float_Literal’ | ‘String_Literal’) ‘,’)* (‘Literal’ | ‘Float_Literal’ | ‘String_Literal’) ‘}#;’*

3.4 Function Declarations

A function declaration consists of the word ‘function’, followed by an assignmentless global table declaration (in this instance, a primitive declaration does not require an identifier), followed by an identifier for the function’s name, followed by a ‘{‘, followed by a list of primitive formally declared parameters that are passed in to the function, followed by a ‘}’, followed by a list of La Mesa statements. To emphasize, formally declared parameters are limited to primitive declared values. Passing around tables in functions would lead to great confusion as to the state of the table at any one particular time. As La Mesa is intended to aid and not confuse programmers, advanced table manipulation should be confined to globally known tables.

Function declaration:-
‘function’ returntype Indentifier ‘(‘
(((‘Int’ | ‘float’ | ‘string’) Identifier),) ((‘Int’ | ‘float’ | ‘string’) Identifier)*
‘) {‘
stmt-list
‘}’

returntype:-
(‘int’ | ‘float’ | ‘string’) |
‘table #{‘(((‘Int’ | ‘float’ | ‘string’) Identifier),) ((‘Int’ | ‘float’ | ‘string’) Identifier)*
‘}#’

3.4.1 Statements

A statement list in a function is simply a sequence of La Mesa statements. A La Mesa statement consists of the familiar constructs to a C/C++/Java Programmer, with some additional support for manipulating tables. The for, while, and if constructs are well known to a programmer; La Mesa adds support for local variable declarations, a pipe operator that loops through a table, assigning each row to a local variable, and the ability to append columns to a table, either from another table or a primitive, as well as the ability to append a row to a table of the same signature of primitive types. The best way to conceptualize statements in La Mesa is that they are concerned with tables, while expressions, listed below, are concerned with primitive types.

statement:-
expression ‘;’
| ‘return’ expression ‘;’
| ‘{‘ expression ‘}’;
| ‘if (‘ expression ‘)’ statement
| ‘if (‘ expression ‘)’ statement ‘else’ statement
| ‘for (‘ expression ‘;’ expression ‘;’ expression ‘)’ statement
| ‘while (‘ expression ‘)’ statement

| *Identifier '|' Identifier '{' statement-list '}'*
| *table_declaration*
|*Identifier '<<' Identifier*
| *Identifier '::' Identifier*

3.4.1.1 The Pipe Operator

The pipe operator is a way to loop through a table. For each row in the table, a previously undeclared variable following the pipe is assigned to each row. The temporary variable has the same signature as the table from which it gets its row.
| *Identifier '|' Identifier '{' statement-list '}'*

3.4.1.2 The RowAppend Operator

The rowappend operator is a way to build up a table, row by row. Good programming style suggests that the first identifier in the sequence is a table to be built or appended to, while the second identifier is a row variable obtained from the Pipe construct above. After the append, the first identifier refers to a table with an additional row, while the second is unchanged and may be reused without modification.

|*Identifier '<<' Identifier*

3.4.1.2 The ColAppend Operator

The colappend operator is a way to add to a table, either column by column or table by table. Good programming style suggests that the first identifier in the sequence is a table to be built or broadened, while the second identifier may be an identifier of any construct. After the append, the first identifier refers to a table with an additional column, while the second is unchanged and may be reused without modification. Adding columns to tables invites all sorts of potential problems with verifying the state and signature of a table at any particular time, the programmer is urged to use it cautiously.

|*Identifier '::' Identifier*

3.4.2 Expressions

An expression is a language construct that deals with primitive values in La Mesa. La Mesa provides many of the constructs available to a C/C++/Java programmer, such as comparative statements, simple arithmetic on ints and floats, as well as reassignment of identifiers. In addition, La Mesa provides an operator to access the value of a particular row a table.

expression:-
literal
| *Float_literal*
| *String_literal*
| *Identifier*

| *expression (‘+’ | ‘-‘ | ‘*’ | ‘/’ | ‘==’ | ‘!=’ | ‘<’ | ‘>’ | ‘>=’ | ‘<=’) expression*
| *identifier ‘=’ expression*
| *identifier ‘.’ identifier ‘=’ expression*
| *identifier ‘(‘ expression-list ‘)’*
| *‘(‘ expression ‘)’*
| *identifier ‘.’ identifier*

3.4.2.1 The Dot Operator

The dot operator is a way to access the element of a table. The identifier to the left of the dot must refer to a table that only contains one row of data. Good programming style suggests that this will most often be an identifier associated with the pipe operator. The identifier to the right must be a valid named column of the first identifier. The dot operator may also be used in assignment.
| *identifier ‘.’ identifier ‘=’ expression*
| *identifier ‘.’ identifier*

Chapter 4: Project Plan

Unfortunately, the process used for planning and specification, development, and testing was ad-hoc. This was the greatest failing of the project implementer, by far. Implementer had some experience with working with excel and some relational database experience, but little in either the language of the compiler or the target language environment. As project fell behind timeline, implementer made disastrous decision to expand feature capability to compensate, without a corresponding understanding of how time-sensitive each component was. When implementer had to reset the project, the necessity of time forced a plan on the project. If the implementer was allowed to go back in time, this is what a project plan should have looked like:

Week 1:
Specification of grammar, implementation of scanner, parser, abstract syntax tree

Week2:
Implementation of type checking system, ensuring that type safety is maintained throughout program.

Week 3:
Implementation of code generation and standard library.

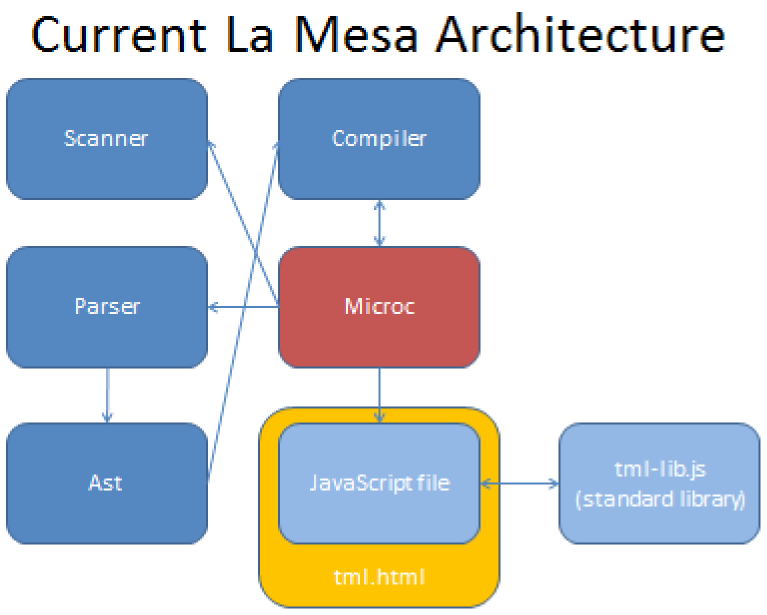
Software Development

Software was implemented using OCaml and make on a linux laptop running Ubuntu. Gedit was used as the text editor. JavaScript was debugged using Firebug plugin for Firefox.

Project Log

See hard copy.

Chapter 5: Architecture Design



Interfaces are similar to the microc project. Scanner generated a program that was fed into the parser, which generated an abstract syntax tree that was validated by the compiler. Compiler generated a JavaScript program, which was viewed through a web browser and debugged in the browser.

Chapter 6: Test Plan

For sample code of each test suite, see attached code. Sample code was chosen so as to fully stress-test each feature added to the compiler. For each operator or feature added, a new test program was created. Once the initial program compiled successfully, temporary modifications were made to the program so that it would fail in the manner intended (i.e. by throwing the correct exception). No automation was used in testing, unfortunately. The full printout of example tests is in the appendix.

Test Case 4 (representative example)

```
/* This is a basic test of function usage. */
float e = 2.71;
float pi = 3.14159;
float pi2;


function string main() {
      pi2 = 22.0 / 7.0;
}
```

Test Case 16 (representative example)

```
/* This function tests the toDOM function in standard library. */

table test_lookup = #{ int test_number, int test_begin_page, int
test_end_page
# 1   , 389 , 431
# 2   , 451, 493
# 8 , 823, 865 }#;

table test_blarg = #{ int test_number, int test_begin_page, int
test_end_page
# 7, 761, 803}#;

table test_lookup2 = #{ int test_number, int test_begin_page, int
test_end_page
      # 3 , 513, 555
      # 4 , 575, 617
      # 5 , 637, 679
      # 6 , 699, 741
      # 7 , 761, 803 }#;

int test4 = 4;
int test6 = 6;

function int main() {
      toDOM(test_lookup2);
      toDOM(test_blarg);
      print(test6);
      toDOM(test_lookup);
      return get_test(test4, test6);
}

function int get_test(int test_request, int test_request2) {
      if(test_request + 2 == test_request2) {
            return test_request;
      } else {
            return 0;
      }
}
```

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq (*| And_ | Or_*)

type prim = Int | Float | String

type signature =
        Signature of (string * prim) array

type mandatorylit =
         LitInt of int
        | LitFloat of float
        | LitString of string

type tablerow =
        LiteralRow of mandatorylit array

type tabledecl =
                FullTableLiteral of string * signature * tablerow array
  | SingleVarLiteral of (string * prim) * mandatorylit

type expr =
    Literal of mandatorylit
  | Id of string
  | Binop of expr * op * expr
  | Assign of string * expr
  | TAssign of string * string * expr
  | Call of string * expr list
  | GetCol of string * string
(*  | Filter of string * expr *)
  | 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
  | Pipe of string * string * stmt list
  | LocalDecl of tabledecl
  | RowAppend of string * string
  | ColAppend of string * string


type func_decl = {
    fname : string;
    returnsig : tabledecl;
    formals : (string * prim) list;
    body : stmt list;
  }

type program = tabledecl list * func_decl list

let string_of_vdecl id = "int " ^ id ^ ";\n"

let string_of_mandatory_lit = function
         LitInt(i) -> string_of_int(i)
        | LitFloat(f) -> string_of_float(f)
        | LitString(s) -> if s = "" then "\"\"" else s
```

```
let string_of_tablerow = function
                        LiteralRow(lits) -> String.concat ", " (Array.to_list (Array.map string_of_mandatory_lit lits))

let string_of_tabletype = function
            (s, p) -> ((match p with Int -> "int" | Float -> "float" | String -> "string") ^ " " ^ s)

let string_of_signature = function
                        Signature(sigs) -> String.concat ", " (Array.to_list (Array.map string_of_tabletype sigs))

let string_of_tdecl = function
            FullTableLiteral(s1, sig_n, tablerows) -> "table " ^ s1 ^ " = #{\n" ^ (string_of_signature sig_n) ^ "\n#" ^
(String.concat "\n#" (Array.to_list (Array.map string_of_tablerow tablerows))) ^ "}#\n"
  | SingleVarLiteral((sname, sprim), lv) -> (match sprim with Int -> "int" | Float -> "float"
                        | String -> "string") ^ " " ^ sname ^ " = " ^ string_of_mandatory_lit(lv) ^ ";\n"

let rec string_of_expr = function
    Literal(l) -> string_of_mandatory_lit l
  | Id(s) -> s
  | Binop(e1, o, e2) ->
      string_of_expr e1 ^ " " ^
      (match o with
            Add -> "+" | Sub -> "-" | Mult -> "*" | Div -> "/"
      | Equal -> "==" | Neq -> "!="
      | Less -> "<" | Leq -> "<=" | Greater -> ">" | Geq -> ">="
      (*| And_ -> "&&" | Or_ -> "||"*)) ^ " " ^
      string_of_expr e2
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | TAssign(v, v', e) -> v ^ "." ^ v' ^ " = " ^ string_of_expr e
  | Call(f, el) ->
      f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Noexpr -> ""
  | GetCol(s1, s2) -> s1 ^ "." ^ s2
(*  | Filter(s1, ex2) -> s1 ^ "[" ^ string_of_expr ex2 ^ "]" *)

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
  | Pipe(s1, s2, stmts) -> s1 ^ "|" ^ s2 ^ "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | LocalDecl(td) -> string_of_tdecl td
            | RowAppend(s1,s2) -> s1 ^ "<<" ^ s2
  | ColAppend(s1, s2) -> s1 ^ "::" ^ s2


let string_of_fdecl fdecl =
  "function " ^ (string_of_tdecl fdecl.returnsig) ^ " " ^ fdecl.fname ^ " (" ^ (String.concat ", " (List.map (fun (s, p) -> (match
p with Int -> "int" | Float -> "float" | String -> "string") ^ " " ^ s ^ " ") fdecl.formals)) ^ ")\n{\n" ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"
```

```
let string_of_program (vars, funcs) =
              let srav = List.rev vars in
  String.concat "" (List.map string_of_tdecl srav) ^ "\n" ^
  String.concat "\n" (List.map string_of_fdecl funcs)
```

```
open Ast

module StringMap = Map.Make(String)

type env = {
   function_index : Ast.func_decl StringMap.t; (* created once; mapping of function name to function AST; prob could be a
list*)
   mutable global_index   : Ast.tabledecl StringMap.t; (* created once but changed; mapping of table name to table AST;
necessary *)
   mutable local_index    : (Ast.tabledecl * Ast.func_decl) StringMap.t; (* dynamically updated; mapping of local table
name to AST; necessary *)
   mutable current_enclosing_function : string;
  }



exception Illegal_table_param of Ast.tabledecl
exception Function_not_found of string

let prim_check_of_literal mandatoryval = (match mandatoryval with LitInt(_) -> Int | LitString(_) -> String | LitFloat(_) ->
Float)

let prim_check_of_table tbsignature littable =
	let signa =
		(match tbsignature with Signature(sigNa) -> sigNa)
		in
		for j = 0 to ((Array.length littable) - 1) do (
			let litrow =
				(match littable.(j) with LiteralRow(litr) -> litr)
			in
			if (Array.length signa <> Array.length litrow ) then
				raise(Failure("Dimension mismatch: " ^ string_of_int j))
			else (for i = 0 to ((Array.length signa) - 1) do (
				if ((match signa.(i) with
						(_, Int) -> Int
					| (_, Float) -> Float
					| (_, String) -> String ) <>
					(prim_check_of_literal litrow.(i) ) )
					then (raise(Failure("Invalid literal in column " ^ string_of_int i ^ "; row " ^
string_of_int j)))
					else ()
				) done
			)
		) done

let rec tdecl_list_build map = function
	|		[] -> map
	| SingleVarLiteral((sname, sprim), oplit) as hd :: tl -> let newmap = StringMap.add sname hd map in
tdecl_list_build newmap tl
	| FullTableLiteral(tbname, tbsig, tbllit) as hd :: tl -> let newmap = (*Array.fold_left (fun m (s, p) -> StringMap.add
s (match p with String_ -> SignatureString(tbname) | Int_ -> SignatureInt(tbname) | Float_ -> SignatureFloat(tbname)) m)
map (match tbsig with Signature(sigN) -> sigN)*) StringMap.add tbname hd map in tdecl_list_build newmap
tl(*(StringMap.add tbname (TableVar(tbsig)) newmap)*)


let rec fdecl_list_build map = function
	|		[] -> map
	| hd :: tl -> let newmap = StringMap.add hd.fname hd map in fdecl_list_build newmap tl
```

```
(** Translate a program in AST form into a JavaScript program.  Throw an
   exception if something is wrong, e.g., a reference to an unknown
   variable or function *)
let translate (globals, functions) =
        (* We need to do two things here.  We need to store the AST tabledecls in a
        data structure so that we can reference it later.  We also need to output of
        the declarations as part of a java class.*)

        let global_table_decls = tdecl_list_build StringMap.empty (List.rev globals) in
        let global_func_decls = fdecl_list_build StringMap.empty (List.rev functions) in

        let rec translate_globals = function
        | [] -> []
        | SingleVarLiteral((nm, prm), lit) :: tl ->
                if((prim_check_of_literal lit) <> prm) then
                        raise(Failure("global: " ^ nm ^ " is assigned a literal that does not match it's declared type"))
                else
                        let javaout = "var " ^ nm ^ " = " ^ Ast.string_of_mandatory_lit(lit) ^ ";\n"
                        in javaout :: translate_globals tl
        | FullTableLiteral(nm, tb_sig, tablerows) :: tl ->
                let () =
                        (try (prim_check_of_table tb_sig tablerows ) with Failure(msg) ->
                                raise(Failure("Error in: " ^ nm ^ ":" ^ msg)))
                and javaout = "var " ^ nm ^ " = {};\n" ^
                        nm ^ ".nameString = \"" ^ nm ^ "\";\n" ^
                        nm ^ ".rows = new Array();\n" ^
                        nm ^ ".headnames = new Array();\n" ^
                        nm ^ ".headtypes = new Array();\n" ^
                        (* Generate the signature string *)
                        nm ^ ".signature = \"" ^ (String.concat "::" (Array.to_list (Array.map (fun (id', prm') -> id' ^ ":" ^

                                                                                                                        (match prm' with
Int -> "int" | Float -> "float" | String -> "string"))

                                                                                                        (match tb_sig with
Signature(sigsig) -> sigsig)
                        ))) ^ "\";\n" ^
                        (* Generate the code to populate the .headnames and .headtypes array *)
                        (String.concat "" (Array.to_list (Array.map (fun (id', prm') -> nm ^ (*".headnames[" ^ nm ^
".headnames.length] = \"" ^*)

                                                                                                                        ".headnames." ^
id' ^ " = " ^ nm ^

                                                                                                                        ".headtypes.lengt
h;\n" ^ nm ^ ".headtypes[" ^ nm ^ ".headtypes.length] = \"" ^

                                                                                                                        (match prm' with
Int -> "int" | Float -> "float" | String -> "string") ^

                                                                                                                        "\";\n")

                                                                                                        (match tb_sig with
Signature(sigsig) -> sigsig)
                        )))
                        ^ nm ^ ".rows[" ^ nm ^ ".rows.length] = new Array();\n" ^
                        (String.concat (nm ^ ".rows[" ^ nm ^ ".rows.length] = new Array();\n")
                                (Array.to_list (Array.map
                                        (fun mlitrow -> let mlitarray = match mlitrow with LiteralRow(lits) -> lits in
                                                String.concat "" (Array.to_list (Array.map

                (fun mlit -> nm ^ ".rows[" ^ nm ^ ".rows.length - 1]" ^

                "[" ^ nm ^ ".rows[" ^ nm ^ ".rows.length - 1].length] = " ^

                (match mlit with LitInt(i) -> string_of_int i | LitFloat(f) -> string_of_float f | LitString(s) -> s)

        ^ ";\n")

                mlitarray

        ))
                                        )
                                        tablerows
                        )))
                in javaout :: translate_globals tl

        in


        let translate thisenv fundecl =
                        (* Now we have our typecheck function *)
                let () = thisenv.current_enclosing_function <- fundecl.fname
                        (*; print_string ("current_enclosing_function set: " ^ fundecl.fname)*) in
                let rec typecheck_of_expr = function
        Literal(l) -> (match l with LitInt(i) -> SingleVarLiteral(("",Int),LitInt(0))
                                                                                                                        |
LitFloat(f) -> SingleVarLiteral(("",Float),LitFloat(0.0))
                                                                                                                        |
LitString(s) -> SingleVarLiteral(("",String),LitString("")))
                | Id(s) -> (try (match (StringMap.find s thisenv.global_index) with
                                                                                                        SingleVarLiteral((_,Int),_)
-> SingleVarLiteral(("",Int),LitInt(0))
                                                                                        | SingleVarLiteral((_,Float),_) ->
SingleVarLiteral(("",Float), LitFloat(0.0))
                                                                                        | SingleVarLiteral((_,String),_) ->
SingleVarLiteral(("",String), LitString(""))
                                                                                        | FullTableLiteral(_, sig_n, _) ->
FullTableLiteral("", sig_n, [||])
                                                                                        )

                                                                        with Not_found -> (try (match (StringMap.find s
thisenv.local_index) with

        (SingleVarLiteral((_,Int),_),_) -> SingleVarLiteral(("",Int),LitInt(0))
                                                                                        | (SingleVarLiteral((_,Float),_),_)
-> SingleVarLiteral(("",Float), LitFloat(0.0))
                                                                                        | (SingleVarLiteral((_,String),_),_)
-> SingleVarLiteral(("",String), LitString(""))
                                                                                        | (FullTableLiteral(_, sig_n, _),_) ->
FullTableLiteral("", sig_n, [||])
                                                                                        )

                                                with Not_found -> raise(Failure("Unknown id: " ^ s))
                                        )
                                                                                )
                    | Binop(e1, o, e2) -> let p1 = (match (typecheck_of_expr e1) with SingleVarLiteral((_,stprm),_) ->
Signature([|("$sv",stprm)|])

                                                    |FullTableLiteral(_,sig_1,_) -> sig_1

                                        )
                                                                                                              and p2 =
(match (typecheck_of_expr e2) with SingleVarLiteral((_,stprm2),_) -> Signature([|("$sv",stprm2)|])

                                                    |FullTableLiteral(_,sig_2,_) -> sig_2

                                        )
                                                                                                              in if(p1
<> p2) then (
                              raise(Failure("\nType mismatch of expression: \n" ^ (Ast.string_of_expr e1) ^ "\n***\n" ^
(Ast.string_of_expr e2)))) else (
                    typecheck_of_expr e2
                                        )
                    | Assign(v, e) -> let p1 = (try (match (StringMap.find v thisenv.global_index) with

                                                                     SingleVarLiteral((_,stprm),_) -> Signature([|
("sv",stprm)|])

                                                                     | FullTableLiteral(_,sig_1,_) -> sig_1

                                        ) with Not_found -> (try (match(StringMap.find v thisenv.local_index) with

                                                                     (SingleVarLiteral((_,stprm),_),_) -> Signature([|
("sv",stprm)|])

                                                                     | (FullTableLiteral(_,sig_1,_),_) -> sig_1

                                                                     )

                                        with Not_found -> raise(Failure("Unknown id: " ^ v))))
                                                                                                       and p2 = (match
(typecheck_of_expr e) with SingleVarLiteral((_,stprm2),_) -> Signature([|("sv",stprm2)|])

                                                    |FullTableLiteral(_,sig_2,_) -> sig_2

                              )
                                                                                                       in if(p1 <> p2)
then (
                              raise(Failure("\nType mismatch in assignment: \n" ^ v ^ "\n***\n" ^ (Ast.string_of_expr e))))
else (
                    typecheck_of_expr e
                                        )
                              | TAssign(v, v', e) -> let p1 = typecheck_of_expr (GetCol(v, v'))

and p2 = typecheck_of_expr e

in if p1 <> p2 then raise(Failure("types do not match in TAssign")) else typecheck_of_expr e
```

```
                    | Call(f, el) -> (try(if(List.for_all2 (fun (_,p) (_,p') -> p = p')
                                                    (let f_formals = StringMap.find f thisenv.function_index in
f_formals.formals)
                                                    (List.map (fun tabledec -> match tabledec with
SingleVarLiteral(sp,_) -> sp
                                                         | s -> raise(Illegal_table_param(s)))
                                                         (List.map typecheck_of_expr el)
                                                    )
                                                                                )then(
      let func = (StringMap.find f thisenv.function_index) in func.returnsig
                                                                                )
else(raise(Failure("sequence of types supplied to " ^ f ^ " does not equal expected types"))
                                                                          )) with Not_found
-> raise(Function_not_found(f))
                                                                                |
Illegal_table_param(s) -> (if(StringMap.mem f thisenv.function_index)
      then(raise(Failure("\n Can't pass a table as parameter: " ^ Ast.string_of_tdecl s ^ "\n")))
      else(raise(Function_not_found(f)))
                                                                                )
                                                                      )
                    | Noexpr -> raise(Failure("A NoExpr has no type; it cannot be type checked."))
                         | GetCol(s1, s2) -> let p1 = (try (match (StringMap.find s1 thisenv.global_index) with
      SingleVarLiteral((_,stprm),_) -> raise(Failure("Can't get a column of a global primitive.\n"))
                                                                      |
FullTableLiteral(_,Signature(sig_1),sigarry) -> let () = if(Array.length sigarry <> 1) then(
      raise(Failure(s1 ^ ", a global, has " ^ string_of_int (Array.length sigarry) ^ " rows, can't refer to a single element")))
else (
      print_string "WARNING: trying to access a column of a global table\n")
                                                                          in sig_1
                         ) with Not_found -> (try (match(StringMap.find s1 thisenv.local_index) with
      (SingleVarLiteral((_,stprm),_),_) -> raise(Failure("Can't get a column of a local primitive.\n"))
                                                                      |
(FullTableLiteral(_,Signature(sig_1),sigarry),_) -> let () = if(Array.length sigarry <> 1)then(
      raise(Failure(s1 ^ ", a local, has " ^ string_of_int (Array.length sigarry) ^ " rows, can't refer to a single
element")))else() in sig_1

                    )
                                    with Not_found -> raise(Failure("Unknown id: " ^ s1))

)
```

```
                        )
                                                                              in if(List.exists (fun elt -> fst elt =
s2) (Array.to_list p1)) then (
                                                                                    (* get the type associated
with the reference*)
                                                                                    (match (List.find (fun elt
-> fst elt = s2) (Array.to_list p1)

) with (_,Int) -> SingleVarLiteral(("",Int),LitInt(0))

                                    | (_,Float) -> SingleVarLiteral(("",Float),LitFloat(0.0))

                                    | (_,String) -> SingleVarLiteral(("",String),LitString(""))
                                                                                    )
                                                                              ) else(
                                                                                    raise(Failure("'" ^ s2 ^ "' is
not a valid signature element."))
                                                                              )
(*  | Filter(s1, ex2) -> s1 ^ "[" ^ javaS_of_expr ex2 ^ "]"
*)

in
        let rec javaS_of_expr = function
   Literal(l) -> (match l with LitInt(i) -> string_of_int i | LitFloat(f) -> string_of_float f | LitString(s) -> s)
  | Id(s) -> s
  | Binop(e1, o, e2) as binexp -> (* Here we need to typecheck each expression before generating the code for it.*)
     (try (match (typecheck_of_expr binexp) with
     SingleVarLiteral(( _, Int), _) -> javaS_of_expr e1 ^
                  (match o with Add -> " + " | Sub -> " - " | Mult -> " * " | Div -> " / " | Equal -> " == " | Neq -> " != "
         | Less -> " < " | Leq -> " <= " | Greater -> " > " | Geq -> " >= " (* | And_ -> "&&" | Or_ -> "||"*)
         ) ^ javaS_of_expr e2
     | SingleVarLiteral((_, Float),_) -> javaS_of_expr e1 ^
                  (match o with Add -> " + " | Sub -> " - " | Mult -> " * " | Div -> " / " | Equal -> " == " | Neq -> " != "
         | Less -> " < " | Leq -> " <= " | Greater -> " > " | Geq -> " >= " (* | And_ -> "&&" | Or_ -> "||"*)
         ) ^ javaS_of_expr e2
     | SingleVarLiteral((_, String),_) -> javaS_of_expr e1 ^
                  (match o with Add -> " + " | Equal -> " == " | Neq -> " != " | Less -> " < " | Leq -> " <= " | Greater -> " > "
                  | Geq -> ">= " | _ -> raise(Failure("Can't perform this operation on a string"))(* | And_ -> "&&" | Or_ ->
"||"*)
         ) ^ javaS_of_expr e2
     | FullTableLiteral(_,_,_) -> javaS_of_expr e1 ^ ".signature" ^
         (match o with Equal -> " == " | Neq -> " != " | _ -> raise(Failure("Can't perform this operation on a table"))
         (* | And_ -> "&&" | Or_ -> "||"*)
         ) ^ javaS_of_expr e2 ^ ".signature"

                                            ) with Failure(msg) -> raise(Failure("Can't generate code: " ^ msg ^ "\n"))
                                    )
  | Assign(v, e) as assignexp -> (try (match(typecheck_of_expr assignexp) with
                  _ -> v ^ " = " ^ javaS_of_expr e) with Failure(msg) -> raise(Failure("Can't generate code: " ^ msg ^ "")))
  | TAssign(v, v', e) as assignexp -> (try (match(typecheck_of_expr assignexp) with
                  _ -> javaS_of_expr (GetCol(v, v')) ^ " = " ^ javaS_of_expr e) with Failure(msg) -> raise(Failure("Can't
generate code: " ^ msg ^ "")))
  | Call(f, el) as callexp -> (try (match(typecheck_of_expr callexp) with
                  _ -> f ^ "(" ^ (String.concat ", " (List.map javaS_of_expr el)) ^ ")")
                                    with Invalid_argument(msg) -> raise(Failure("Call to function " ^ f ^ " does not match the
declared argument list for that function: " ^ msg))
                                    | Function_not_found(func) -> (* Code for standard library.*)
```

```
                                (match f with
                                        "print" -> "alert(" ^ (if(List.length el = 1)then(match typecheck_of_expr(List.hd el) with
                                                        SingleVarLiteral(_,_) -> javaS_of_expr (List.hd el)
                                                | FullTableLiteral(_,_,_) -> raise(Failure("Call toDOM on a table instead of
print."))
                                                )else(raise(Failure("Wrong # of parameters passed to print")))) ^ ")"
                                | "toDOM" -> "$$tml$$.toDOM(" ^ (if(List.length el = 1)then(match typecheck_of_expr(List.hd
el) with
                                                        SingleVarLiteral(_,_) -> raise(Failure("Call print on a primitive instead
of toDOM."))
                                                | FullTableLiteral(_,_,_) -> javaS_of_expr (List.hd el)
                                                )else(raise(Failure("Wrong # of parameters passed to toDOM")))) ^ ")"
                                | "delete" -> (if(List.length el = 1)then(match typecheck_of_expr(List.hd el) with
                                                        SingleVarLiteral(_,_) -> let () = print_string "WARNING: deleting a
primitive will have no effect\n"
                                                                in "delete " ^ javaS_of_expr (List.hd el)
                                                | FullTableLiteral(_,_,_) -> "$$tml$$.del(" ^ javaS_of_expr (List.hd el)
                                                )else(raise(Failure("Wrong # of parameters passed to toDOM")))) ^ ")"
                                | _ -> raise(Failure("Function " ^ func ^ " not found"))
                                )
                                )(*End try *)
  | Noexpr -> ""
  (* first thing in GetCol is to call the typechecker.  After running that function, you can be confident that s1 exists, that s2
          refers to a valid element.*)
  | GetCol(s1, s2) as getcolexp -> try(match(typecheck_of_expr getcolexp) with _ -> s1 ^ ".rows[0][" ^ s1 ^ ".headnames." ^
s2 ^ "]" )

                        with Failure(msg) -> raise(Failure("Can't generate code: " ^ msg))
(*  | Filter(s1, ex2) -> s1 ^ "[" ^ javaS_of_expr ex2 ^ "]" *)

          in let rec javaS_of_stmt = function
    Block(stmts) -> (*let () = print_string "HEYME" in *)
      "{\n" ^ String.concat "" (List.map javaS_of_stmt stmts) ^ "}\n"
  | Expr(expr) -> javaS_of_expr expr ^ ";\n"
  | Return(expr) -> (* If the typecheck of the expression doesn't match the returnsig of the enclosing function, then throw an
error*)
                        let () =
                                let encl_func = (StringMap.find thisenv.current_enclosing_function thisenv.function_index) in
                                if encl_func.returnsig <> typecheck_of_expr expr then
                                        raise(Failure("Return statement : " ^ Ast.string_of_expr expr ^ " does not match return
statement of: " ^
                                                thisenv.current_enclosing_function))
                                else ()
                        in "return " ^ javaS_of_expr expr ^ ";\n"
  | If(e, s, Block([])) -> "if (" ^ javaS_of_expr e ^ ")\n" ^ javaS_of_stmt s
  | If(e, s1, s2) ->  "if (" ^ javaS_of_expr e ^ ")\n" ^
      javaS_of_stmt s1 ^ "else\n" ^ javaS_of_stmt s2
  | For(e1, e2, e3, s) ->
      "for (" ^ javaS_of_expr e1  ^ " ; " ^ javaS_of_expr e2 ^ " ; " ^
      javaS_of_expr e3  ^ ") " ^ javaS_of_stmt s
  | While(e, s) -> "while (" ^ javaS_of_expr e ^ ") " ^ javaS_of_stmt s
  | Pipe(s1, s2, stmts) -> let () = if (try(match (StringMap.find s1 thisenv.global_index) with

                                                                        FullTableLiteral(_,_,arry) -> Array.length arry

                                                                | SingleVarLiteral(_,_) -> raise(Failure(s1 ^ " is a global
primitive, not a table\n"))
```

```
                                                                ) with Not_found -> try (match (StringMap.find s1
thisenv.local_index) with

                                                                                    (FullTableLiteral(_,_,arry),_) -> Array.length arry

                                                                            | (SingleVarLiteral(_,_),_) -> raise(Failure(s1 ^ " is a local
primitive, not a table\n"))

                                                                ) with Not_found -> raise(Failure( s1 ^ " is not a valid table
variable.\n"))

                                                        ) = 0 then raise(Failure(s1 ^ " does not have any rows to iterate through.\n"))

                              else (try(match (StringMap.find s1 thisenv.global_index) with

                                                                            (*          FullTableLiteral(_,_,_) as ftl -> thisenv.local_index
<- StringMap.add s2 (ftl,(StringMap.find thisenv.current_enclosing_function thisenv.function_index)) thisenv.local_index
*)

                                                                                    FullTableLiteral(name, sig_n, litrowarry) -> let
newarry = [|litrowarry.(0)|] in

                                                                                            thisenv.local_index <- StringMap.add s2
(FullTableLiteral(name, sig_n, newarry), (StringMap.find thisenv.current_enclosing_function thisenv.function_index))
thisenv.local_index

                                                                            | SingleVarLiteral(_,_) -> raise(Failure(s1 ^ " is a global
primitive, not a table\n"))

                                                                ) with Not_found -> try (match (StringMap.find s1
thisenv.local_index) with

                                                                                    (*(FullTableLiteral(_,_,_),_) as ftl ->
thisenv.local_index <- StringMap.add s2 ftl thisenv.local_index *)

                                                                                    (FullTableLiteral(name, sig_n, litrowarry),encl_func)
-> let newarry = [|litrowarry.(0)|] in thisenv.local_index <- StringMap.add s2 (FullTableLiteral(name, sig_n,
newarry),encl_func) thisenv.local_index

                                                                            | (SingleVarLiteral(_,_),_) -> raise(Failure(s1 ^ " is a local
primitive, not a table\n"))

                                                                ) with Not_found -> raise(Failure( s1 ^ " is not a valid table
variable.\n"))

                                                        )
        (*let () = print_string (s2 ^ " points to table " ^ s1 ^ "\n") in *)
                in "var " ^ s2 ^ " = {};\nfor (var $" ^ s1 ^ " = 0; $" ^ s1 ^ " < " ^ s1 ^ ".rows.length; $" ^ s1 ^ "++ ) {\n" ^
                        s2 ^ ".rows = new Array();\n" ^
                        s2 ^ ".rows[0] = " ^ s1 ^ ".rows[$" ^ s1 ^ "];\n" ^ s2 ^ ".signature = " ^ s1 ^ ".signature;\n" ^
                        s2 ^ ".headnames = " ^ s1 ^ ".headnames;\n" ^ s2 ^ ".headtypes = " ^ s1 ^ ".headtypes;\n" ^
                        (String.concat "" (List.map javaS_of_stmt stmts)) ^ "}\n"
  | LocalDecl(td) -> (match td with SingleVarLiteral((s, _), _) as svl ->
        let () = (*print_string ("found variable: " ^ s ^ "::" ^ thisenv.current_enclosing_function ^ "\n");*)
thisenv.local_index <- StringMap.add s (svl,(StringMap.find thisenv.current_enclosing_function thisenv.function_index))
thisenv.local_index
        in String.concat "" (translate_globals [td])
                                                                                                    | FullTableLiteral(s,_,_) as ftl ->
```

```
                                        let () = thisenv.local_index <- StringMap.add s (ftl,
                                        (StringMap.find thisenv.current_enclosing_function thisenv.function_index))
thisenv.local_index
        in String.concat "" (translate_globals [td])
                                                                                )
 | ColAppend(s1, s2) -> let p1 = (try (match (StringMap.find s1 thisenv.global_index) with

        FullTableLiteral(_,_,_) as ftlit -> (ftlit, "GL")
                                                                                              |       _ ->
raise(Failure("Can't append a column to a global primitive.\n"))

                                 ) with Not_found -> (try (match(StringMap.find s1 thisenv.local_index) with

        (FullTableLiteral(_,_,_) as ftlit,encl_func) -> (ftlit, encl_func.fname)
                                                                                              |       _ ->
raise(Failure("Can't append a column to a local primitive.\n"))


                )

                                               with Not_found -> raise(Failure("Unknown ColAppend id: " ^ s1))


)

                )
                                                                                                   and p2 = (try
(StringMap.find s2 thisenv.global_index) with

                                 Not_found -> (try (match(StringMap.find s2 thisenv.local_index) with
                                                                                                   (ftlit2,_) -> ftlit2

                                                                                                   )

                with Not_found -> raise(Failure("Unknown ColAppend id: " ^ s2))

                                                                                     )

                )
                                               in let p1parts = (match (fst p1) with
FullTableLiteral(_,Signature(sig_n),litrows) -> (sig_n, litrows)

                                                                                     | _ -> raise(Failure("NEVER
CALLED")))
                                               and p2parts = (match p2 with FullTableLiteral(_,Signature(sig_n2),litrows2) ->
(sig_n2, litrows2.(0))

                                               | SingleVarLiteral(sig_n2,onelit) -> ([|sig_n2|], LiteralRow([|onelit|])))
                                               in let newsigarray = Array.append (fst p1parts) (fst p2parts)
                                               and litstoappend = Array.to_list (match (snd p2parts) with
LiteralRow(litrowarray) -> litrowarray)
                                               in let newtablelit = Array.of_list(List.map
                                                      (fun litrow -> LiteralRow(Array.of_list((Array.to_list(match litrow
with LiteralRow(litrowarray2) -> litrowarray2)) @ litstoappend)))
                                                             (Array.to_list (snd p1parts)))
                                               in let () = if (snd p1) = "GL" then thisenv.global_index <- (StringMap.add s1
(FullTableLiteral(s1, Signature(newsigarray), newtablelit)) thisenv.global_index) else thisenv.local_index <-
(StringMap.add s1 ((FullTableLiteral(s1, Signature(newsigarray), newtablelit)),(StringMap.find (snd p1)
thisenv.function_index)) thisenv.local_index)
                                               in
                                               "if(" ^ s2 ^ ".rows) {\n" ^
                                               "for(var $" ^ s2 ^ " in " ^ s2 ^ ".headnames) {\n" ^
                                               "if(" ^ s1 ^ ".headnames[$" ^ s2 ^ "]) {\nthrow \"Duplicate Column
header.\";\n}\nelse {\n" ^
                                               s1 ^ ".headnames[$" ^ s2 ^ "] = " ^ s1 ^ ".headtypes.length;\n" ^
                                               s1 ^ ".headtypes.push(" ^ s2 ^ ".headtypes[" ^ s2 ^ ".headnames[$" ^ s2 ^
"]]);\n}\n" ^
                                               "}\n" ^
                                               s1 ^ ".signature += \"::\" + " ^ s2 ^ ".signature;\n" ^
                "}\n else {" ^
                s1 ^ ".headnames." ^ s2 ^ " = " ^ s1 ^ ".headtypes.length;\n" ^
                                               s1 ^ ".headtypes.push(\"" ^
                                               (match (fst p2parts).(0) with (_,prm) -> (match prm with Int -> "int" | Float ->
"float" | String -> "string")) ^
                                               "\");\n" ^ s1 ^ ".signature += \"::" ^ s2 ^ ":" ^
                                               (match (fst p2parts).(0) with (_,prm) -> (match prm with Int -> "int" | Float ->
"float" | String -> "string")) ^
                                               "\";\n" ^"}\n" ^
                "for (var $" ^ s1 ^ " = 0; $" ^ s1 ^ " < " ^ s1 ^ ".rows.length; $" ^ s1 ^ "++ ) {\n" ^
                "if(" ^ s2 ^ ".rows) {\n" ^ s1 ^ ".rows[$" ^ s1 ^ "] = " ^ s1 ^ ".rows[$" ^ s1 ^ "].concat(" ^ s2 ^
".rows[0]);\n" ^
                "}\n else {" ^ s1 ^ ".rows[$" ^ s1 ^ "].push(" ^ s2 ^ ");\n}" ^ "\n}\n"

        | RowAppend(s1,s2) -> let p1 = (try (match (StringMap.find s1 thisenv.global_index) with

        FullTableLiteral(_,Signature(sig_1),litarry) -> (sig_1, litarry)
                                                                                              |           _ ->
raise(Failure("Can't append a row to a global primitive.\n"))

                                 ) with Not_found -> (try (match(StringMap.find s1 thisenv.local_index) with

        (FullTableLiteral(_,Signature(sig_1),litarry),_)  -> (sig_1, litarry)
                                                                                              |           _ ->
raise(Failure("Can't append a row to a local primitive.\n"))


                )

                                               with Not_found -> raise(Failure("Unknown rowAppend id: " ^ s1))


)

                )
                                                                                                   and p2 = (try
(match (StringMap.find s2 thisenv.global_index) with

        FullTableLiteral(_,Signature(sig_2),litarry2) -> (sig_2,litarry2)
                                                                                              | _ -> raise(Failure("Can't
append a row to a global primitive.\n"))

                                 ) with Not_found -> (try (match(StringMap.find s2 thisenv.local_index) with

        (FullTableLiteral(_,Signature(sig_2),litarry2),_) -> (sig_2,litarry2)
                                                                                              | _ -> raise(Failure("Can't
append a row to a local primitive.\n"))

                    )

                                        with Not_found -> raise(Failure("Unknown rowAppend id: " ^ s2))


)

                    )
                                                                                      in if(fst p1 <> fst p2) then (
                              raise(Failure("\nType mismatch of rowAppend: \n" ^ s1 ^ "\n***\n" ^ s2))) else (
                                        let () = (try (match (StringMap.find s1 thisenv.global_index) with

          FullTableLiteral(s,sig_n,_) -> (thisenv.global_index <-

          (StringMap.add s1 (FullTableLiteral(s, sig_n, (Array.append (snd p1) (snd p2)))) thisenv.global_index))
                                                                                                |         _ -> ()

                              ) with Not_found -> (try (match(StringMap.find s1 thisenv.local_index) with

          (FullTableLiteral(s,sig_n,_),encl_func) -> (thisenv.local_index <-

          (StringMap.add s1 (FullTableLiteral(s, sig_n, (Array.append (snd p1) (snd p2))), encl_func) thisenv.local_index))
                                                                                                |         _ -> ()


                    )

                                        with _ -> ()


)

                    ) in
                    "for (var $" ^ s2 ^ " = 0; $" ^ s2 ^ " < " ^ s2 ^ ".rows.length; $" ^ s2 ^ "++ ) {\n" ^
                                        s1 ^ ".rows.push($$tml$$.cpy(" ^ s2 ^ ".rows[$" ^ s2 ^ "]));\n}\n"
                                        )
          in
                    let () = for i = 0 to (List.length fundecl.formals) - 1 do
                              let localvars = Array.of_list fundecl.formals
                              in match localvars.(i) with
                                        (s, Int) -> thisenv.local_index <- (StringMap.add s (SingleVarLiteral((s, Int),
LitInt(0)),fundecl) thisenv.local_index)
                              |         (s, Float) -> thisenv.local_index <- (StringMap.add s (SingleVarLiteral((s, Float),
LitFloat(0.0)),fundecl) thisenv.local_index)
                              | (s, String) -> thisenv.local_index <- (StringMap.add s (SingleVarLiteral((s, String),
LitString("")),fundecl) thisenv.local_index)
                    done
                    in
                    let func_string = "function " ^ fundecl.fname ^ "(" ^ (*(let () = print_string
thisenv.current_enclosing_function in "") ^*)
                    (String.concat ", " (List.map (fun (s, p) -> s) fundecl.formals)) ^ ")\n{\n" ^ (*(let () = print_string "POP" in
"") ^ *)
                    (String.concat "" (List.map javaS_of_stmt fundecl.body)) ^ "\n}\n" (*^ (let () = print_string "GOES" in "")
*)
                    (* Now remove all the local variables that were declared in this function *)
```

```
                in
                let () = (*print_string "BBB";*) StringMap.iter
                                        (fun key (_, encl_func) -> if(thisenv.current_enclosing_function =
encl_func.fname)then(
                                                thisenv.local_index <- StringMap.remove key thisenv.local_index
                                        )else()
                                        ) thisenv.local_index(*; print_string "AAA"*)
                in func_string
                (* End of declarations within translate, now time to set environment and call the function *)
                in let thisenv = { function_index = global_func_decls;
                                                                                global_index =
global_table_decls;
                                                                                local_index =
StringMap.empty;

current_enclosing_function = "";
                                                                }
        in (* First we need to import the standard library *)
                ("if(!$$tml$$) { throw new Error(\"STANDARD LIBRARY NOT FOUND\"); }\n") ^
                (* Next we translate globals, then functions *)
                (String.concat "" (List.rev (translate_globals globals))) ^ String.concat "" (List.map (translate thisenv)
functions)
                (* If it's the main function, invoke it *)
                ^ (if List.exists (fun fnc ->if(fnc.fname = "main")then true else false) functions then "\nmain();\n" else
                                raise(Failure("No main function found")))
```

```
{ open Parser }

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "/*"     { comment lexbuf }           (* Comments *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "return" { RETURN }
| "int"    { INT }
| "|"                       { PIPE }
| "#{"              { LTABLE }
| "}#"              { RTABLE }
| "#"                       { TABLESEP }
| "::"              { COLAPPEND }
(*| "&&"                    { LOGICAL_AND }
| "||"              { LOGICAL_OR } *)
| "<<"              { ROWAPPEND }
| "."                       { DOTOP }
(*| "["                     { LBRACK }
| "]"                       { RBRACK } *)
| "string" { STRING }
| "float"  { FLOAT }
| "table"  { TABLE }
| "function" { FUNCTION }
| ['0'-'9']+"."['0'-'9']+ as float_lxm { FLOAT_LITERAL(float_of_string float_lxm) }
| "\""['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_' ' ']*"\"" as string_lxm { STRING_LITERAL(string_lxm) }
| ['0'-'9']+ as lxm { LITERAL(int_of_string lxm) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as lxm { ID(lxm) }
| eof { EOF }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
%token PLUS MINUS TIMES DIVIDE ASSIGN
%token EQ NEQ LT LEQ GT GEQ
%token RETURN IF ELSE FOR WHILE INT
%token STRING FLOAT
%token LTABLE RTABLE TABLESEP
/*%token LBRACK RBRACK */
%token PIPE
%token DOTOP
%token ROWAPPEND
%token COLAPPEND
%token FUNCTION
%token TABLE
%token <float> FLOAT_LITERAL
%token <string> STRING_LITERAL
%token <int> LITERAL
%token <string> ID
%token EOF

%nonassoc NOELSE NOASSIGN
%nonassoc ELSE
%right ASSIGN
/*%left LOGICAL_AND LOGICAL_OR*/
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE

%start program
%type <Ast.program> program

%%

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

/* start modified part */
tdecl:
            primitive_decl ASSIGN literal SEMI            { SingleVarLiteral($1, $3) }
  | INT ID SEMI
            { SingleVarLiteral(($2,Int), LitInt(0)) }
  | STRING ID SEMI
            { SingleVarLiteral(($2,String), LitString("")) }
  | FLOAT ID SEMI
            { SingleVarLiteral(($2,Float), LitFloat(0.0)) }
            | TABLE ID ASSIGN LTABLE typelist RTABLE SEMI { FullTableLiteral($2, Signature(Array.of_list(List.rev
$5)), [||]) }
            | TABLE ID ASSIGN LTABLE typelist TABLESEP tableliteral RTABLE SEMI { FullTableLiteral( $2,
Signature(Array.of_list(List.rev $5)), Array.of_list(List.rev $7)) }

primitive_decl:
             INT ID
            { ($2,Int) }
            |           STRING ID
            { ($2,String) }
            | FLOAT ID
            { ($2,Float) }

literal:
             FLOAT_LITERAL
            { LitFloat($1) }
            | STRING_LITERAL                                                              { LitString($1) }
            | LITERAL
            { LitInt($1) }

typelist:
            | primitive_decl                                                          { $1 :: [] }
            | typelist COMMA primitive_decl   { $3 :: $1 }

tableliteral:
                        valuesequence
            {LiteralRow( Array.of_list (List.rev $1)) :: []}
  | tableliteral TABLESEP valuesequence     {LiteralRow( Array.of_list (List.rev $3)) :: $1}

valuesequence:
                        literal
            {[$1]}
            | valuesequence COMMA literal                              {$3 :: $1}

fdecl:
             FUNCTION returntype ID LPAREN formals_opt RPAREN LBRACE stmt_list RBRACE
     { { fname = $3;
     returnsig = $2;
             formals = $5;
             body = List.rev $8 } }

returntype:
                        INT                                                         { SingleVarLiteral(("",Int),
LitInt(0)) }
            | STRING                                              { SingleVarLiteral(("",String), LitString("")) }
            | FLOAT                                                       { SingleVarLiteral(("",Float),
LitFloat(0.0)) }
            | TABLE LTABLE typelist RTABLE                     { FullTableLiteral("", Signature(Array.of_list(List.rev $3)),
[||]) }

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

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

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

stmt:
    expr SEMI { Expr($1) }
  | 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_opt SEMI expr_opt RPAREN stmt
     { For($3, $5, $7, $9) }
  | WHILE LPAREN expr RPAREN stmt { While($3, $5) }
  | ID PIPE ID LBRACE stmt_list RBRACE { Pipe($1, $3, $5) }
  | tdecl { LocalDecl($1) }
  | ID ROWAPPEND ID SEMI  { RowAppend($1, $3) }
  | ID COLAPPEND ID     SEMI     { ColAppend($1, $3) }

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

expr:
    literal          { Literal($1) }
  | 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 LOGICAL_AND expr { Binop($1, And_, $3) }
  | expr LOGICAL_OR expr { Binop($1, Or_, $3) } */
  | ID ASSIGN expr   { Assign($1, $3) }
  | ID DOTOP ID ASSIGN expr { TAssign($1,$3,$5) }
  | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
  | LPAREN expr RPAREN { $2 }
  | ID DOTOP ID                                    { GetCol($1, $3) }
/*  | ID LBRACK expr RBRACK { Filter($1, $3) } */

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

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

```
(*type action = Ast | Interpret | Bytecode | Compile*)
(* ("-i", Interpret);
("-b", Bytecode);
  | Interpret -> ignore (Interpret.run program)
  | Bytecode -> let listing =
      Bytecode.string_of_prog (Compile.translate program)
    in print_endline listing
  | Compile -> Execute.execute_prog (Compile.translate program)
*)

type action = Ast | Compile

let _ =
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);
                                            ("-c", Compile) ]
  else Compile in
  let filename = if Array.length Sys.argv > 2 then
            Sys.argv.(2) else "LaMesa_" ^ string_of_int (Random.int 100) ^ ".js" in
  let lexbuf = Lexing.from_channel stdin in
  let program = Parser.program Scanner.token lexbuf in
  match action with
    Ast -> let listing = Ast.string_of_program program
          in print_string listing
  | Compile -> let outfile = open_out filename in
                                                                        output_string outfile (Compile.translate program)
```

```
var $$tml$$ = {};

/* OLD OCAML CODE
(*          | FullTableNoLiteral(tbname1, tbsig1) :: tl -> let javaout = "var " ^ tbname1 ^ " = {};\n" ^
                              tbname1 ^ ".root = document.createElement(\"table\");\n" ^ tbname1 ^
".root.setAttribute(\"id\",\"" ^ tbname1 ^ "\");\n" ^
                              "document.body.appendChild(" ^ tbname1 ^ ".root);\n" ^ tbname1 ^ ".rows = new Array();\n" ^
tbname1 ^ ".head = " ^ tbname1 ^
                              ".root.createTHead();\n" ^
                              (* VARNAME.body = VARNAME.root.appendChild(document.createElement('tbody')); *)
                              tbname1 ^ ".body = " ^ tbname1 ^ ".root.appendChild(document.createElement('tbody'));\n"^
                              (* VARNAME.headrow = VARNAME.head.insertRow(0); *)
                              tbname1 ^ ".headrow = " ^ tbname1 ^ ".head.insertRow(0);\n" ^
                              (* test_lookup.headcells = new Array(); *)
                              tbname1 ^ ".headcells = new Array();\n"
                                     in javaout :: tdecl_java_build tl *)
*/
$$tml$$.del = function(variable) {
        if(variable.nameString) {
                if(document.getElementById(variable.nameString)) {
                        document.body.removeChild(document.getElementById(variable.nameString));
                }
        }
}
// Copy constructor for an array
$$tml$$.cpy = function(TmlRow) {
        newTmlRow = new Array();
        for(var $i = 0; $i < TmlRow.length; $i++) {
                newTmlRow[$i] = TmlRow[$i];
        }
        return newTmlRow;
}


$$tml$$.toDOM = function(tmltable) {
        var domTable = {};
        domTable.root = document.createElement("table");
        domTable.root.setAttribute("id",tmltable.nameString);
        domTable.rows = new Array();
        domTable.head = domTable.root.createTHead();
        domTable.body = domTable.root.appendChild(document.createElement('tbody'));
        //Create the <tr> in the head part of the table
        domTable.headrow = domTable.head.insertRow(0);
        domTable.headcells = new Array();

        /* OLD OCAML CODE
        let javaS_of_singletontype tbname5 (id, prim) =

        (*VARNAME.headcells.push(VARNAME.headrow.appendChild(document.createElement('th'))); *)

        tbname5 ^ ".headcells.push(" ^ tbname5 ^

".headrow.appendChild(document.createElement('th')));\n" ^
                                     (*VARNAME.headcells[VARNAME.headcells.length - 1].innerHTML =
"test_number"; *)

        tbname5 ^ ".headcells[" ^ tbname5 ^ ".headcells.length - 1].innerHTML = \"" ^ id ^ "\";\n" ^
                                     (*test_lookup.headcells[test_lookup.headcells.length - 1].id = "test_number";*)
```

```
		tbname5 ^ ".headcells[" ^ tbname5 ^ ".headcells.length - 1].id = \"" ^ tbname5 ^ "_" ^ id ^ "\";\n" ^
						(*test_lookup.headcells[test_lookup.headcells.length - 1].className = "int";*)

		tbname5 ^ ".headcells[" ^ tbname5 ^ ".headcells.length - 1].className = \"" ^


		(match prim with Int -> "int" | Float -> "float" | String -> "string") ^ "\";\n"
		*/
		var j;
		for(var i in tmltable.headnames) {
			j = tmltable.headnames[i];
			domTable.headcells[j] = domTable.headrow.appendChild(document.createElement('th'));
			domTable.headcells[j].innerHTML = i;
			domTable.headcells[j].id = i;
			domTable.headcells[j].className = tmltable.headtypes[tmltable.headnames[i]];
		}

		/*OLD OCAML CODE
(* function string_of_mandatory_lits takes primitive types and converts them into strings *)
let javaS_of_tablerow tbname2 = function
		LiteralRow(lits) -> tbname2 ^ ".rows.push(" ^ tbname2 ^ ".body.insertRow(" ^ tbname2 ^ ".rows.length));\n" ^
(String.concat "" (Array.to_list (Array.map (javaS_of_mandatory_lit tbname2) lits)))
*)
(*
let javaS_of_mandatory_lit tbname4 = function
		 LitInt(i) -> "var " ^ tbname4 ^ "_ = " ^ tbname4 ^ ".rows[" ^ tbname4 ^ ".rows.length - 1].insertCell(" ^ tbname4
^ ".rows[" ^ tbname4 ^ ".rows.length - 1].cells.length);\n" ^ tbname4 ^ "_.innerHTML = " ^ string_of_int(i) ^ ";\n" ^
tbname4 ^ "_.id = \"" ^
		 tbname4 ^ "_r\" + " ^
		 tbname4 ^ ".rows[" ^ tbname4 ^ ".rows.length - 1].rowIndex + \"c\" + " ^ tbname4 ^ "_.cellIndex;\n"
		| LitFloat(f) -> "var " ^ tbname4 ^ "_ = " ^ tbname4 ^ ".rows[" ^ tbname4 ^ ".rows.length - 1].insertCell(" ^
tbname4 ^ ".rows[" ^ tbname4 ^ ".rows.length - 1].cells.length);\n" ^ tbname4 ^ "_.innerHTML = " ^ string_of_float(f) ^
";\n" ^ tbname4 ^"_.id = \"" ^
			tbname4 ^ "_r\" + " ^
		 tbname4 ^ ".rows[" ^ tbname4 ^ ".rows.length - 1].rowIndex + \"c\" + " ^ tbname4 ^ "_.cellIndex;\n"
		| LitString(s) -> "var " ^ tbname4 ^ "_ = " ^ tbname4 ^ ".rows[" ^ tbname4 ^ ".rows.length - 1].insertCell(" ^
tbname4 ^ ".rows[" ^ tbname4 ^ ".rows.length - 1].cells.length);\n" ^ tbname4 ^ "_.innerHTML = " ^ s ^ ";\n" ^ tbname4 ^
"_.id = \"" ^
			tbname4 ^ "_r\" + " ^
		 tbname4 ^ ".rows[" ^ tbname4 ^ ".rows.length - 1].rowIndex + \"c\" + " ^ tbname4 ^ "_.cellIndex;\n"
 */
		var k;
		for(var i = 0; i < tmltable.rows.length; i++) {
			domTable.rows[i] = domTable.body.insertRow(i);
			for(var j = 0; j < tmltable.rows[i].length; j++) {
				k = domTable.rows[i].insertCell(j);
				k.innerHTML = tmltable.rows[i][j];
			}
		}



		document.body.appendChild(domTable.root);

}
```

```
if(!$$tml$$) { throw new Error("STANDARD LIBRARY NOT FOUND"); }
var test_lookup = {};
test_lookup.nameString = "test_lookup";
test_lookup.rows = new Array();
test_lookup.headnames = new Array();
test_lookup.headtypes = new Array();
test_lookup.signature = "test_number:int::test_begin_page:int::test_end_page:int";
test_lookup.headnames.test_number = test_lookup.headtypes.length;
test_lookup.headtypes[test_lookup.headtypes.length] = "int";
test_lookup.headnames.test_begin_page = test_lookup.headtypes.length;
test_lookup.headtypes[test_lookup.headtypes.length] = "int";
test_lookup.headnames.test_end_page = test_lookup.headtypes.length;
test_lookup.headtypes[test_lookup.headtypes.length] = "int";
test_lookup.rows[test_lookup.rows.length] = new Array();
test_lookup.rows[test_lookup.rows.length - 1][test_lookup.rows[test_lookup.rows.length - 1].length] = 1;
test_lookup.rows[test_lookup.rows.length - 1][test_lookup.rows[test_lookup.rows.length - 1].length] = 389;
test_lookup.rows[test_lookup.rows.length - 1][test_lookup.rows[test_lookup.rows.length - 1].length] = 431;
test_lookup.rows[test_lookup.rows.length] = new Array();
test_lookup.rows[test_lookup.rows.length - 1][test_lookup.rows[test_lookup.rows.length - 1].length] = 2;
test_lookup.rows[test_lookup.rows.length - 1][test_lookup.rows[test_lookup.rows.length - 1].length] = 451;
test_lookup.rows[test_lookup.rows.length - 1][test_lookup.rows[test_lookup.rows.length - 1].length] = 493;
test_lookup.rows[test_lookup.rows.length] = new Array();
test_lookup.rows[test_lookup.rows.length - 1][test_lookup.rows[test_lookup.rows.length - 1].length] = 8;
test_lookup.rows[test_lookup.rows.length - 1][test_lookup.rows[test_lookup.rows.length - 1].length] = 823;
test_lookup.rows[test_lookup.rows.length - 1][test_lookup.rows[test_lookup.rows.length - 1].length] = 865;
var test_lookup2 = {};
test_lookup2.nameString = "test_lookup2";
test_lookup2.rows = new Array();
test_lookup2.headnames = new Array();
test_lookup2.headtypes = new Array();
test_lookup2.signature = "test_number:int::test_begin_page:int::test_end_page:int";
test_lookup2.headnames.test_number = test_lookup2.headtypes.length;
test_lookup2.headtypes[test_lookup2.headtypes.length] = "int";
test_lookup2.headnames.test_begin_page = test_lookup2.headtypes.length;
test_lookup2.headtypes[test_lookup2.headtypes.length] = "int";
test_lookup2.headnames.test_end_page = test_lookup2.headtypes.length;
test_lookup2.headtypes[test_lookup2.headtypes.length] = "int";
test_lookup2.rows[test_lookup2.rows.length] = new Array();
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 3;
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 513;
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 555;
test_lookup2.rows[test_lookup2.rows.length] = new Array();
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 4;
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 575;
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 617;
test_lookup2.rows[test_lookup2.rows.length] = new Array();
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 5;
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 637;
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 679;
test_lookup2.rows[test_lookup2.rows.length] = new Array();
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 6;
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 699;
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 741;
test_lookup2.rows[test_lookup2.rows.length] = new Array();
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 7;
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 761;
test_lookup2.rows[test_lookup2.rows.length - 1][test_lookup2.rows[test_lookup2.rows.length - 1].length] = 803;
var question_info = {};
```

```
question_info.nameString = "question_info";
question_info.rows = new Array();
question_info.headnames = new Array();
question_info.headtypes = new Array();
question_info.signature = "prb_type:string::prb_num:int::pg_num:int::diff:string";
question_info.headnames.prb_type = question_info.headtypes.length;
question_info.headtypes[question_info.headtypes.length] = "string";
question_info.headnames.prb_num = question_info.headtypes.length;
question_info.headtypes[question_info.headtypes.length] = "int";
question_info.headnames.pg_num = question_info.headtypes.length;
question_info.headtypes[question_info.headtypes.length] = "int";
question_info.headnames.diff = question_info.headtypes.length;
question_info.headtypes[question_info.headtypes.length] = "string";
question_info.rows[question_info.rows.length] = new Array();
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = "abs_value";
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = 15;
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = 418;
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = "medium";
question_info.rows[question_info.rows.length] = new Array();
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = "abs_value";
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = 17;
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = 456;
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = "medium";
question_info.rows[question_info.rows.length] = new Array();
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = "average";
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = 18;
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = 835;
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = "hard";
question_info.rows[question_info.rows.length] = new Array();
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = "equations";
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = 6;
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = 454;
question_info.rows[question_info.rows.length - 1][question_info.rows[question_info.rows.length - 1].length] = "easy";
function main()
{
var xtd_question_info = {};
xtd_question_info.nameString = "xtd_question_info";
xtd_question_info.rows = new Array();
xtd_question_info.headnames = new Array();
xtd_question_info.headtypes = new Array();
xtd_question_info.signature = "prb_num:int::pg_num:int::test_num:int";
xtd_question_info.headnames.prb_num = xtd_question_info.headtypes.length;
xtd_question_info.headtypes[xtd_question_info.headtypes.length] = "int";
xtd_question_info.headnames.pg_num = xtd_question_info.headtypes.length;
xtd_question_info.headtypes[xtd_question_info.headtypes.length] = "int";
xtd_question_info.headnames.test_num = xtd_question_info.headtypes.length;
xtd_question_info.headtypes[xtd_question_info.headtypes.length] = "int";
xtd_question_info.rows[xtd_question_info.rows.length] = new Array();
$$tml$$.toDOM(question_info);
$$tml$$.toDOM(test_lookup);
$$tml$$.toDOM(xtd_question_info);
var test_num = 0;
alert(test_num);
$$tml$$.del(xtd_question_info);
var xtd_question_row = {};
xtd_question_row.nameString = "xtd_question_row";
xtd_question_row.rows = new Array();
xtd_question_row.headnames = new Array();
xtd_question_row.headtypes = new Array();
xtd_question_row.signature = "prb_num:int::pg_num:int";
xtd_question_row.headnames.prb_num = xtd_question_row.headtypes.length;
xtd_question_row.headtypes[xtd_question_row.headtypes.length] = "int";
xtd_question_row.headnames.pg_num = xtd_question_row.headtypes.length;
xtd_question_row.headtypes[xtd_question_row.headtypes.length] = "int";
xtd_question_row.rows[xtd_question_row.rows.length] = new Array();
xtd_question_row.rows[xtd_question_row.rows.length - 1][xtd_question_row.rows[xtd_question_row.rows.length -
1].length] = 0;
xtd_question_row.rows[xtd_question_row.rows.length - 1][xtd_question_row.rows[xtd_question_row.rows.length -
1].length] = 0;
if(test_num.rows) {
for(var $test_num in test_num.headnames) {
if(xtd_question_row.headnames[$test_num]) {
throw "Duplicate Column header.";
}
else {
xtd_question_row.headnames[$test_num] = xtd_question_row.headtypes.length;
xtd_question_row.headtypes.push(test_num.headtypes[test_num.headnames[$test_num]]);
}
}
xtd_question_row.signature += "::" + test_num.signature;
}
 else {xtd_question_row.headnames.test_num = xtd_question_row.headtypes.length;
xtd_question_row.headtypes.push("int");
xtd_question_row.signature += "::test_num:int";
}
for (var $xtd_question_row = 0; $xtd_question_row < xtd_question_row.rows.length; $xtd_question_row++ ) {
if(test_num.rows) {
xtd_question_row.rows[$xtd_question_row] = xtd_question_row.rows[$xtd_question_row].concat(test_num.rows[0]);
}
 else {xtd_question_row.rows[$xtd_question_row].push(test_num);
}
}
$$tml$$.toDOM(xtd_question_row);
alert(test_num);
$$tml$$.del(xtd_question_row);
var question_row = {};
for (var $question_info = 0; $question_info < question_info.rows.length; $question_info++ ) {
question_row.rows = new Array();
question_row.rows[0] = question_info.rows[$question_info];
question_row.signature = question_info.signature;
question_row.headnames = question_info.headnames;
question_row.headtypes = question_info.headtypes;
var test_lookup_row = {};
for (var $test_lookup = 0; $test_lookup < test_lookup.rows.length; $test_lookup++ ) {
test_lookup_row.rows = new Array();
test_lookup_row.rows[0] = test_lookup.rows[$test_lookup];
test_lookup_row.signature = test_lookup.signature;
test_lookup_row.headnames = test_lookup.headnames;
test_lookup_row.headtypes = test_lookup.headtypes;
if (question_row.rows[0][question_row.headnames.pg_num] >= test_lookup_row.rows[0]
[test_lookup_row.headnames.test_begin_page])
{
if (question_row.rows[0][question_row.headnames.pg_num] <= test_lookup_row.rows[0]
[test_lookup_row.headnames.test_end_page])
{
xtd_question_row.rows[0][xtd_question_row.headnames.test_num] = test_lookup_row.rows[0]
[test_lookup_row.headnames.test_number];
xtd_question_row.rows[0][xtd_question_row.headnames.prb_num] = question_row.rows[0]
[question_row.headnames.prb_num];
xtd_question_row.rows[0][xtd_question_row.headnames.pg_num] = question_row.rows[0]
[question_row.headnames.pg_num];
for (var $xtd_question_row = 0; $xtd_question_row < xtd_question_row.rows.length; $xtd_question_row++ ) {
xtd_question_info.rows.push($$tml$$.cpy(xtd_question_row.rows[$xtd_question_row]));
}
}
}
}
}
$$tml$$.toDOM(xtd_question_info);
return 0;

}

main();
```

```
OBJS = ast.cmo parser.cmo scanner.cmo compile.cmo microc.cmo

TESTS = \
arith1 \
arith2 \
fib \
for1 \
func1 \
func2 \
func3 \
gcd \
global1 \
hello \
if1 \
if2 \
if3 \
if4 \
ops1 \
var1 \
while1

TARFILES = Makefile testall.sh scanner.mll parser.mly \
	ast.ml bytecode.ml interpret.ml compile.ml execute.ml microc.ml \
	$(TESTS:%=tests/test-%.mc) \
	$(TESTS:%=tests/test-%.out)

microc : $(OBJS)
	ocamlc -o microc $(OBJS)

.PHONY : test
test : microc testall.sh
	./testall.sh

scanner.ml : scanner.mll
	ocamllex scanner.mll

parser.ml parser.mli : parser.mly
	ocamlyacc parser.mly

%.cmo : %.ml
	ocamlc -c $<

%.cmi : %.mli
	ocamlc -c $<

microc.tar.gz : $(TARFILES)
	cd .. && tar czf microc/microc.tar.gz $(TARFILES:%=microc/%)

.PHONY : clean
clean :
	rm -f microc parser.ml parser.mli scanner.ml testall.log \
	*.cmo *.cmi *.out *.diff

# Generated by ocamldep *.ml *.mli
ast.cmo:
ast.cmx:
#bytecode.cmo: ast.cmo
#bytecode.cmx: ast.cmx
```

```
#compile.cmo: bytecode.cmo ast.cmo
#compile.cmx: bytecode.cmx ast.cmx
compile.cmo: ast.cmo
compile.cmx: ast.cmx
#execute.cmo: bytecode.cmo ast.cmo
#execute.cmx: bytecode.cmx ast.cmx
#interpret.cmo: ast.cmo
#interpret.cmx: ast.cmx
#microc.cmo: scanner.cmo parser.cmi interpret.cmo execute.cmo compile.cmo \
#    bytecode.cmo ast.cmo
#microc.cmx: scanner.cmx parser.cmx interpret.cmx execute.cmx compile.cmx \
#    bytecode.cmx ast.cmx
microc.cmo: scanner.cmo parser.cmi compile.cmo ast.cmo
microc.cmx: scanner.cmx parser.cmx compile.cmx ast.cmx

parser.cmo: ast.cmo parser.cmi
parser.cmx: ast.cmx parser.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
parser.cmi: ast.cmo
```

```
/* test-001.lm
 * Validate that only correct literals are assigned to global variables.*/

float a = 1.0;
float b;

int c = 4;
int d;

string e = "funny";
string f;
```

```
/* lamesa002.lm -- test that only correct literals are assigned to a table */

table test_lookup = #{ int test_number, int test_begin_page, float test_end_page
# 1         , 389      , 431.0
# 2         ,            451, 493.0
# 8 , 823, 865.0 }#;
```

```
/* lamesa003.lm -- A more robust test combining tests 002 and 001. */

table test_lookup = #{ int test_number, int test_begin_page, float test_end_page
# 1         , 389       , 431.0
# 2         ,           451, 493.0
# 8 , 823, 865.0 }#;

table sobeit = #{ int page, string passage, float avg_happiness
# 237, "and_thus", 4.7
# 237, "and_thus", 4.7
}#;
```

```
/* This is a basic test of function usage. */
float e = 2.71;
float pi = 3.14159;
float pi2;


function string main() {
        pi2 = 22.0 / 7.0;
}
```

```
/* This is a more robust test of different language constructs. */
float pi = 3.14159;
float pi2;
string approx;
int error;

function string main() {
	pi2 = 22.0 / 7.0;
	if(pi2 - pi > 0.0) {
		approx = "more";
		return "correct";
	}
	else {
		approx = "less";
	}
	return approx;
}
```

```
/* This tests local declarations. */
float pi = 3.14159;

function float main() {
        float pi2;
        return pi;
}
```

```
/* This tests local declarations. */
float pi;

function float pi_lookup() {
	float pi3;
	pi = 3.14159;
	pi3 = pi;
}

function float main() {
	float pi2;
	pi2 = pi_lookup();
	return pi_lookup();
}
```

```
/* This function tests table return type */

table test_lookup = #{ int test_number, int test_begin_page, int test_end_page
# 1         , 389      , 431
# 2         ,             451, 493
# 8 , 823, 865 }#;

table test_lookup2 = #{ int test_number, int test_begin_page, int test_end_page
# 3 , 513, 555
# 4 , 575, 617
# 5 , 637, 679
# 6 , 699, 741
# 7 , 761, 803 }#;


function table #{ int test_number, int test_begin_page, int test_end_page }# main() {
        test_lookup2;
}
```

```
/* This function tests table signature comparisons. */

table test_lookup = #{ int test_number, int test_begin_page, int test_end_page
# 1        , 389      , 431
# 2        ,            451, 493
# 8 , 823, 865 }#;

table test_lookup2 = #{ int test_number, int test_begin_page, int test_end_page
# 3 , 513, 555
# 4 , 575, 617
# 5 , 637, 679
# 6 , 699, 741
# 7 , 761, 803 }#;


function table #{ int test_number, int test_begin_page, int test_end_page }# main() {
            return test_lookup == test_lookup2;
}
```

```
/* This function tests table signature comparisons. */

table test_lookup = #{ int test_number, int test_begin_page, int test_end_page
# 1         , 389      , 431
# 2         ,             451, 493
# 8 , 823, 865 }#;

table test_lookup2 = #{ int test_number, int test_begin_page, int test_end_page
# 3 , 513, 555
# 4 , 575, 617
# 5 , 637, 679
# 6 , 699, 741
# 7 , 761, 803 }#;

table test_blarg = #{ int test_number, int test_begin_page, int test_end_blarg }#;

function table #{ int test_number, int test_begin_page, int test_end_page }# main() {
        return test_lookup != test_lookup2;
}
```

```
/* This function tests the pipe operator. */

table test_lookup = #{ int test_number, int test_begin_page, int test_end_page
# 1         , 389      , 431
# 2         ,             451, 493
# 8 , 823, 865 }#;

table test_blarg = #{ int test_number, int test_begin_page, int test_end_page }#;

int test7;

function int main() {
            int test4;
            table test_lookup2 = #{ int test_number, int test_begin_page, int test_end_page
            # 3 , 513, 555
            # 4 , 575, 617
            # 5 , 637, 679
            # 6 , 699, 741
            # 7 , 761, 803 }#;
            test_lookup2 | rowOftest_lookup {
                        test4 = test4 + 1;
            }
            test_blarg = get_test();
            return test4;
}

function table #{ int test_number, int test_begin_page, int test_end_page }# get_test() {
            return test_lookup;
}
```

```
/* This function tests the pipe operator. */

table test_lookup = #{ int test_number, int test_begin_page, int test_end_page
# 1         , 389      , 431
# 2         ,            451, 493
# 8 , 823, 865 }#;

table test_blarg = #{ int test_number, int test_begin_page, int test_end_page
# 7, 761, 803}#;

int test7;

function int main() {
            int test4;
            table test_lookup2 = #{ int test_number, int test_begin_page, int test_end_page
            # 3 , 513, 555
            # 4 , 575, 617
            # 5 , 637, 679
            # 6 , 699, 741
            # 7 , 761, 803 }#;
            test_lookup2 | rowOftest_lookup {
                        test4 = rowOftest_lookup.test_end_page;
            }
            test_blarg = get_test();
            return test4;
}

function table #{ int test_number, int test_begin_page, int test_end_page }# get_test() {
            return test_lookup;
}
```

```
/* This function tests the rowAppend operator. */

table test_lookup = #{ int test_number, int test_begin_page, int test_end_page
# 1         , 389      , 431
# 2         ,              451, 493
# 8 , 823, 865 }#;

table test_blarg = #{ int test_number, int test_begin_page, int test_end_page
# 7, 761, 803}#;

int test7;

function int main() {
        int test4;
        table test_lookup2 = #{ int test_number, int test_begin_page, int test_end_page
        # 3 , 513, 555
        # 4 , 575, 617
        # 5 , 637, 679
        # 6 , 699, 741
        # 7 , 761, 803 }#;
        test_lookup2 | rowOftest_lookup {
                if(rowOftest_lookup.test_number != 7) {
                        test_blarg << rowOftest_lookup;
                }
        }
        test_blarg = get_test();
        return test4;
}

function table #{ int test_number, int test_begin_page, int test_end_page }# get_test() {
        return test_lookup;
}
```

```
/* This function tests formal parameters */

table test_lookup = #{ int test_number, int test_begin_page, int test_end_page
# 1         , 389      , 431
# 2         ,          451, 493
# 8 , 823, 865 }#;

table test_blarg = #{ int test_number, int test_begin_page, int test_end_page
# 7, 761, 803}#;

table test_lookup2 = #{ int test_number, int test_begin_page, int test_end_page
        # 3 , 513, 555
        # 4 , 575, 617
        # 5 , 637, 679
        # 6 , 699, 741
        # 7 , 761, 803 }#;

int test4 = 4;
int test6 = 4;

function int main() {
        return get_test(test4, test6);
}

function int get_test(int test_request, int test_request2) {
        if(test_request == test_request2) {
                return test_request;
        } else {
                return 0;
        }
}
```

```
/* This function tests the print function in standard library. */

table test_lookup = #{ int test_number, int test_begin_page, int test_end_page
# 1         , 389      , 431
# 2         ,            451, 493
# 8 , 823, 865 }#;

table test_blarg = #{ int test_number, int test_begin_page, int test_end_page
# 7, 761, 803}#;

table test_lookup2 = #{ int test_number, int test_begin_page, int test_end_page
            # 3 , 513, 555
            # 4 , 575, 617
            # 5 , 637, 679
            # 6 , 699, 741
            # 7 , 761, 803 }#;

int test4 = 4;
int test6 = 6;

function int main() {
            print(test4);
            return get_test(test4, test6);
}

function int get_test(int test_request, int test_request2) {
            if(test_request + 2 == test_request2) {
                        return test_request;
            } else {
                        return 0;
            }
}
```

```
/* This function tests the toDOM function in standard library. */

table test_lookup = #{ int test_number, int test_begin_page, int test_end_page
# 1         , 389      , 431
# 2         ,           451, 493
# 8 , 823, 865 }#;

table test_blarg = #{ int test_number, int test_begin_page, int test_end_page
# 7, 761, 803}#;

table test_lookup2 = #{ int test_number, int test_begin_page, int test_end_page
        # 3 , 513, 555
        # 4 , 575, 617
        # 5 , 637, 679
        # 6 , 699, 741
        # 7 , 761, 803 }#;

int test4 = 4;
int test6 = 6;

function int main() {
        toDOM(test_lookup2);
        toDOM(test_blarg);
        print(test6);
        toDOM(test_lookup);
        return get_test(test4, test6);
}

function int get_test(int test_request, int test_request2) {
        if(test_request + 2 == test_request2) {
                return test_request;
        } else {
                return 0;
        }
}
```

```
/* This function tests the toDOM function in standard library. */

table test_lookup = #{ int test_number, int test_begin_page, int test_end_page
# 1         , 389      , 431
# 2         ,          451, 493
# 8 , 823, 865 }#;

table test_blarg = #{ int test_number, int test_begin_page, int test_end_page
# 7, 761, 803}#;

table test_lookup2 = #{ int test_number, int test_begin_page, int test_end_page
        # 3 , 513, 555
        # 4 , 575, 617
        # 5 , 637, 679
        # 6 , 699, 741
        # 7 , 761, 803 }#;

table question_info =
#{ string prb_type          ,int prb_num        ,int pg_num         ,string diff
#                   "abs_value"             ,15                             ,418                            ,"medium"
#  "abs_value"              ,17                             ,456                            ,"medium"
#  "average"                        ,18                             ,835                            ,"hard"
#  "equations"              ,6                              ,454                            ,"easy"
}#;

int test4 = 4;
int test6 = 6;

function int main() {
        /*toDOM(test_lookup2);
        toDOM(question_info);*/
        /*print(test6);
        toDOM(test_lookup);*/
        /*test_lookup << test_lookup2;
        toDOM(test_lookup);*/
        table ext_question_info =
        #{ string prb_type, int prb_num, int pg_num, int test_num
        #  "abs_value"              ,15                             ,418                            , 0
        #  "abs_value"              ,17                             ,456                            , 0
        #  "average"                        ,18                             ,835                            , 0
        #  "equations"              ,6                              ,454                            , 0
        }#;
        toDOM(ext_question_info);
        ext_question_info | ext_question_row {
                test_lookup | test_lookup_row {
                        if( ext_question_row.pg_num > test_lookup_row.test_begin_page) {
                                if(ext_question_row.pg_num < test_lookup_row.test_end_page) {
                                        ext_question_row.test_num = test_lookup_row.test_number;
                                }
                        }
                }
        }
        print(test6);
        delete(ext_question_info);
        toDOM(ext_question_info);
        return 0;
}
```

```
/* This function tests the toDOM function in standard library. */

table test_lookup = #{ int test_number, int test_begin_page, int test_end_page
# 1         , 389      , 431
# 2         ,             451, 493
# 8 , 823, 865 }#;

table test_lookup2 = #{ int test_number, int test_begin_page, int test_end_page
            # 3 , 513, 555
            # 4 , 575, 617
            # 5 , 637, 679
            # 6 , 699, 741
            # 7 , 761, 803 }#;

table question_info =
#{ string prb_type            ,int prb_num       ,int pg_num         ,string diff
#                       "abs_value"                   ,15                               ,418                              ,"medium"
#  "abs_value"                  ,17                                ,456                              ,"medium"
#  "average"                                ,18                                ,835                              ,"hard"
#  "equations"                  ,6                                  ,454                              ,"easy"
}#;

function int main() {
            table xtd_question_info = #{ int prb_num, int pg_num
            # 0,        0
            }#;
            int test_num;
            xtd_question_info :: test_num;
            toDOM(xtd_question_info);
            print(test_num);
            delete(xtd_question_info);
            print(test4);
            /*toDOM(test_lookup);*/
            test_lookup2 :: question_info;
            print(test6);
            toDOM(test_lookup2);

            return 0;
}
```

```
/* This function tests the toDOM function in standard library. */

table test_lookup = #{ int test_number, int test_begin_page, int test_end_page
# 1        , 389     , 431
# 2        ,          451, 493
# 8 , 823, 865 }#;

table test_lookup2 = #{ int test_number, int test_begin_page, int test_end_page
        # 3 , 513, 555
        # 4 , 575, 617
        # 5 , 637, 679
        # 6 , 699, 741
        # 7 , 761, 803 }#;

table question_info =
#{ string prb_type          ,int prb_num       ,int pg_num        ,string diff
#                   "abs_value"                  ,15                         ,418                        ,"medium"
#  "abs_value"              ,17                          ,456                        ,"medium"
#  "average"                        ,18                          ,835                        ,"hard"
#  "equations"              ,6                           ,454                        ,"easy"
}#;

function int main() {
        table xtd_question_info = #{ int prb_num, int pg_num, int test_num
        }#;
        toDOM(question_info);
        toDOM(test_lookup);
        toDOM(xtd_question_info);
        int test_num;
        print(test_num);
        delete(xtd_question_info);
        table xtd_question_row = #{ int prb_num, int pg_num
        # 0,       0
        }#;
        xtd_question_row :: test_num;

        toDOM(xtd_question_row);
        print(test_num);
        delete(xtd_question_row);
        question_info | question_row {
                test_lookup | test_lookup_row {
                        if(question_row.pg_num >= test_lookup_row.test_begin_page) {
                                if(question_row.pg_num <= test_lookup_row.test_end_page) {
                                        xtd_question_row.test_num = test_lookup_row.test_number;
                                        xtd_question_row.prb_num = question_row.prb_num;
                                        xtd_question_row.pg_num = question_row.pg_num;
                                        xtd_question_info << xtd_question_row;
                                }
                        }
                }
        }
        toDOM(xtd_question_info);

        return 0;
}
```