Report Table Generation Language

Mohammad Haq

December 23, 2015

1 Introduction

There are many tools for importing a data set and analyzing it. A tool like Excel might be too heavyweight and clumsy for a repetitive task that imports a small data set and does a few data manipulations. On the other hand a general purpose programming language like Perl might be too complex with need for standard library calls to do the same task. Excel does offer a tabular graphical environment. Perl, or a similar general purpose programming language, would need library calls or a slew of print statements to generate a decent formatted output. The Report Table Generation Language offers a balance by providing an easy to use general purpose programming language with built-in features for importing, manipulating and printing tables.

The general purpose features found in mainstream languages are objects, functions, arrays, and basic types (eg doubles, integers, strings). The two special built-in features that this language offers are automatic parsing of CSV data into an array of objects, and the ability to transform data unnamed functions, and have the data printed to the screen.

For this language to be useful for an average program, many features would need to be added. For example, a library would be needed to import data from a non-CSV sources. Full support for methods would be useful. A feature for doing what-if analysis would also be a nice addition. Although basic printing facilities are offered, more robust string handling and printing libraries would make report generation a lot more effective.

This document provides an introduction to the language, a reference manual, which can be used to implement the language, and experiences developing the software.

2 Language Tutorial

2.1 Building the compiler

2.1.1 Requirements

The following tools will be required to build the compiler from source.

1. Ocaml tool set
2. Gcc - C compiler
3. Python - For running tests
4. Make - for building the software
5. Linux or Unix environment

2.1.2 Unpack the source

```
$tar -xzvf rtbl.tar
```

2.1.3 Build the compiler

```
$make
```

2.1.4 Run tests

Running tests ensures that the software built properly. Check to see that all the tests ran successfully.

```
$python test.py
```

2.2 Compiling a simple program

In your editor of choice, type the following program in the file hello.txt.

```
Print("Hello world");
```

Then compile your program. The compilation is a two-step process. First direct your source program into the start executable. The output will be a C program called output.c. The second step is compiling the C file to an executable of your choice.

```
$./start < hello.txt
$gcc -o hello output.c libcsv.o
$./hello
Hello world
```

This gives you a basic overview of how to write code and compile it using the table generation compiler.

2.3 Language Tutorial

2.3.1 Basics

A tbl program consists of object definitions, function definitions, table definitions, and statements to utilize the language features. Note that there is no main function. Code that lives outside of any of the aforementioned definitions get executed sequentially as a statement block. Functions work in the same way as in most procedural languages - there is a parameter list, and calls are made with the corresponding arguments. Variables declared inside of a function are considered local. Table definitions allow the utilization of automatic iteration over a dataset, data transformation using unnamed functions, and pretty-printing in tabular form. One basic feature in the language is the automatic parsing and deserializing of the data into Objects. This can be convenient for analyzing data sets.

2.3.2 Basic Field Types

The basic field types supported in the language are String, Int, Float, and Boolean. Custom product types can be defined using Objects.

2.3.3 Objects

Objects only support basic field types. Notice in the example below that each field is separated by a comma, the last field does not have a comma, and the definition is terminated by a semi-colon.

```
/* Cashflow Object */
Object Cashflow {
number -> Int,
startDate -> String,
endDate -> String,
payDate -> String,
principal -> Float,
rate -> Float,
yearFraction -> Float
};
```

To create an object, the following syntax has to be used. To instantiate the object, the constructor has to be called.

```
Obj Cashflow payment;
payment := Cashflow();
```

Objects members can be accessed using the dot notation

```
payment.payDate := "31Dec15";
payment.rate := 0.06;
payment.yearFraction := 1.0;
```

2.3.4 Functions

Here is an example of a function definition. Notice that the Func keyword has to be used, along with a return type. Since the body of the function is a statement block, to return a value from the function, a return statement has to be used.

```
Func Float sum( Float a, Float b )
{
return a + b;
}
```

Calling a function just requires passing in the arguments

```
sum(3.0,4.0);
```

2.3.5 Tables

To define a table, using the syntax below. The return type can only be a basic type. The code that follows the colon is an unnamed function with an implicit *item* parameter, which corresponds to the current item being iterated. Local variables, if any are defined, come first. Then statements, if any are needed, and the final block of code has to be an expression. Notice that a return is not required here.

```
Table Employee(Person) {
  name -> String : item.name,
  age -> Int : item.age,
  toRetirementAge -> Int : 65 - item.age
};
```

To generate a table, a table instance has to be created. An array or the csv input file (Input) has to be provided to the Table constructor.

```
Tab Price results := Price(Input);
```

To print out the table, the Table's print method has to be called

```
results.print();
```

2.3.6 Control Flow

The only looping mechanism provided in the language is a for loop.

```
Int count := 0;

for( i=0; i < 10; i := 1 + 1 )
{
count := count + 1
}
```

An if-then-else expression is provided in the language to branch execution. The predicate expression evaluates to a Boolean, and the 'then' and 'else' branches have an expression that must evaluate to the same type.

```
Int jimAge := 20;
Int janeAge := 30;
Int maxAge;

maxAge := if( jimAge < janeAge )
then { janeAge } else { jimAge };
```

2.3.7 Arrays

An array of consecutive items can be created. Array indexing starts from zero. For example,

```
Float periods[5] := {0.0,0.25,0.5,0.75,1.0};
Float first;

first := periods[0];
```

3 Language Reference Manual

3.1 Lexical Conventions

3.1.1 Tokens

For lexical analysis, the following token classes are defined: identifiers, keywords, constants, string literals, operators and separators. Comments are ignored as well as the remaining white space.

3.1.2 Comments

Comments are in the C-Style. The start and end of a comment are designated by the '/*' and '*/' tokens respectively. Comments do not nest and cannot be added inside a string. An example comment is:

```
/* This is a comment */
```

3.1.3 Identifiers

Identifiers are case-sensitive. The identifier may include numbers, letters, and underscores.

$identifier \rightarrow (letter|\_)(letter|digit|\_)*$

3.1.4 Keywords

The following are keywords used in the language, and may not be used as identifiers:

Table : Used for defining Table objects.

String : String data type

Int : Int data type

Boolean : Boolean data type

Float : Float data type

Func : For defining functions

if : Used in if-then-else control structure

then : Used in if-then-else control structure

else : Used in if-then-else control structure

for : Used in for loops

Input : Global Input table

true : Boolean true value

false : Boolean false value

item : current object in iteration

Obj : Obj type specifier for custom Object types

Tab : Tab type specifier for Table objects

Input : Reference to the input CSV array that is parsed from a csv file

CSV : To mark which class the input CSV file is supposed to be parsed into

3.1.5 Constants

Int Ints contain a sequence of digits and an optional minus sign (-) at the beginning to indicate a negative number. The integer can be an number between 2,147,483,648 and 2,147,483,647.

$Int \rightarrow (-)?(digit)+$

String A String literal is a sequence of characters surrounded by double quotes. For example,

$String \rightarrow'' (non-quote)*''$

```
String hello := "Hello World";
```

Float A floating point constant consists of a sequence of one of more digits followed by a decimal point. The digits after the decimal point make the fractional component of the floating point number. Notice that a decimal and fractional component are required.

$Float \rightarrow ('-')?(digit) +' .'(digit)+$

Boolean The only acceptable Boolean constant values are true and false.

$Boolean \rightarrow true|false$

3.2 Data Types

Besides the basic data types, custom product types, called Objects, can be defined.

3.2.1 Basic Data Types

The primitive data types are: Int, String, Float and Boolean. More details can be found in the Constants section.

3.2.2 Objects

These can be used like struct/records. After an Object definition is provided, a type is created, and instances of the Object can be created using the constructor which has the same name as the type. These instances have a fixed size of memory allocated to them. The value for each member can be set or accessed using a '.'.

```
obj_decl -> OBJECT ID { field_list };
field_list -> ID -> type_name | field_list , ID -> type_name
```

3.2.3 Tables

The definition for the table consists of series of fields. Each field has a name, a return type, and an unnamed function block. The function block can start with an optional list of variable declarations, followed by an optional list of statements, but must end with an expression. The implicit parameter to each of the function blocks is an instance of the Object passed into the Table, which can be accessed by the *item* object. The *item* object remains in scope while each of the member functions are evaluated. After a Table has been defined, a Table instance can be created by either passing in an array or the 'Input' keyword, which represents the array of objects parsed from the input CSV file. The Table object has a print method which prints the data to standard out.

```
table_decl -> Table ID ( ID ) { table_column_list };
table_column_list ->   table_column | table_column_list , table_column
table_column ->   ID -> type_name : column_func
column_func -> var_decl_list stmt_list expr
```

3.2.4 Arrays

An Array is a data structure that contains a fixed block of memory containing a fixed number of consecutive Objects.

Declaring Arrays An array can be declared by indicating the data type, the identifier, and the number of objects it can store. The number of elements must be greater than zero. For example,

```
array_decl -> ID [ literal ];
```

```
Person people[10];
```

Initializing Arrays An Array can be initialized by providing it a list of values separated by commas. For example,

```
Person joe := Person();
```

```
Person jane := Person();

Person people[2] := { joe, jane };
```

Accessing Array An Array element can be accessed by indexing into the array. The array index starts at 0. For example,

```
Person jane := people[1];
```

3.3 Meaning of Identifiers

The identifiers in the language can be variables, function names, table names, and members of objects. Identifiers also have scope.

3.3.1 Variables

A variable name acts as a pointer to a block of memory for an object.

3.3.2 Types

Integer Integers are 32-bit signed integer values

Strings Strings are objects with an array of ASCII characters.

Floats Floats are represented as double-precision floating point numbers.

Boolean The only values are *true* and *false*.

Objects New data types can be created by defining new Objects.

3.4 Conversions

There are no implicit conversions from one type to another type.

3.5 Expressions

3.5.1 Postfix Expressions

Array References An array reference consists of two components. The first part has type Array and the part inside the brackets has type integral. For example:

```
name[index];        /* Get */
```

Function Calls A function call consists of a name followed by list of arguments separated by commas.

```
func_call -> ID ( args_opt );
args_opt -> args_list | e
args_list -> args_list , expr
```

For example:

```
Int total := sum( 80, 100 );
```

Method Calls a method call consists of an Object, a period, a name, and then a comma separated list of arguments. Only the print() method is supported for Tables.

For example:

```
Tab Person people := Person(Input);
people.print();
```

Object Member References Consists of accessing a member from a table. The object name comes first, then a dot, and then an identifier representing the field.

3.5.2 Multiplicative Operators

The multiplicative operators are *, /, and %. These are left-associative.

3.5.3 Additive Operators

The additive operators are + and -. These are left-associative.

3.5.4 Relational Operators

The relational operators are <, >, <=, and >=. They are left-associative.

3.5.5 Equality Operators

Equality Operators are == and !=. They are left-associative.

3.5.6 Logical AND Operator

The && operator. This is left-associative.

3.5.7 Logical OR Operator

The || operator. This is left-associative.

3.5.8 Assignment Expression

The assignment operator is := (one colon and one equal sign). This is right-associative.

3.5.9 if-then-else Expression

First a predicate is evaluated to a boolean type in the if clause. If the value is true, then the 'then' clause is evaluated. If the predicate returns false, then the else expression is evaluated.

```
expr -> if( expr ) then {expr} else {expr}
```

3.6 Definitions

Definitions describe how functionality for the language constructs.

3.6.1 Object

An Object definition requires providing a type name, and defining each of the member fields, which consist of a name, a return type. The field definitions are separated by commas.

To create a Object do the following:

```
Object Person{
  name -> String,
  age -> Int
};
```

3.6.2 Table

Only Table definitions can have an unnamed functions for its members. The implicit return type for the function is the type of the member field. The function also has an implicit parameter called *item*, which is the current Object being examined. Since this function is a short-form function, the full function definition with a return type and formal parameter list is not required.

To create a Table do the following:

```
Table Employee(Person) {
  name -> String : item.name,
  age -> Int     : item.age,
  toRetirementAge -> Int : 65 - item.age
};
```

3.6.3 Functions

Functions must be declared before they are called. A function definition begins with the token 'Func', followed by the return type, the name, and then the comma separated parameter list. Each parameter must provide a type. Then a block of statements is defined. If the function is to return a value, the last statement must be the return statement.

```
Func Int minAge( Person a, Person b ) {
  Int age;

  age := if( a.age > b.age )
    then { a.age }
    else { b.age };
  return age;
}
```

3.6.4 Arrays

An array can be declared by indicating the data type, the name, and the number of objects it can store. If there is no initializer the array will be zeroed out. The number of elements must be greater than zero.

```
Person people[10];
```

3.7 Statements

3.7.1 Expression Statements

All expressions can be converted into a statements by adding a semicolon.

```
stmt -> expr ;
```

3.7.2 for

A for loop provides the programmer a controlled iterator, which consists of an initializer, a constraint, and a post iteration. The constraint has to evaluate to a Boolean. If the constraint is satisfied (i.e. is true), the expression inside the brackets is evaluated. For example,

```
for_loop -> for ( expr_opt ; expr_opt ; expr_opt ) { stmt_list }
```

```
Person people[10];
Int i;

for( i := 0; i < 10; i := i + 1 ) {
    people[i] := Person( "dummy", i );
}
```

3.8 Scope

All functions and Data Tables have global scope. Variables declared inside functions have local scope. Variables not in a function have global scope. All Types and constructors have global scope.

4 Project Plan

4.1 Process for planning, specification, development, testing

The plan for the project was to start with the proposal. Based on the feedback iterate on the ideas until a basic feature set was determined. Then build the language reference manual. Using the language reference manual do the first part of the development - the scanner and parser using ocamllex and ocamlyacc. This step required a back-and-forth with the modeling of the Abstract-Syntax-Tree datastructures. Once that was in a steady state, a python script was written for running automatic tests. Then the second part of the development was building the semantically checked abstract syntax tree, which was the semantic analysis part of the code. The final part was the code generation step. After a baseline compiler was setup, then it was an iterative process of testing individual pieces of code and making the corresponding fixes.

4.2 Programming Style Guide

Since this was a one-person project, the coding style was a bit lax. The C-code has a very strict guide. All identifiers are in camel case. The code for

building the table system lives in the file table.c and basic code lives in the file base.h Runtime function and object names start with underscores to keep them from clashing with user functions and objects. Single line comments are used only when non-obvious or non-intuititive functionality is written. Otherwise functionality should be decipherable from the code.

In Ocaml, function names have underscores separating the words. Full words are used over abbreviations whenever it is not too verbose. The type declarations for the abstract syntax tree live in the file ast.ml. The code for the semantically checked abstract syntax-tree live in the file sast.ml. The code generation code is in the file compile.ml. The main file executing the code lives in start.ml. Big chuncks of code related to common task (eg CSV parsing code generation) are separated using multi-line comment blocks. Otherwise single line comments are used.

Tokens in the lexer specification file (ie scanner.mll) are capitalized full-words. The non-terminals used in the grammar rules for the parser specification use underscores to separate words, and sometimes use single word abbreviations.

4.3 Project Timeline

- Proposal - 30Sep15
- Language Reference Manual - 4Nov15
- Scanner and Lexer - 25Nov15
- Semantic Analyzer - 17Dec15
- Code Generation - 21Dec15
- Final Testing - 22Dec15

4.4 Roles and Responsibilities

All tasks were done by me

4.5 Software Development Environment

- Operating System - Ubuntu Linux 14.04

- Editor - Emacs with Tuareg Ocaml Mode
- Scanner - Ocamllex
- Parser - OCamlyacc
- Compilers - Ocamlc and gcc
- Make - building executable
- Git - version control
- Python - automated testing
- Latex - Written deliverables

4.6 Project Log

Git log is attached in the appendix

5 Architectural Design

5.1 Compiler High Level

The compiler contains 4 stages outlined in the figure below (Figure 1). The first stage is the scanner. A scanner definition is created in the file scanner.mll, which contains regular expressions describing the token, and the corresponding token word. The ocamllex program takes a definition file and generates the scanner. The token words, acting as terminals, will be fed into the parser, which uses a file that contains the rules making up the grammar for the language, to generate the ocaml code that can be used to parse the tbl code. A parse of a piece of code results in an abstract-syntax-tree, which is described in the ast.ml file. The ast.ml file also contains functions to recursively print all the nodes in the tree. This can be useful for debugging and evaluating the parse performed on a piece of software. In the third stage, the parse tree is semantically checked. The code doing the semantic checking can be found in the file sast.ml. As the aforementioned tree is traversed a new tree is formed in each step; this new tree is the semantically checked abstract syntax tree. Although typically a semantically checked tree would be passed into the code generation module, in this case, the original

abstract-syntax-tree is passed. The additional information in the semantically checked tree would probably be better for the code generator, but the regular abstract-syntax tree was chosen for convenience. The code generator lives in the file compile.ml. This takes an AST and generates C-code for all the different nodes. It keeps an environment, which is passed around, for looking up helpful details.

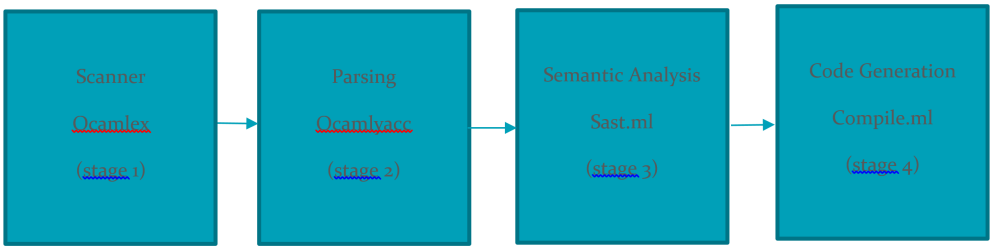


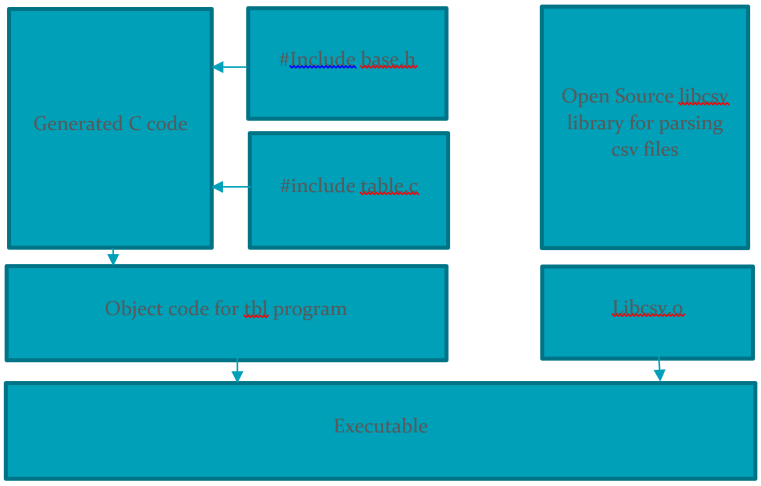
Figure 1

The interface between each of these components is outlined below

```
Scanner -> Scanner.mll - Scanner.token
Parsing -> Parser.ml  Parser.program
Semantic Analysis ->Sast.ml  check ( program : Ast.program)
Code Generation -> Compile.ml  compile ( program: Ast.program)
```

5.2 Runtime Environment

The code generator generates C-code, which is compiled using gcc to an executable that represents the tbl program. In order for the compiled C-code to function it requires some modules written in C. The first module is the open source CSV parsing library libcsv. Compiling this code using gcc generates the object code libcsv.o. Taking advantage of this C-code is the self-written code in base.h, which provides the glue to do the CSV parsing. The second module required is the table generation and printing code. This is found in the file table.c. Some other primitives like basic types, memory allocation for objects, and array allocation are found in table.c. The diagram below describes the interaction.



5.3 C Code Runtime

The generated C code relies heavily on the runtime system as described above. Here is a discussion about various pieces that work together. We mentioned, above, table.c contains some of the functions for generating the table data structure, iterating over the input array, and finally printing the results. Here are the highlights from this module

```
__Table_ -> Data structure that contains the meta data about the table.
__TableColumn_ -> Contains information about each column
__TableReturn_ -> Return object for each columns printable type
ObjectNew() -> Allocate memory for an Object
__TableNew -> Create a new table instance
_TablePrint -> Print the table
```

Here are the highlighted structs and functions found in base.h

```
OBJECT_HEADER -> Macro for common object field types
LoadCSVFile() -> Load a CSV file into an array
__calls -> Tracker for number of objects
* dataArray -> Global array for parsed objects
```

6 Testing Plan

6.1 Sample Programs

6.1.1 Cashflow Calculator

Here is the TBL code for a program that reads in a sample CSV file, transforms the data, and prints out the new values. First is the input file:

```
$cat csv.test
0,1sep15,30sep15,30sep15,1000000,0.02,0.083
1,1oct15,31oct15,30oct15,1000000,0.02,0.083
2,1nov15,30nov15,31nov15,1000000,0.02,0.083
3,1dec15,31dec15,31dec15,1000000,0.02,0.083
```

Here is the code file:

```
/* Define CSV Input Class */
CSV(Cashflow);

/* Financial Example */
Float rateMultiplier := 2.0;

/* Cashflow Object */
Object Cashflow {
number -> Int,
startDate -> String,
endDate -> String,
payDate -> String,
principal -> Float,
rate -> Float,
yearFraction -> Float
};

/* Discount Factors used to price */
Float discountFactors[4] := { 0.99, 0.98, 0.97, 0.96 };
```

```
/* Data Table Definition */
/* Note: Cashflow is the type of the object to be iterated */
Table Price(Cashflow) {
number -> Int : item.number,
price -> Float :
Float df;

/* Discount Factor */
df := discountFactors[item.number];

/* Price of a cashflow is rateMultiplier * principal * rate * yearFraction */
rateMultiplier * item.principal * item.rate * item.yearFraction * df };

/* Build Table */
Tab Price results := Price(Input);

/* Print Table */
results.print();
```

Here is the C code for this cashflow calculator

```
#include "base.h"
typedef enum __ProgramType_ {
__ProgramType_Int = 1,
__ProgramType_Float = 2,
__ProgramType_String = 3,
__ProgramType_Boolean = 4,
__ProgramType_Cashflow = 5} __ProgramType_;

typedef struct Cashflow {
 OBJECT_HEADER
int number;
char * startDate;
char * endDate;
char * payDate;
```

```
double principal;
double rate;
double yearFraction;
} Cashflow;


typedef struct __ProgramTypeInfo_ {
  char * name;
  __ProgramType_ type;
  int size;
} __ProgramTypeInfo_;

__ProgramTypeInfo_ __programTypeList_[] = {
 { "None", 0, 0 },
 { "Int" , __ProgramType_Int, sizeof(Int) },
 { "Float" , __ProgramType_Float, sizeof(Float) },
 { "String" , __ProgramType_String, sizeof(String) },
 { "Boolean" , __ProgramType_Boolean, sizeof(Boolean) },
 { "Cashflow" , __ProgramType_Cashflow, sizeof(Cashflow) }
};

#include "table.c"

__Table_ *  results ;
double discountFactors[4] = {0.99,0.98,0.97,0.96};
double rateMultiplier=2.;

void * _Price_getItem_( void * data, int index )
{
Cashflow* d;
Cashflow* item;

d = (Cashflow*) data;
item = &d[index];
return item;
}
__TableReturn_ __Price__number( void * input )
{
```

```
__TableReturn_ __ret_;
Cashflow * item = (Cashflow *) input;
__ret_._int = item->number;
return __ret_;;
}
 __TableReturn_ __Price__price( void * input )
{
__TableReturn_ __ret_;
Cashflow * item = (Cashflow *) input;
double df;(df=discountFactors[item->number]);
__ret_._float =
((((rateMultiplier)*(item->principal))*(item->rate))*(item->yearFraction))*(df);
return __ret_;;
}
TABLE_START( Price)
TABLE_COLUMN( "number", __ProgramType_Int, __Price__number )
TABLE_COLUMN( "price", __ProgramType_Float, __Price__price )
TABLE_END( Price)

void __Cashflow_cb1( void * s, size_t i, void * p )
{
Cashflow * arr;
Cashflow * item;

arr = (Cashflow *) p;
item = &arr[ __calls ];
if( item->fields == 0 )
  item->number = atoi( (char *) s );
else if( item->fields == 1 )
 item->startDate = newString( (char *) s );
else if( item->fields == 2 )
 item->endDate = newString( (char *) s );
else if( item->fields == 3 )
 item->payDate = newString( (char *) s );
else if( item->fields == 4 )
 item->principal = atof( (char *) s );
else if( item->fields == 5 )
 item->rate = atof( (char *) s );
```

```
else if( item->fields == 6 )
 item->yearFraction = atof( (char *) s );
item->fields++;
}

int main( int argc, char ** argv ) {
char * fileName = NULL;
dataArray = ArrayNew( __ProgramType_Cashflow, 1000 );
if( argc > 2 ) {
  if( strcmp( argv[1], "-input" ) == 0)
   fileName = argv[2];
 }
LoadCSVFile( fileName, __Cashflow_cb1,  cb2, dataArray );
results = __TableNew( "Price", TABLE_GETITEM( Price ),
    TABLE_COLUMNS( Price), 2, __ProgramType_Cashflow, __calls, dataArray );
;
__TablePrint( results);
}
```

6.1.2 GCD

Here is the TBL code for GCD

```
Func Int gcd( Int a, Int b )
{
  Int c;

  for( ; a != 0; )
  {
     c := a;
     a := b%a;
     b := c;
  }

  return b;
}
```

```
Int val;

val := gcd( 100, 30 );

Print(val);
```

Here is the generated C Program

```
#include "base.h"
typedef enum __ProgramType_ {
__ProgramType_Int = 1,
__ProgramType_Float = 2,
__ProgramType_String = 3,
__ProgramType_Boolean = 4} __ProgramType_;


typedef struct __ProgramTypeInfo_ {
  char * name;
  __ProgramType_ type;
  int size;
} __ProgramTypeInfo_;

__ProgramTypeInfo_ __programTypeList_[] = {
 { "None", 0, 0 },
 { "Int" , __ProgramType_Int, sizeof(Int) },
 { "Float" , __ProgramType_Float, sizeof(Float) },
 { "String" , __ProgramType_String, sizeof(String) },
 { "Boolean" , __ProgramType_Boolean, sizeof(Boolean) }
};

#include "table.c"

int val;


static int gcd(int a,int b ) {
```

```
int c;for( ; (a)!=(0) ; ){ (c=a); (a=(b)%(a)); (b=c);};
return b;}

int main( int argc, char ** argv ) {
char * fileName = NULL;
;
(val=gcd(100,30));
printf("%d", val );
}
```

6.2 Test Suites

A poor decision I made was not implementing print functionality for all types at the beginning. As a result, at the beginning instead of using automated tests, I was testing individual files. The bulk of these files didn't have output. I was mostly manually checking the parsing and the code generation. As I developed better printing facilities for my language, I started generating output. I was then able to build a regression test suite where I can check to see if my changes broke anything. Since I spent so much time on code generation I couldn't invest the time for a strong test suite. My test suite only has about 40 tests, and a lot of these tests don't show a lot. Some of them don't generate output, and some of them are failure cases. While I'm sure there are many bugs in my compiler, largely due to not having full coverage in my test suite, I was able to manually test the bulk of the functionality I was planning on testing.

6.3 Test-Cases

The test cases that I chose tested mainly two things. The first was that the code was properly being parsed. Second was that the code generation for each of the components was correct. A bulk of the test cases have some fragments of code, which allows for easy inspection of the parsing and the code generation. There are a few test cases that have bigger programs that utilize many of the language features. Correct output from the bigger code pieces suggest many of the key features are functioning properly.

6.4 Automation

I wrote a python script test.py that reads in the files from the test directory, runs the compiler on it to generate the c-code, compiles the C-code to an executable, and then prints its output to the test directory, where it can be compared to the preset output file. The automation works successfully. Unfortunately the test cases don't provide full coverage.

7 Lessons Learned

This was a complicated project that required knowledge of several different technologies and compilers concepts. Scanning and Parsing ended up being an iterative process as figuring out the correct grammar and eliminating shift-reduce errors was challenging. The code generation requires a lot of careful thought, and an iterative process works less efficiently. Testing is crucial. Automated test-suites are required, and careful testing plan is required. With so much complexity and preparation required, the biggest lesson is starting early, developing a plan, and working regularly on the project instead of trying to accomplish any of the aforementioned tasks in big bursts.

Here is a list of lessons for future teams

- Start early - Common sense advice for almost everything in life, especially programming assignments
- Spread out the work - Big bursts don't work for such a complicated project
- The ratio of time required for scanning : parsing : semantic analysis : code generation : testing is probably 1 : 1 : 2 : 8 : 4
- Having to change your grammar at the last minute is painful. Having the parser produce a complete parse-tree before the other steps are started is very helpful
- Code generation is very complicated. From the examples it seems easy, but as you add more features, the complexity increases non-linearly

8 Appendix

8.1 Source Code

ast.ml

```
type op = Add | Sub | Mult | Div | Equal | Neq | Less | Leq | Greater | Geq | Mod

type stmt =
    Block of stmt list
  | Expr of expr
  | For of expr * expr * expr * stmt list
  | Foreach of string * stmt list
  | Return of expr

and expr =
    Literal of int
  | LitFloat of float
  | LitBool of bool
  | LitString of string
  | Id of string
  | Binop of expr * op * expr
  | Assign of expr * expr
  | Call of string * expr list
  | IfThenElse of expr * expr * expr * stmt list * stmt list
  | ArrayAccess of string * expr
  | MemberAccess of string * string
  | MethodCall of string * string * expr list
  | CSVInput
(*  | If of expr * expr * expr *)
  | Noexpr

type progType =
    Int
  | Float
  | String
  | Boolean
  | Record of string
```

```
  | Table of string

type varDecl = {
    vname : string;
    varType : progType;
    varArraySize: int option;
    varAssign: expr list option;
}

type funcDecl = {
    name : string;
    formals : varDecl list;
    locals : varDecl list;    (* Needs Initializer *)
    body : stmt list;
    retType : progType;
}

type tableColumnFunc = {
    locals: varDecl list;
    statements: stmt list;
    lastExpr: expr;
  }

type tableColumn = {
    tblColName : string;
    tblColType: progType;
    tblColFunc: tableColumnFunc;
}

type tableDecl = {
    tableName : string;
    iterClass : string;
    columns: tableColumn list;
}

type objDecl = {
     objName : string;
     objFields : (string * progType) list;
```

```
}

type csvDecl = {
    csvInputClass: string;
}

type program = {
    varDecls : varDecl list;
    funcDecls : funcDecl list;
    objDecls : objDecl list;
    tableDecls : tableDecl list;
    stmts : stmt list;
    csvDecls: csvDecl list;
  }


let rec string_of_expr = function
    Literal(l) -> string_of_int l
  | LitFloat(l) -> string_of_float l
  | LitBool(l) -> string_of_bool l
  | LitString(l) -> l
  | Id(s) -> s
  | Binop(e1,op,e2) -> string_of_binop (e1,op,e2)
  | Assign(var,exp) -> string_of_expr var ^ "init " ^ string_of_expr exp ^ "\n"
  | Call(name,args) -> name ^ "(" ^ String.concat ","
    (List.map string_of_expr args) ^ ")\n"
  | IfThenElse(pred,e2,e3,sl1,sl2) -> "if( " ^ string_of_expr pred ^ ") then( "
    ^ string_of_expr e2 ^ " ) else ( " ^ string_of_expr e3 ^ " )\n"
  | ArrayAccess(name,e) -> name ^ "[" ^ string_of_expr e ^ "]"
  | MemberAccess(name,field) -> "name: " ^ name ^ " field " ^ field
  | MethodCall(name,field,args) -> "name: " ^ name ^ " field " ^ field ^ "(" ^
    String.concat "," (List.map string_of_expr args) ^ ")\n"
  | CSVInput -> "CSVInput"
  | Noexpr -> ""

and string_of_binop (e1,op,e2) =
  let string_e2 = " " ^ string_of_expr e2 ^ "\n" in
  string_of_expr e1 ^ " " ^
```

```
  match op with
    Add -> "+" ^ string_e2
  | Sub -> "-" ^ string_e2
  | Mult -> "*" ^ string_e2
  | Mod -> "%" ^ string_e2
  | Div -> "/" ^ string_e2
  | Equal -> "==" ^ string_e2
  | Neq -> "!=" ^ string_e2
  | Less -> "<" ^ string_e2
  | Leq -> "<=" ^ string_e2
  | Greater -> ">" ^ string_e2
  | Geq -> ">=" ^string_e2

let rec string_of_stmt = function
    Block(stmts) ->
      "{\n" ^ String.concat " " (List.map string_of_stmt stmts) ^ "}\n"
  | Expr(e) -> string_of_expr e ^ ";\n";
  | For(e1,e2,e3,stmts) -> "For (" ^ string_of_expr e1 ^ "; " ^ string_of_expr e2
    ^ string_of_expr e3 ^ ")" ^ String.concat " " (List.map string_of_stmt stmts)
  | Foreach(iden,stmts) -> "Foreach " ^ iden ^ " { " ^ String.concat " " (List.map
  | Return(e) -> "return " ^ string_of_expr e ^ "; \n"

let string_of_progType = function
    Int -> "Int"
  | String -> "String"
  | Float -> "Float"
  | Boolean -> "Boolean"
  | Record(name) -> name
  | Table(name) -> name

let string_of_varDecl dec =
  let nameType = "vname " ^ dec.vname ^ " type "
  ^ string_of_progType dec.varType in
  let assignment =
    match dec.varArraySize with
      Some(size) -> " Array size " ^ string_of_int size
    | None -> "" in
  let init_assign =
```

```
    match dec.varAssign with
      Some(li) -> String.concat "," (List.map string_of_expr li )
    | None -> " "  in
  nameType ^ assignment ^ init_assign


let string_of_funcDecl fdecl =
  fdecl.name ^ "(" ^ String.concat ", " (List.map string_of_varDecl fdecl.formals)
  ^ ")\n{\n" ^
  String.concat "" (List.map string_of_varDecl fdecl.locals) ^
  String.concat "" (List.map string_of_stmt fdecl.body) ^
  "}\n"

let string_of_objectDecl o =
  let string_of_fieldName (_name,_type) = _name ^ string_of_progType _type in
  "Object: " ^ o.objName ^ "\n" ^
  String.concat "\n" (List.map string_of_fieldName o.objFields) ^ "\n"

let string_of_tableDecl tbl =

  let string_of_columnFunc fn = String.concat "\n" (List.map string_of_varDecl fn.
    ^ "\n" ^ String.concat "\n" (List.map string_of_stmt fn.statements) ^
    "\n" ^ string_of_expr fn.lastExpr in

  let string_of_column col = col.tblColName ^ " -> " ^ string_of_progType col.tblC

  "Table " ^ tbl.tableName ^ "--Object-> " ^ tbl.iterClass ^ " -> " ^
  String.concat "\n" (List.map string_of_column tbl.columns)


let string_of_program p =
  String.concat "" (List.map string_of_varDecl p.varDecls) ^ "\n" ^
  String.concat "\n" (List.map string_of_funcDecl p.funcDecls) ^ "\n" ^
  String.concat " " (List.map string_of_objectDecl p.objDecls) ^ "\n" ^
  String.concat " " (List.map string_of_tableDecl p.tableDecls) ^ "\n" ^
  String.concat ";" (List.map string_of_stmt p.stmts) ^ "\n"
```

compile.ml

```
open Ast
module StringMap = Map.Make(String)

(* Function Type *)
type functionType =
  Constructor of string
| TableConstructor of string * int (* tableName, columnSize *)
| FuncCall

(* Var Info *)
type varInfo =
   RegVar
 | ArrayVar

(* Translation Environment *)
type environment = {
  function_index : functionType StringMap.t;
  var_index : (progType * varInfo * int) StringMap.t;
  method_index : (string * string) list;
  type_list: (string * int) list;
  csv_class: string option;
  }

(* Handle Binary Operation *)
let string_op = function
    Add -> "+"
  | Sub -> "-"
  | Mult -> "*"
  | Div -> "/"
  | Mod -> "%"
  | Equal -> "=="
  | Neq -> "!="
  | Less -> "<"
  | Leq -> "<="
  | Greater -> ">"
```

```
  | Geq -> ">="

let rec get_type env = function
    Literal(v) -> (Literal(v), Int )
  | LitFloat(v) -> (LitFloat(v), Float )
  | LitBool(v) -> (LitBool(v), Boolean )
  | LitString(v) -> (LitString(v), String )
  | Id(name) -> (
    let (t,v,num) =
     (try
          StringMap.find name env.var_index
       with Not_found -> raise( Failure "Variable Not Found")) in
    (Id(name),t))
  | Binop(e1,op,e2) -> get_type env e1
  | Assign(eVar,e) -> (
     let (_,t) = get_type env e in
     Assign(eVar,e),t)
  | Call( name, expr_li ) -> (* TODO Check function table for type *)
      Call(name,expr_li), Int
  | IfThenElse( e1, e2, e3, sl1, sl2 ) ->
     (* Get Types *)
     let (_,t1) = get_type env e1 in
     IfThenElse( e1, e2, e3, sl1, sl2), t1
   | MemberAccess(name,field) -> MemberAccess(name,field),Int (* Lookup type *)
   | ArrayAccess(name,e) -> ArrayAccess(name,e),Int (* Lookup type *)
   | MethodCall(name,field,args) -> MethodCall(name,field,args),Int (* Lookup type
   | CSVInput -> CSVInput,Int (* No Type here *)
   | Noexpr -> raise( Failure "No expression here")

(* Handle Expressions *)
let rec expr env = function
    Literal(l) -> string_of_int l
  | LitFloat(l) -> string_of_float l
  | LitBool(l) -> string_of_bool l
  | LitString(l) -> l
  | Id(s) -> s
  | Binop(e1,op,e2) -> "(" ^ expr env e1 ^ ")" ^ string_op op ^ "(" ^ expr env e2
  | Assign(var,exp) -> "(" ^ expr env var ^ "=" ^ expr env exp ^ ")"
```

```
  | Call( name, args ) -> if name = "Print" then (
         match args with
           [] -> "printf(\"\")"
         | hd::tl -> (
             let (_,t) = get_type env hd in
             match t with
               Int -> "printf(\"%d\", " ^ expr env hd ^ " )"
             | Float -> "printf(\"%f\", " ^ expr env hd ^ " )"
             | Boolean -> "printf(\"%s\", (" ^ expr env hd ^ "==true ? \"true\" :
             | String -> "printf(\"%s\", " ^ expr env hd ^ " )"
             | _ -> "printf( \"Cannot print this type\"\\n)"))
         else (name ^ "(" ^ (String.concat "," (List.map (expr env) args)) ^ ")")
  | MemberAccess(name,field) -> name ^ "->" ^ field
  | MethodCall(name,field,args) -> if field = "print"
                                   then "__TablePrint( " ^ name ^ ")"
                                   else "Unimplemented()"
  | CSVInput -> ""
  | ArrayAccess(name,e) -> (
    let (t,varT,num) =
    (try
        StringMap.find name env.var_index
     with Not_found -> (Int,RegVar,100)) in
    match t with
      Record(oName) -> "&" ^ name ^ "[" ^ expr env e ^ "]"
    | _ -> name ^ "[" ^ expr env e ^ "]" )
  | IfThenElse(pred,e2,e3,sl1,sl2) -> "(" ^ expr env pred ^ ") ? (" ^ expr env e2
  | Noexpr -> ""

(* Handle list of statements *)
let rec stmt env = function
    Block sl -> String.concat " " (List.map (stmt env) sl)
  | Expr e -> expr env e ^ ";"
  | For (e1,e2,e3,sl) -> "for(" ^ expr env e1 ^ " ; " ^ expr env e2 ^ " ; " ^ expr
    "{ " ^ (String.concat " " (List.map (stmt env) sl)) ^ "}"
  | Foreach(s1,sl) -> "Unimplemented"
  | Return(e) -> "return " ^ expr env e ^ ";"

let translate_c_type t =
```

```
  match t with
    Ast.Int -> "int"
  | Ast.String -> "char *"
  | Ast.Boolean -> "Boolean"
  | Ast.Float -> "double"
  | Ast.Record(name) -> name ^ "* "
  | Ast.Table(name) -> "__Table_ * "

let translate_c_type_array t =
  match t with
    Ast.Int -> "int"
  | Ast.String -> "char *"
  | Ast.Boolean -> "Boolean"
  | Ast.Float -> "double"
  | Ast.Record(name) -> name
  | Ast.Table(name) -> "__Table_ * "

let translate_c_type_id = function
    Ast.Int -> "__ProgramType_Int"
  | Ast.Float -> "__ProgramType_Float"
  | Ast.Boolean-> "__ProgramType_Boolean"
  | Ast.String -> "__ProgramType_String"
  | Ast.Record(str) -> "__ProgramType_" ^ str
  | Ast.Table(str) -> "Unimplemented"

let translate_c_type_id_for name = "__ProgramType_" ^ name;;

let translate_object_constructor oName =
  "(" ^ oName ^ " *) ObjectNew( __ProgramType_" ^ oName ^ " )"

let translate_table_constructor name tblColSize dataClass dataArrName dataArrSize
  "__TableNew( \"" ^ name ^ "\", TABLE_GETITEM( " ^ name ^ " ),
    TABLE_COLUMNS( " ^ name ^ "), " ^ string_of_int tblColSize ^
    ", " ^ translate_c_type_id_for dataClass ^ ", " ^ dataArrSize ^ ", " ^
    dataArrName ^ " )"

let translate_table_constructor_single name tName colSize =
  "__TableNew( \"" ^ name ^ "\", TABLE_GETITEM( " ^ name ^ " ),
```

```
    TABLE_COLUMNS( " ^ name ^ " ), " ^ string_of_int colSize ^
    ", 0 , 0, NULL )"

let translate_varDecl env varDecl =
  let csvClass = function
    Some(nm) -> nm
    |  _ -> " " in
  let name = translate_c_type varDecl.varType ^ " " ^ varDecl.vname in
  let name2 = translate_c_type_array varDecl.varType ^ " " ^ varDecl.vname in
  let funcMap = env.function_index in
  let nonArrayAssign env = function
      Some(hd::tl) -> (match hd with
         Call( name, headExpr::tailExpr ) -> (let callType =
           try StringMap.find name funcMap
           with Not_found -> FuncCall in
           match callType with
             Constructor(oName) -> translate_object_constructor name
           | TableConstructor(tName,colLen) -> (
              match headExpr with
                CSVInput -> translate_table_constructor
                  name colLen (csvClass env.csv_class) "dataArray" "__calls"
              | Id(str) -> translate_table_constructor
                  name colLen "Int" str "10" (* TODO GET ARRAY ITEM COUNT *)
              | _ -> raise( Failure "Bad Table Constructor Call" ))
           | FuncCall -> expr env hd)
         | _ -> "=" ^ expr env hd )
    | _ -> "" in
  let assignable =
    match varDecl.varArraySize with
      None -> name ^ if varDecl.varAssign = None then ""
        else nonArrayAssign env varDecl.varAssign
    | Some(size) -> name2 ^
       match varDecl.varAssign with
         None -> ""
       | Some(li) -> "[" ^ string_of_int size ^ "] = {" ^
         String.concat "," (List.map (expr env) li) ^ "}"
  in
  assignable ^ ";"
```

```
let translate_funcDecl env funcDecl =

  (* Handle formals *)
  let string_of_formal formal =
    translate_c_type formal.varType ^ " " ^ formal.vname
  in

  (* Print Function *)
  "static " ^ translate_c_type funcDecl.retType ^ " " ^
    funcDecl.name ^ "(" ^ (String.concat ","
    (List.map string_of_formal funcDecl.formals)) ^ " ) {\n"
    ^ (String.concat ";\n" (List.map (translate_varDecl env) funcDecl.locals))
    ^ (String.concat ";\n" (List.map (stmt env) funcDecl.body)) ^ "}\n"
  (* Function End *)

let translate_tableDecl env tableDecl =

  (* C Table Function Return Type *)
  let c_table_ret_type = "__TableReturn_ " in

  (* C Column Function Name *)
  let c_column_decl_name name = "__" ^ tableDecl.tableName ^
    "__" ^ name in

  (* C Column Function Decl *)
  let c_method_decl name = c_table_ret_type ^ c_column_decl_name name ^
    "( void * input )\n{\n" in

  (* C Get Item Function *)
  let c_getItem_decl = "void * _" ^ tableDecl.tableName ^ "_getItem_" ^
    "( void * data, int index )\n{\n" ^
    tableDecl.iterClass ^ "* d;\n" ^
    tableDecl.iterClass ^ "* item;\n\n" ^
    "d = (" ^ tableDecl.iterClass ^ "*) data;\n" ^
    "item = &d[index];\n" ^
    "return item;\n}\n" in
```

```
(* Column Function translate *)
let translate_table_column_func env col retType =
  let retVal = (expr env) col.lastExpr in
  let func_ret_code =
    match retType with
      Int -> "__ret_._int = " ^ retVal ^ ";\n"
    | Float -> "__ret_._float = " ^ retVal ^ ";\n"
    | String -> "__ret_._str = " ^ retVal ^ ";\n"
    | Boolean -> "__ret_._boolean = " ^ retVal ^ ";\n"
    | Table(s) -> "UNIMPLEMENTED"
    | Record(s) -> "UNIMPLEMENTED" in
  (String.concat ";\n" (List.map (translate_varDecl env) col.locals))
  ^ (String.concat ";\n" (List.map (stmt env) col.statements)) ^
  func_ret_code ^ "return __ret_;" ^ ";\n}\n" in

(* C Table Function Preface *)
let c_table_func_preface = c_table_ret_type ^ "__ret_;\n" ^
  tableDecl.iterClass ^ " * item = (" ^ tableDecl.iterClass ^ " *) input;\n" in

(* Translate Column *)
let translate_column env col = c_method_decl col.tblColName ^
  c_table_func_preface ^
  translate_table_column_func env col.tblColFunc col.tblColType in

(* C print Column Functions *)
let c_column_functions = String.concat " "
  (List.map (translate_column env) tableDecl.columns) in

(* C Column item for list *)
let c_column_Item col = "TABLE_COLUMN( \"" ^ col.tblColName ^ "\", " ^
  translate_c_type_id col.tblColType ^ ", " ^ c_column_decl_name col.tblColName
  " )" in

(* Print Column Info *)
let c_column_list = "TABLE_START( " ^ tableDecl.tableName ^ ")\n" ^
  String.concat "\n" (List.map c_column_Item tableDecl.columns) ^ "\n" ^
  "TABLE_END( " ^ tableDecl.tableName ^ ")\n" in
```

```
  (* Print *)
  c_getItem_decl ^ c_column_functions ^ c_column_list

let translate_objectDecl env obj =
  let c_prelude = "typedef struct " ^ obj.objName ^ " {\n OBJECT_HEADER \n" in
  let field_string (name,t) = translate_c_type t ^ " " ^ name in
  let fields = String.concat ";\n" (List.map field_string obj.objFields) in
  let c_end = ";\n} " ^ obj.objName ^ ";\n" in
  c_prelude ^ fields ^ c_end

let translate_main env varDecl_list stmt_list =
  let start = "int main( int argc, char ** argv ) {\n" in
  let filter_only_assignables li =
    List.filter (fun varDecl ->
      if varDecl.varArraySize != None || varDecl.varAssign != None
      then true else false) li in
  let varDecl_string = String.concat "\n" (List.map (translate_varDecl env)
    (filter_only_assignables varDecl_list)) in
  let stmts_string = String.concat "\n" (List.map (stmt env) stmt_list ) in
  start ^ varDecl_string ^ stmts_string ^ "\n}"

let load_constructors p _map =
  let _map2 = List.fold_left
    (fun m objD -> StringMap.add objD.objName (Constructor(objD.objName)) m ) _map
  List.fold_left
   (fun m tableD -> StringMap.add tableD.tableName (TableConstructor(tableD.tableN
    _map2 p.tableDecls

let filter_unassigned varDecls_li =
  List.filter (fun varDecl -> if varDecl.varArraySize = None && varDecl.varAssign
  then true else false ) varDecls_li

(* List of counters *)
let rec inc_list = function
    0 -> []
  | num -> num :: inc_list (num-1)

(* Zip two lists *)
```

```
let rec zip l1 l2 =
  match l1,l2 with
  | [],_ -> []
  | _, []-> []
  | (x::xs),(y::ys) -> (x,y) :: (zip xs ys)

let load_type_list objDecls =
  let basic_types = [ ("Boolean",4); ("String",3); ("Float",2); ("Int",1) ] in
  let obj_len = List.length objDecls in
  let nums = inc_list (obj_len + 4) in
  let new_types = List.map (fun obj -> obj.objName) objDecls in
  let items = zip new_types nums in
  items @ basic_types

let print_program_typeInfo = "
typedef struct __ProgramTypeInfo_ {
  char * name;
  __ProgramType_ type;
  int size;
} __ProgramTypeInfo_;\n\n"

let translate_types_enum t_list =
  let preface = "typedef enum __ProgramType_ {\n" in
  let translate_type (t,num) = "__ProgramType_" ^ t ^ " = " ^
    string_of_int num in
  let epilogue = "} __ProgramType_;\n\n" in
  preface ^ String.concat ",\n" (List.map translate_type t_list) ^ epilogue

let translate_type_table t_list =
  let preface = "__ProgramTypeInfo_ __programTypeList_[] = { \n" in
  let first_item = " { \"None\", 0, 0 },\n" in
  let translate_type_item (t,num) = " { \"" ^ t ^ "\" , " ^ "__ProgramType_" ^
    t ^ ", sizeof(" ^ t ^ ") }" in
  let epilogue = "\n};\n\n" in
  preface ^ first_item ^ String.concat ",\n" (List.map translate_type_item t_list)
  epilogue

let translate_varDecl2_globals env varDecl_list =
```

```
  let name varDecl = translate_c_type varDecl.varType ^ " " ^ varDecl.vname in
  let name2 varDecl = translate_c_type_array varDecl.varType ^ " " ^ varDecl.vname
(*  let assignable = List.filter (fun varDecl -> varDecl.varArraySize != None ||
    varDecl.varAssign != None) varDecl_list in*)
  let nonArrayAssign env = function
      Some(hd::tl) -> (match hd with
         Call( name, args ) -> " " (*(let callType =
           try StringMap.find name funcMap
           with Not_found -> FuncCall in
           match callType with
             Constructor -> translate_object_constructor name
           | TableConstructor -> translate_table_constructor_single name
           | FuncCall -> expr hd)*)
        | _ -> "=" ^ expr env hd )
    | _ -> "" in
  let translate2 env varDecl =
    match varDecl.varArraySize with
      None -> name varDecl ^ (if varDecl.varAssign = None then ""
        else nonArrayAssign env varDecl.varAssign)
    | Some(size) -> name2 varDecl ^
       match varDecl.varAssign with
         None -> "[" ^ string_of_int size ^ "]"
       | Some(li) -> "[" ^ string_of_int size ^ "] = {" ^
         String.concat "," (List.map (expr env) li) ^ "}"
  in String.concat ";\n" (List.map (translate2 env) varDecl_list) ^ ";\n"


let translate_varDecl2_assignments env varDecl_list =
  let csvClass = function
    Some(nm) -> nm
    |  _ -> " " in
  let assignable = List.filter (fun varDecl -> varDecl.varArraySize == None &&
    varDecl.varAssign != None) varDecl_list in
  let funcMap = env.function_index in
  let nonArrayAssign env vname = function
      Some(hd::tl) -> (match hd with
         Call( name, args ) -> (let callType =
           try StringMap.find name funcMap
```

```
            with Not_found -> FuncCall in
            match callType with
              Constructor(oName) -> vname ^ " = " ^ translate_object_constructor na
            | TableConstructor(tName,colSize) ->
                (match args with
                   [] -> " "
                 | car::cadr ->
                    (match car with
                      CSVInput ->  vname ^ " = " ^ translate_table_constructor name
                     | Id(nm) -> (
                       let (t,varT,num) =
                       (try
                           StringMap.find nm env.var_index
                        with Not_found -> (Int,RegVar,100)) in
                       let cls = function
                         Record(n) -> n
                         | _ -> raise (Failure "Bad Array Access" ) in
                         vname ^ " = " ^ translate_table_constructor name colSize (
                     | _ -> ""))
            | FuncCall -> "")
          | _ -> "" )
      | _ -> "" in
    let translate2 env varDecl =
      match varDecl.varArraySize with
        None -> (if varDecl.varAssign = None then ""
          else nonArrayAssign env varDecl.vname varDecl.varAssign)
      | _ -> " "
    in String.concat ";\n" (List.map (translate2 env) assignable) ^ ";\n"

(*****************************************************************************)
(* L O A D   I N F O                                                         *)
(*****************************************************************************)

let load_var_decls varDecls _map =
  let vtype = function
      None -> RegVar
    | _ -> ArrayVar in
  let vNum = function
```

```
      Some(n) -> n
    | _ -> 0 in
  List.fold_left (fun m v -> StringMap.add v.vname (v.varType, vtype v.varArraySiz

(************************************************************************)
(* C S V    P A R S I N G                                               *)
(************************************************************************)

let rec zero_counter = function
    0 -> []
  | num -> (num-1)::(zero_counter (num-1))

let string_to_field = function
    Int -> "atoi( (char *) s )"
  | Float -> "atof( (char *) s )"
  | String -> "newString( (char *) s )"
  | Boolean -> "strcmp( (char *), \"true\" ) == 0 ? true : false"
  | Record(name) -> raise( Failure "Can't parse CSV object if it has an Object fie
  | Table(name) -> raise( Failure "Can't parse CSV object if it has an Table field

let translate_csv_field_loader objDecl =
  let preface = "void __" ^ objDecl.objName ^
    "_cb1( void * s, size_t i, void * p )\n{\n" in
  let locals = objDecl.objName ^ " * arr;\n" ^ objDecl.objName ^ " * item;\n\n" in
  let ptrs = "arr = (" ^ objDecl.objName ^ " *) p;\nitem = &arr[ __calls ];\n" in
  let fieldLength = List.length objDecl.objFields in
  let counts = zero_counter fieldLength in
  let fields_counts = zip objDecl.objFields (List.rev counts) in
  let print_field_transform ((name,t),num) =
    if num = 0 then "if( item->fields == 0 )\n  item->" ^ name ^ " = "
      ^ string_to_field t ^ ";\n"
    else "else if( item->fields == " ^ string_of_int num ^ " )\n item->" ^ name ^
      ^ string_to_field t ^ ";\n" in
  let epilogue = "item->fields++;\n}\n" in
  preface ^ locals ^ ptrs ^ String.concat ""
    (List.map print_field_transform fields_counts) ^ epilogue

let translate_csv_loader env objDecls =
```

```
  let objDeclFind li className =
    try
      List.find (fun obj -> obj.objName = className) li
    with Not_found -> raise( Failure "No Object definition for CSV class" ) in
  match env.csv_class with
    Some(cl) -> let objDecl = objDeclFind objDecls cl in
                translate_csv_field_loader objDecl
  | _ -> "";;

let print_csv_input = "if( argc > 2 ) {\n  if( strcmp( argv[1], \"-input\" ) == 0)
  "   fileName = argv[2];\n }\n"

let load_csv_data csvClass =
  let loadDataArray cl = "dataArray = ArrayNew( __ProgramType_" ^ cl ^ ", 1000 );\
   match csvClass with
   Some(cl) -> loadDataArray cl ^ print_csv_input ^ "LoadCSVFile( fileName, __" ^
  | _ -> "";;

let print_csv_preface = "char * fileName = NULL;\n";;

let get_csv_class = function
    [] -> None
  | hd::tl -> Some(hd.csvInputClass)

(*************************************************************************)
(* P R I N T   M E T H O D S                                             *)
(*************************************************************************)

let load_method_list tblDecls =
  List.fold_left (fun li tbl -> (tbl.tableName,"print") :: li) [] tblDecls

(*************************************************************************)
(* M A I N                                                               *)
(*************************************************************************)

let translate_main2 env varDecl_list stmt_list =
  let start = "int main( int argc, char ** argv ) {\n" in
  let varDecl_string = translate_varDecl2_assignments env varDecl_list in
```

```
  let stmts_string = String.concat "\n" (List.map (stmt env) stmt_list ) in
  start ^ print_csv_preface ^ load_csv_data env.csv_class ^ varDecl_string ^ stmts

let translate_program p =
  let env = { function_index = load_constructors p StringMap.empty;
              var_index = load_var_decls p.varDecls StringMap.empty;
              method_index = load_method_list p.tableDecls;
              type_list = load_type_list p.objDecls;
              csv_class = get_csv_class p.csvDecls } in

  (* Includes *)
  let c_prelude = "#include \"base.h\"\n" in
  let c_table_include = "#include \"table.c\"\n\n" in

  (* Build *)
  c_prelude ^ translate_types_enum (List.rev env.type_list) ^
  String.concat "\n" (List.map (translate_objectDecl env) p.objDecls) ^ "\n" ^
  print_program_typeInfo ^ translate_type_table (List.rev env.type_list) ^
  c_table_include ^
  translate_varDecl2_globals env p.varDecls ^ "\n" ^
  String.concat " " (List.map (translate_tableDecl env) p.tableDecls) ^ "\n" ^

  (* Print functions for CSV data *)
  translate_csv_loader env p.objDecls ^

  String.concat " " (List.map (translate_funcDecl env) p.funcDecls) ^ "\n" ^
(*  String.concat "\n" (List.map (translate_varDecl env) (filter_unassigned p.varD
  translate_main2 env p.varDecls (List.rev p.stmts)
```

parser.mly

```
%{ open Ast %}

%token SEMI LPAREN RPAREN LBRACE RBRACE LBRACK RBRACK COMMA COLON DOT
%token PLUS MINUS TIMES DIVIDE MOD
%token EQ NEQ LT LEQ GT GEQ
```

```
%token INT_TYPE FLOAT_TYPE STRING_TYPE BOOLEAN_TYPE
%token ARROW COMMA
%token OBJECT_DEF OBJECT_TYPE
%token TABLE_DEF TABLE_TYPE
%token CSV_DEF
%token IF THEN ELSE
%token CSV_INPUT
%token FOR FOREACH
%token RETURN
%token FUNC
%token ASSIGN
%token <string> STRING_LITERAL
%token <float> FLOAT
%token <int> LITERAL
%token <string> ID
%token <bool> BOOLLIT
%token EOF

%right ASSIGN
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD

%start program
%type <Ast.program> program

%%

program:
   /* nothing */               { { varDecls = []; funcDecls = []; objDecls = []; t
  | program var_decl           { { varDecls = $2 :: $1.varDecls; funcDecls = $1.fu
  | program func_decl          { { varDecls = $1.varDecls; funcDecls = $2 :: $1.fu
  | program obj_decl           { { varDecls = $1.varDecls; funcDecls = $1.funcDecl
  | program table_decl         { { varDecls = $1.varDecls; funcDecls = $1.funcDecl
  | program stmt               { { varDecls = $1.varDecls; funcDecls = $1.funcDecl
  | program csv_decl          { { varDecls = $1.varDecls; funcDecls = $1.funcDecls;
```

```
obj_decl:
  OBJECT_DEF ID LBRACE field_list RBRACE SEMI { { objName = $2; objFields = List.r

table_decl:
  TABLE_DEF ID LPAREN ID RPAREN LBRACE table_column_list RBRACE SEMI { { tableName

csv_decl:
  CSV_DEF LPAREN ID RPAREN SEMI                { { csvInputClass = $3 } }

table_column_list:
  table_column                                 { [$1] }
  | table_column_list COMMA table_column       { $3 :: $1 }

table_column:
  ID ARROW type_name COLON column_func     {{ tblColName = $1; tblColType = $3; tb

column_func:
   var_decl_list stmt_list expr            { { locals = List.rev $1; statements =

field_list:
  ID ARROW type_name           { [ ( $1, $3 ) ] }
  | field_list COMMA ID ARROW type_name { ($3, $5) :: $1 }

var_decl:
  type_name var_decl_id assign_opt SEMI { { vname = fst $2; varType = $1; varArray

var_decl_id:
  ID array_size_opt           { ($1,$2) }

array_size_opt:
  /*  nothing */               { None }
 | LBRACK LITERAL RBRACK       { Some($2) }

func_var_decl:
  type_name ID                 { { vname = $2; varType = $1; varArraySize = None;

assign_opt:
  /* nothing */                { None }
```

```
  |  ASSIGN initialize_assign  { Some($2) }

initialize_assign:
  | constructor_call           { [$1] }
  | init_item                  { [$1] }
  | array_init                 { $1 }

array_init:
  | LBRACE array_init_list RBRACE { List.rev $2 }

array_init_list:
  | init_item                       { [$1] }
  | array_init_list COMMA init_item { $3 :: $1 }

init_item:
  constant_expr                { $1 }

type_name:
    INT_TYPE                   { Int }
  | FLOAT_TYPE                 { Float }
  | STRING_TYPE                { String }
  | BOOLEAN_TYPE               { Boolean }
  | OBJECT_TYPE ID             { Record($2) }
  | TABLE_TYPE ID              { Table($2) }
/*  | ID                         { Record($1) }  */

func_decl:
  FUNC type_name ID LPAREN params_opt RPAREN LBRACE var_decl_list stmt_list RBRACE
    { { name = $3; formals = $5; locals = List.rev $8; body = List.rev $9; retType

params_opt:
    /* nothing */              { [] }
  | params_list                { List.rev $1 }

params_list:
    func_var_decl              { [$1] }
  | params_list COMMA func_var_decl { $3 :: $1 }
```

```
var_decl_list:
    /* Nothing */               { [] }
  | var_decl_list var_decl      { $2 :: $1 }

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

stmt:
    expr SEMI                   { Expr($1) }
  | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN LBRACE stmt_list RBRACE
  | FOREACH LPAREN ID RPAREN LBRACE stmt_list RBRACE { Foreach($3,List.rev $6) }
  | LBRACE stmt_list RBRACE { Block(List.rev $2) }
  | RETURN expr SEMI            { Return($2) }

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

constant_expr:
    LITERAL                     { Literal($1) }
  | FLOAT                       { LitFloat($1) }
  | BOOLLIT                     { LitBool($1) }
  | STRING_LITERAL              { LitString($1) }

constructor_call:
  | ID LPAREN args_opt RPAREN   { Call( $1, $3 ) }

expr:
  constant_expr                 { $1 }
  | ID                          { Id($1) }
  | CSV_INPUT                   { CSVInput }
  | ID DOT ID                   { MemberAccess($1,$3) }
  | ID DOT ID LPAREN args_opt RPAREN { MethodCall($1,$3,$5) }
  | ID LBRACK expr RBRACK       { ArrayAccess($1,$3)  }
  | 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 MOD expr                { Binop( $1, Mod, $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 ASSIGN expr             { Assign($1,$3) }
  | IF LPAREN expr RPAREN  THEN LBRACE stmt_list expr RBRACE
    ELSE LBRACE stmt_list expr RBRACE      { IfThenElse( $3, $8, $13, $7, $12) }
  | constructor_call             { $1 }

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

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

sast.ml

```
module StringMap = Map.Make(String)

(* Symbol Table *)
type symbol_table = {
  parent : symbol_table option;
  vars : Ast.varDecl list;
  }

(* Translation Environment  *)
type env = {
  function_index : Ast.funcDecl StringMap.t;
  object_index : Ast.objDecl StringMap.t;
```

```
  var_decls : Ast.varDecl StringMap.t;
  locals : symbol_table;
  }

let symbol_table_add_var scope (decl : Ast.varDecl) =
  decl :: scope.vars

let rec symbol_table_find_var scope name =
  try
    List.find (fun (decl : Ast.varDecl) -> decl.vname = name) scope.vars
  with Not_found ->
    match scope.parent with
      Some(parent) -> symbol_table_find_var parent name
    | _ -> raise Not_found

(* Expression Sum Type *)
type expression_detail =
    Literal of int
  | LitFloat of float
  | LitBool of bool
  | LitString of string
  | Id of string
  | Binop of Ast.expr * Ast.op * Ast.expr
  | Assign of Ast.expr * Ast.expr
  | Call of string * Ast.expr list
  | IfThenElse of Ast.expr * Ast.expr * Ast.expr * Ast.stmt list * Ast.stmt list
  | ArrayAccess of string * Ast.expr
  | MemberAccess of string * string
  | MethodCall of string * string * Ast.expr list
  | CSVInput
(*  | MemberAccess of expr * expr
  | ArrayAccess of expr * expr  )
  | If of expr * expr * expr *)
  | Noexpr

type expression = expression_detail * Ast.progType

type statement =
```

```
    Block of Ast.stmt list (* Maybe a scope *)
  | Expression of expression * env * Ast.expr
  | For of Ast.expr * Ast.expr * Ast.expr * statement list * env
  | Foreach of string * statement list * env
  | Return of Ast.expr

(* Zip two lists *)
let rec zip l1 l2 =
  match l1,l2 with
  | [],_ -> []
  | _, []-> []
  | (x::xs),(y::ys) -> (x,y) :: (zip xs ys)

(* Check expression -> answer a Sast.expression *)
let rec check_expr env = function
    Ast.Literal(v) -> (Literal(v), Ast.Int )
  | Ast.LitFloat(v) -> (LitFloat(v), Ast.Float )
  | Ast.LitBool(v) -> (LitBool(v), Ast.Boolean )
  | Ast.LitString(v) -> (LitString(v), Ast.String )
  | Ast.Id(name) ->

     (* Check local variable is in scope *)
     let varDecl =
       try
         StringMap.find name env.var_decls
       with Not_found ->
          raise( Failure "Could not find local variable")
     in
     Id(name), varDecl.varType

  | Ast.Binop(e1,op,e2) ->
     let (_,t1) = check_expr env e1
     and (_,t2) = check_expr env e2 in
         if op <> Ast.Equal && op <> Ast.Neq && op <> Ast.Leq && op <> Ast.Less &&
           (* Make sure both left and right side are the same type TODO: Other typ
           if t1 <> t2 then raise( Failure "Mismatched type")
           else Binop( e1, op, e2), t1
         else
```

```
        Binop( e1, op, e2), Ast.Boolean
| Ast.Assign(eVar,e) -> (
   let (_,t1) = check_expr env eVar
   and (_,t2) = check_expr env e in
   Assign(eVar,e),t1)

       (*     if (t1 != t2) then (Assign(eVar,e),t1)
     else raise( Failure "Mismatch type in assignment" ))*)

| Ast.Call( name, expr_li ) -> (

   match name with
     "Print" -> Call( name, expr_li), Ast.Int
   | _ -> (

   (* Check name exists in function table *)
    let func_decl name =
      try
        StringMap.find name env.function_index
      with Not_found ->
        raise( Failure "A: No function name found" )
    in

   (* Check count of arguments and compare to func definition *)
    let compareArguments exprLi formals =
      if List.length exprLi <> List.length formals
      then raise( Failure "Arguments and parameters don't match")
      else
        let zipList = zip exprLi formals in
        let result = checkParameters env zipList in
        match result with
          false -> raise (Failure "Mismatch in arguments and parameters" )
        | true -> true
     in

   (* TODO: Check return type *)
     let fd = func_decl name in
     if ((compareArguments expr_li fd.formals) = false)
```

```
      then raise( Failure "Mismatch in arguments and parameters" )
      else Call(name,expr_li), fd.retType))

  | Ast.IfThenElse( e1, e2, e3, sl1, sl2 ) ->

     (* Get Types *)
     let (_,t1) = check_expr env e1
     and (_,t2) = check_expr env e2
     and (_,t3) = check_expr env e3 in

     (* Check type of predicate *)
     let predicate = t1 <> Ast.Boolean in

     (* Check that t2 and t3 match *)
     let thenAndElse = t2 <> t3 in

     (* Check error and answer *)
     if predicate || thenAndElse then raise( Failure "Mismatch type in if-then-els
     else
       IfThenElse( e1, e2, e3, sl1, sl2), Ast.Int (* Use Int Type here *)
  | Ast.ArrayAccess(name,expr) -> let arrVar =
     try
         StringMap.find name env.var_decls
     with Not_found ->
       raise( Failure "No variable found for array access" ) in
     let (_,t) = check_expr env expr in ArrayAccess(name,expr),t
(*     if (t = arrVar.varType) then ArrayAccess(name,expr),t
       else raise( Failure "Type mismatch")*)


  | Ast.MemberAccess(name,field) ->
     let varD =
       try
         StringMap.find name env.var_decls
       with Not_found ->
          raise( Failure "Could not find local variable for member access") in
     let clName = function
       Ast.Record(name) -> name
```

```
      | _ -> raise( Failure "Not an object for member access" ) in
     let obj =
     try
          StringMap.find (clName varD.varType) env.object_index
     with Not_found ->
       raise( Failure "No object found for array access" ) in
     let (name,t) =
       List.find (fun (a,b) -> a = field) obj.objFields in
     MemberAccess(name,field),t
  | Ast.MethodCall(s1,s2,e_li) -> MethodCall(s1,s2,e_li),Ast.Int
  | Ast.CSVInput -> CSVInput,Ast.Int
   | Ast.Noexpr -> raise( Failure "No expression here")

and checkParameters env zipList =
  let cmp ( e, (decl : Ast.varDecl) ) =
    let _,t = check_expr env e in
    if t <> decl.varType then None
    else Some(true)
  in
  let results = List.map cmp zipList in
  let fail = List.filter (fun a -> if a = None then true else false) results in
  match fail with
    [] -> true
  | _ -> false


(* Check individual statements -> answer Sast.statement *)
let rec check_stmt environment = function
    Ast.Block(stmts) -> Block(stmts)   (* TODO *)
  | Ast.Expr(e) ->
     let result = check_expr environment e in
     Expression( result, environment, e )
  | Ast.For(e1,e2,e3,li) ->
     (* Check that e2 is a Boolean *)
     let (_,t) = check_expr environment e2 in

     (* Check individual statements *)
     let statementList = (List.map (check_stmt environment) li) in
```

```
      (* Answer checked for loop *)
      if t != Ast.Boolean then raise( Failure "For loop doesn't have a Boolean")
      else For(e1,e2,e3,statementList,environment)
  | Ast.Return(e) -> let (_,t) = check_expr environment e in
      Return(e)
  | Ast.Foreach( name, stmt_li ) ->(
      (* Lookup name *)

      (* Check statements *)
      let statementList = (List.map (check_stmt environment) stmt_li) in

      (* Answer checked foreach loop *)
      Foreach( name, statementList, environment ));;

let load_var_decls _map (varDecls : Ast.varDecl list) =
  List.fold_left (fun m (varD : Ast.varDecl) -> StringMap.add varD.vname varD m) _

(* Check function definitions and answer bool option *)
let check_func_decl environment (funcDecl : Ast.funcDecl) =(

  (* Check if name exists in environment *)
  let checkFuncNameExists =
    try
      StringMap.find funcDecl.name environment.function_index
    with Not_found ->
      raise( Failure "B: No function name found" )
  in

  (* Build new environment with empty locals *)
  let env = {
    function_index = environment.function_index;
    object_index = environment.object_index;
    var_decls = load_var_decls (load_var_decls (environment.var_decls) funcDecl.lo
    locals = { parent = None; vars = [] } } in

  (* check statements -> get back a statement list *)
  let stmts = (List.map (check_stmt env) funcDecl.body) in
```

```
  (* Answer OKAY *)
  checkFuncNameExists;
  stmts)


let load_function_index _map funcDecls = (
  List.fold_left (fun m (func : Ast.funcDecl) -> StringMap.add func.name func m) _


let load_object_index _map objDecls = (
  List.fold_left (fun m (obj : Ast.objDecl) -> StringMap.add obj.objName obj m) _m



let check ( program : Ast.program ) =

  (* Create Environment for scope and tracking *)
  let environment = {
    function_index = load_function_index StringMap.empty program.funcDecls;
    object_index = load_object_index StringMap.empty program.objDecls;
    var_decls = load_var_decls StringMap.empty program.varDecls;
    locals = { parent = None; vars = [] } } in

  let funcChecks = List.map (check_func_decl environment) program.funcDecls in
  let stmtChecks = List.map (check_stmt environment) program.stmts in
  funcChecks;
  stmtChecks;
  Some(true)


  (* var Decl names *)
(*  let var_decl_names varDeclList =
    List.map (fun (varDecl : Ast.varDecl) -> varDecl.vname) varDeclList
  in*)

  (* Load global variables into the environment *)
(*  let load_global_vars = List.fold_left (fun m obj -> StringMap.add obj true m )
```

```
    environment.global_index (var_decl_names program.varDecls)
  in*)

  (* Load objects into environment *)

  (* get function names *)
(*  let function_names funcDecl = List.map
    (fun (funcDecl : Ast.funcDecl) -> funcDecl.name) funcDecl
  in*)

  (* Also load declarations for all functions *)
(*  let load_function_names = List.fold_left (fun m obj -> StringMap.add obj true
    environment.function_index (function_names program.funcDecls )
  in*)

  (* Load function declarations *)
(*  let load_function_declarations (funcDecls : Ast.funcDecl list) =
    List.fold_left (fun li obj -> obj :: li) [] funcDecls
  in*)

  (* Need to check that objects and variable declarations are okay *)

  (* Check the functions *)
(*  let check_func_decls env funcDecls =
    List.fold_left (check_func_decl env) (Some true) funcDecls
  in*)

  (* Answer check of all functions *)
(*  load_global_vars;
  load_function_names;
  load_function_declarations program.funcDecls;
  check_func_decls environment program.funcDecls*)
```

scanner.ml

```
{ open Parser}
```

```
rule token = parse
  [' ' '\r' '\n' '\t'] { token lexbuf }   (* Whitespace *)
| "/*"                 { comment lexbuf } (* Comments *)
| '('                  { LPAREN }
| ')'                  { RPAREN }
| '{'                  { LBRACE }
| '}'                  { RBRACE }
| '['{ LBRACK }
| ']'{ RBRACK }
| ';'                  { SEMI }
| ':'                  { COLON }
| '+'                  { PLUS }
| '-'                  { MINUS }
| '*'                  { TIMES }
| '/'                  { DIVIDE }
| ":="                 { ASSIGN }
| ","                  { COMMA }
| "->"                 { ARROW }
| "="                  { EQ }
| "!="                 { NEQ }
| '<'                  { LT }
| "<="                 { LEQ }
| ">"                  { GT }
| ">="                 { GEQ }
| "%"                  { MOD }
| "if"                 { IF }
| "then"               { THEN }
| "else"               { ELSE }
| "for"                { FOR }
| "foreach"            { FOREACH }
| "return"             { RETURN }
| '.'{ DOT }
| "Input"              { CSV_INPUT }      (* CSV Input Array *)
| "Int"                { INT_TYPE }       (* Integer Type *)
| "String"             { STRING_TYPE }    (* String Type *)
| "Boolean"            { BOOLEAN_TYPE }   (* Boolean Type *)
| "Float"              { FLOAT_TYPE }     (* Float Type *)
```

```
| "Object"              { OBJECT_DEF }     (* Object Def *)
| "Obj"                 { OBJECT_TYPE }    (* Object Type *)
| "Table"               { TABLE_DEF }      (* Table Def *)
| "Tab"                 { TABLE_TYPE }     (* Table Type *)
| "CSV"                 { CSV_DEF }        (* CSV Def *)
| "Func"                { FUNC }           (* Function *)
| "true" | "false" as lit { BOOLLIT(bool_of_string lit) }  (* True/False Value *)
| eof { EOF }    (* End Of File *)
|  ('-')?(['0'-'9']+)"."(['0'-'9']+)?(['e''E']('+'|'-')?['0'-'9']+)?('F''f''L''l')
   ('-')?"."(['0'-'9']+)(['e''E']('+'|'-')?['0'-'9']+)?('F''f''L''l')?
   ('-')?(['0'-'9']+)"."?['e''E']('+'|'-')?(['0'-'9']+)('F''f''L''l')? as num
      { FLOAT( float_of_string num) }
| ('-')?['0'-'9']+ as lit     { LITERAL(int_of_string lit) }
| ['a'-'z' 'A'-'Z' '_']['a'-'z' 'A'-'Z' '0'-'9' '_']* as id { ID(id) }
| '"' [^'"']* '"' as strlit   { STRING_LITERAL(strlit) }
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

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

table.c

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

/*typedef enum __ProgramType_ {
  __ProgramType_Int = 1,
  __ProgramType_Float = 2,
  __ProgramType_String = 3,
  __ProgramType_Boolean = 4,
  } __ProgramType_; */

typedef struct __TableReturn_ {
  __ProgramType_ returnType;
```

```
  char * _str;
  int _int;
  double _float;
  void * _ptr;
  int _boolean;
} __TableReturn_;

typedef __TableReturn_ (*__TableColumnFunc_)();
typedef void * (*__TableColumnGetItemFunc_)();

typedef struct __TableColumn_ {
  char * name;
  __ProgramType_ returnType;
  __TableColumnFunc_ func;
} __TableColumn_ ;

typedef struct __Table_ {
  char * name;
  int columnCount;
  __TableColumn_ * columns;
  __TableColumnGetItemFunc_ getItem;
  int objectType;
  int objectArrayCount;
  void * objectArray;
} __Table_;

#define TABLE_START( name ) __TableColumn_ _TableColumnList_##name##_[] = {
#define TABLE_COLUMN( name, type, func ) { name, type, func },
#define TABLE_END(name) };

#define TABLE_COLUMNS(name) _TableColumnList_##name##_
#define TABLE_GETITEM(name) _##name##_getItem_

static char * newString( char * name )
{
  char * new;
  int k, n;
```

```
  if( name == NULL )
    return NULL;

  k = strlen( name );
  n = k + 1;
  new = (char *) malloc(n);
  strncpy( new, name, n );
  new[n] = '\0';

  return new;
}

static void * ObjectNew( __ProgramType_ type )
{
  __ProgramTypeInfo_ * info;
  int listSize;
  void * mem;

  listSize = sizeof(__programTypeList_)/sizeof(__ProgramTypeInfo_);
  if( type < 1 || type >= listSize )
    {
    printf("Error: Invalid type for Object Creation\n");
    return NULL;
    }

  info = &__programTypeList_[type];
  if( info == NULL )
    {
    printf("Error: Invalid type for Object Creation\n");
    return NULL;
    }

  mem = malloc( info->size );
  memset( mem, 0, info->size );

  return mem;
}
```

```
static void * ArrayNew( __ProgramType_ type, int size )
{
  __ProgramTypeInfo_ * info;
  int listSize, n;
  void * mem;

  listSize = sizeof(__programTypeList_)/sizeof(__ProgramTypeInfo_);
  if( type < 1 || type >= listSize )
    {
    printf("Error: Invalid type for Object Creation\n");
    return NULL;
    }

  info = &__programTypeList_[type];
  if( info == NULL )
    {
    printf("Error: Invalid type for Object Creation\n");
    return NULL;
    }

  n = info->size * size; /* Total Bytes needed for the array */
  mem = malloc( n );
  memset( mem, 0, n );

  return mem;
}

static void freeString( char * name )
{
  if( name == NULL )
    return;

  free( name );
}

static __Table_ * __TableNew( char * name,
  __TableColumnGetItemFunc_ getItemFunc,
  __TableColumn_ * columnList, int columnCount,
```

```
  int objectType, int objectArrayCount,
  void * objectArray )
{
  __Table_ * table;

  table = (__Table_ *) malloc( sizeof(__Table_) );
  memset( table, 0, sizeof(__Table_) );

  table->name = newString(name);
  table->getItem = getItemFunc;
  table->columns = columnList;
  table->columnCount = columnCount;
  table->objectType = objectType;
  table->objectArrayCount = objectArrayCount;
  table->objectArray = objectArray;
  return table;
}

static __TableFree( __Table_ * table )
{
  free( table );
}

static void __TablePrint( __Table_ * table )
{
  int i, j, m, n;

  /* Init */
  m = table->columnCount;
  n = table->objectArrayCount;

  /* Print Header */
  for( i=0; i < m; i++ )
    {
    __TableColumn_ * col;

    col = &table->columns[i];
    printf( "%15s", col->name );
```

```
    }
  printf("\n");

  for( i=0; i < n; i++ )
    {
    for( j=0; j < m; j++ )
      {
      __TableColumn_ * col;
      __TableReturn_ ret;

      col = &table->columns[j];
      ret = col->func( table->getItem(table->objectArray,i) );
      //      switch( ret.returnType )
      switch( col->returnType )
         {
        case __ProgramType_Int:
          printf( "%14d", ret._int );
          break;
        case __ProgramType_Float:
          printf( "%14f", ret._float );
          break;
        case __ProgramType_String:
          printf( "%14s", ret._str );
          break;
        case __ProgramType_Boolean:
          printf( "%14d", ret._boolean );
          break;
        }

      printf(",");
      }
    printf("\n");
    }
}
```