Return of the Table
Final Report

Jared Pochtar and Michael Vitrano

January 7, 2012

Contents

1 Language White Paper **4**
- 1.1 Introduction . . . 4
- 1.2 DataTypes . . . 4
  - 1.2.1 Table Operations . . . 5
- 1.3 Other Features . . . 6

2 Language Tutorial **7**
- 2.1 Running the Compiler . . . 7
- 2.2 Hello World! . . . 7
- 2.3 Running Hello World . . . 7

3 Language Reference Manual **9**
- 3.1 Introduction . . . 9
- 3.2 Basic Syntax . . . 9
  - 3.2.1 Comments . . . 9
  - 3.2.2 Keywords . . . 9
  - 3.2.3 Identifiers . . . 10
  - 3.2.4 Literals . . . 10
  - 3.2.5 String Literals . . . 10
  - 3.2.6 Variables . . . 10
- 3.3 Datatypes . . . 10
  - 3.3.1 Primitive Types . . . 10
  - 3.3.2 User-Defined Types . . . 10
  - 3.3.3 Records . . . 11
  - 3.3.4 Tables . . . 11
  - 3.3.5 Nulls . . . 11
- 3.4 Operators . . . 11
  - 3.4.1 Assignment Operator = . . . 11
  - 3.4.2 Comparison Operators . . . 11
  - 3.4.3 And and Or Operators, &&, ||, and, or . . . 12
  - 3.4.4 Integer/Floating Point Operations +, -, *, / . . . 12
  - 3.4.5 Compound Integer/Floating Point Operations +=, -=, *=, /= . . . 13
  - 3.4.6 String Operators . . . 13
  - 3.4.7 Record Operators . . . 13
  - 3.4.8 Table Operators . . . 14

3.5 Casting . . . 15
3.5.1 Primitive Casting . . . 15
3.5.2 Table Casting . . . 16
3.6 Statements . . . 16
3.6.1 Expression statements . . . 16
3.6.2 Compound statements . . . 16
3.6.3 If-else statements . . . 16
3.6.4 Iteration statements . . . 16
3.7 Functions . . . 17
3.7.1 Function Definitions . . . 17
3.7.2 Function Calls . . . 18
3.8 Scope . . . 18
3.8.1 Namespaces . . . 18
3.8.2 Blocks . . . 18
3.8.3 Lexical scope of identifiers . . . 18
3.9 Standard Library . . . 18
3.9.1 print . . . 18
3.9.2 stdin . . . 18
3.9.3 setWorkingDir . . . 19
3.9.4 system . . . 19
3.9.5 argc and argv . . . 19
3.9.6 Table Functions . . . 19
3.9.7 String Functions . . . 20

4 Project Plan **21**
4.1 Team Responsibilities . . . 21
4.2 Programming Conventions . . . 21
4.2.1 Java . . . 21
4.2.2 OCaml . . . 21
4.2.3 Time Line . . . 21

5 Architectural Design **23**
5.1 Block Diagram . . . 23
5.2 Command Line Interface . . . 23

6 Test Plan **25**
6.1 Example Test Programs . . . 25
6.1.1 Map, Filter, Join . . . 25
6.1.2 User Purchases . . . 27
6.1.3 String Interpolation . . . 30
6.2 Test Selection . . . 31
6.3 Test Suite . . . 32
6.3.1 individualTest.sh . . . 32
6.3.2 testAll.sh . . . 32
6.3.3 testClean.sh . . . 33
6.3.4 Automation . . . 33

7 Lessons Learned **34**
7.1 Michael's Lessons . . . 34
7.2 Jared's Lessons . . . 34
7.3 Recommendations to Future Groups . . . 35

8 Code Listing **36**
8.1 Translator . . . 36
8.1.1 ast.ml . . . 36
8.1.2 main.ml . . . 37
8.1.3 namespace.ml . . . 38
8.1.4 overloading.ml . . . 43
8.1.5 parser.mly . . . 47
8.1.6 scanner.mll . . . 51
8.1.7 translator.ml . . . 53
8.2 Framework . . . 59
8.2.1 WorkingDir.java . . . 59
8.2.2 UserDefinedType.java . . . 59
8.2.3 TupleType.java . . . 61
8.2.4 Table.java . . . 62
8.2.5 RtUtil.java . . . 66
8.2.6 Record.java . . . 66
8.2.7 JoinPredicate.java . . . 68
8.2.8 JoinMap.java . . . 68
8.2.9 FilterPredicate.java . . . 68
8.2.10 FilterMap.java . . . 69
8.2.11 Data.java . . . 69
8.2.12 CsvInterpreter.java . . . 70

Chapter 1

Language White Paper

1.1 Introduction

Return of the Table, RT for short, is a language intended to make working with tables in a programmatic setting easier. It is designed so that a programmer can use the full functionality of tables for retrieving, storing, and manipulating data.

Relational databases form the core of many information technology systems that need to store large amounts of data in an efficient and organized manner. While interacting with these databases with SQL is useful for efficiently accessing a particular subset of data, there are many tasks that are either cumbersome or impossible. For example, applying a function to each record in a relational table is a multistep process, requiring the programmer to write a query to retrieve the relevant data, parse it, apply the function and return the new data to the table. Further, there is no way to use SQL constructs to check input validity using regular expressions.

RT attempts to simplify the interaction with relational databases by making tables first- class objects. Users are able to load information from a database, manipulate it using the familiar imperative programming paradigm and optionally commit the data back to the database.

1.2 DataTypes

The primitive datatypes in RT are drawn from the basic SQL datatypes of varchar, integer and real. RT names these string, int and float respectively. RT also includes a bool type. In addition to the four primitive types, RT allows the user to create User-Defined Types (UDTs), which are comprised of an arbitrary combination of RT primitive types. Further, there are Compound types, which are the concatenation of UDTs. UDTs and Compound types are intended to mimic the schemas of tables in relational databases. In relational databases, tables consist of records adhering to the schema defined for the table. This paradigm is mimicked in RT, where tables consist of records of a single UDT. It should be noted that Records are also considered to be tables with exactly one row of data. Below is an example of defining a UDT and creating a table of that with Records of that type:

```
type User{
string name
string city
int age
```

```
float income
bool active
}

User table users = table: User(``Michael'',``North Brunswick'', 21, 1000.50, false)
users = users +  User(``Jared'',``New York'', 16, 1000000.50, true)
```

1.2.1 Table Operations

RT includes a number of operations for working with Tables. These fall into 3 categories, Filters, Joins and Maps. The Filter operation takes one table and one boolean predicate as arguments and returns a new table consisting of the Records from the original table for which the predicate evaluates to true. Joins take two tables and a predicate, performs the cross product of the tables and returns filtered the results based on the predicate. Note that the return type of a Join is the Compound of the two original table's types. Maps can take one or two tables as well as an expression that returns a Record. The Map returns a table consisting of the Records returned by the Record expression. In a Map with one table, the Record expression is evaluated for every Record in the original table. In a Map with two tables, the Record expression is evaluated for every Record in the cross product of the two original tables. The following is an example of the different table functions:

```
type User{
string name
int age
}

User table users = table: User(``Michael'', 21)
users = users +  User(``Jared'', 16)

type Product{
string itemName
float cost
}

Product table products = table: Product(``Soap'', 1.99)
products = products +  Product(``Chocolate'', 16)

type Purchased{
string cName
string pName
}

User table michael = users[.name:``Michael''] // filter

User#Product table users_products = users[true]products // join
```

```
// single table map
Purchased table purchases1 = users_products[Purchased(name, itemName)]

// double table map, produces the same result as purchases1
Purchased table purchases2 = users[Purchased(name, itemName)]products
```

1.3 Other Features

RT compiles to Java code and therefore provides many of the operators that Java provides for the RT primitive types. Additionally, there are RT libraries that facilitate printing of primitives, Records and Tables as well as interfacing with stored data in comma separated value format.

Chapter 2

Language Tutorial

2.1 Running the Compiler

To compile an RT program the following command is run:

```
./rtc.sh <source-file> <optional destination> <optional debug>
```

Running the rtc.sh script requires that a source-file is given. If no destination is provided, the compiled .class is placed in the present working directory. If the debug flag is raised, the intermediate .java file is also placed into either the present working directory or the destination depending on whether one is provided.

2.2 Hello World!

The most basic RT program will define a User-Defined type, create a table of this type, add data and print that data to the user. Defining a UDT is done with the *type* keyword as show in the example program below:

```

type Hello_world {
	string howdy
}

Hello_world greeting = Hello_world("Hello, World!")

Hello_world table hellos = table: Hello_world

hellos = hellos + greeting

hellos = hellos + Hello_world("Isn't this wonderful?")

print(hellos)
```

Next, we will declare a Record of type *Hello_world* named *greeting* containing the string "Hello, World!". We then instantiate an empty *Hello_world* table named *hellos* and append *greeting*. We also append an annoymous *Hello_world* to *hellos* before we print the table in the last line of the program.

2.3 Running Hello World

To run a compiled .class, the following command is executed:

```
./rt.sh <class-file> <optional args ... >
```

Running the rt.sh script links the compiled .class file against the RtLib and runs executes it. Command line arguments can be passed to the program by appending them to the end of the call to the script.
The output of HelloWorld.rt is:

```
|  Hello_world's howdy  |
|     Hello, World!     |
| Isn't this wonderful? |
```

Chapter 3

Language Reference Manual

3.1 Introduction

RT, short for Return of the Table, is a language intended to make working with tables in a programmatic setting easier. It is designed so that a programmer can use the full functionality of tables for retrieving, storing, and manipulating data.

Relational databases form the core of many information technology systems that need to store large amounts of data in an efficient and organized manner. While interacting with these databases with SQL is useful for efficiently accessing a particular subset of data, there are many tasks that are either cumbersome or impossible. For example, applying a function to each record in a relational table is a multistep process, requiring the programmer to write a query to retrieve the relevant data, parse it, apply the function and return the new data to the table. Further, there is no way to use SQL constructs to check input validity using regular expressions.

RT attempts to simplify the interaction with relational databases by making tables first- class objects. Users are able to load information from a database, manipulate it using the familiar imperative programming paradigm and optionally commit the data back to the database.

3.2 Basic Syntax

Spaces, tabs, and carriage returns constitute whitespace; they are necessary to separate tokens but are not recognized in any other way by the compiler. Newlines, typically considered whitespace, are not whitespace in the RT language; they are used to denote the end of an expression-statement and other required points. The comma can be used in RT to break up long lines in RT without signaling the end of an expression-statement

3.2.1 Comments

RT supports both single and mutli line comments. The characters `//` introduce a single line comment;a single line comment is terminated by a newline. The `/*` characters introduce a multi line comment; all subsequent text will be regarded as comments until the `*/` character is found. RT supports nested comments.

3.2.2 Keywords

The following are keywords in RT and are not to be used otherwise.

int	null	string	float	bool	type	for	if
else	while	table	true	false	void	do	
return	and	or					

3.2.3 Identifiers

An identifier is anything beginning with a letter and continuing with any combination of letters, numbers, primes, and underscores '_'. It can be used to name variables, user defined types, or functions.

3.2.4 Literals

Any sequence of digits is an integer literal. Any set of two sequences of digits separated by a decimal point '.' is a floating-point literal. Any sequence of characters enclosed by double quotes is a string literal.

3.2.5 String Literals

String literals may have any number of interpolations in them. An interpolation is an expression inside a string, which is cast to a string and concatenated to the part of the string before it, and then the following string and any other interpolations are concatenated to that. The $ symbol denotes a string interpolation. The standard form of a string interpolation is

"string content $interpolated expression more string content" The interpolated expression can be any RT expression that can be cast to a string. There is an abbreviated form, which is simply "$stringvariable", where the braces are omitted. In this form the $ is followed by an identifier which is assumed to be a variable representing a string. It cannot include a member access.

3.2.6 Variables

A declaration specifies the interpretation given to an identifier, but does not necessarily reserve storage associated with it. Declarations have the form:

datatype identifier

Where *datatype* is a valid RT datatype as described below, and *identifier* is a legal identifier as described above.

3.3 Datatypes

3.3.1 Primitive Types

There are four primitive datatypes in RT: `int`, `float`, `string`, `bool`. For the remainder of this Reference Manual, `int` and `float` will be refered to as Arithmetic types.

3.3.2 User-Defined Types

User-defined types, (UDT), consist of a sequence of primitive types. The syntax for specifying a UDT using the `type` keyword is as follows:

`type` *type-name*{
  member-declaration-list
}

Compound types are UDTs assembled as the concatenation of the members of the component UDTs. The syntax for declaring a Compound UDT is as follows:

type1#type2#type3

There can be an arbitrary number of component UDT types in a Compound type. Compound types containing two or more of the same UDT are allowed, but it is only possible to access the first instance of the UDT in the type. There are no constructors for compound types and they must be created using Maps or Joins, which are described below.

3.3.3 Records

Records are instances of a UDT. They can be constructed using the following syntax:

type-name(expr1, expr2, ... , exprN)

Where the return type for each *(expr)* matches that of the corresponding member of the UDT. Records are passed by reference and their references are preserved across joins. For example, A member set in a record that is the product of a join is the same member in the component record that donated it's members to the joined record.

3.3.4 Tables

Tables are typed according to a UDT and comprised of collection of Records of the same type.

3.3.5 Nulls

Every record or table type has a distinct null. These are declared as null Type for a null of type Type, or null Type table for a null of type Type table. Nulls of various types are not equal, and cannot be compared.

3.4 Operators

RT has numerous operators defined over primitive and user-defined datatypes.

3.4.1 Assignment Operator =

Assignment is performed as follows:

id = expression

Where the return type of *expression* matches that of *id*, and the left hand side is either a non-captured variable, or a member access (even off of a non-variable, such as a function return).

3.4.2 Comparison Operators

Comparison operators return a `bool` according to the definitions below.

Equality Operator ==

expression1 == expression2

Tests the equality of the two operand expressions and returns a bool. For primitive types, tests whether the literal value of each *expression* is equivalent. The equality test is deep and the operator will recursively check nested user-defined types. For table types, tests whether the expressions refer to the same table in memory. Use of this operator for comparison of primitive and table types will result in a compile time error.

Not Equal Operator !=

expression1 != expression2

Tests the inequality of the two operand expressions and returns a bool. For primitive types, tests whether the literal value of each *expression* is not equivalent. The inequality test is deep and the operator will recursively check nested user-defined types. For table types, tests whether the expressions refer to the same table in memory. Use of this operator for comparison of primitive and table types will result in a compile time error.

Inequality Operators >, <, >= , <=

expression1 > expression2
expression1 < expression2
expression1 >= expression2
expression1 <= expression2

These operators are defined only for expressions with arithmetic types. Tests whether *expression1* is greater than, less than, greater than or equal to, or less than or equal to *expression2* respectively. Use of this operator on expressions with non-Arithmetic types will result in a compile time error.

3.4.3 And and Or Operators, &&, ||, and, or

expression1 && expression2
expression1 and expression2
expression1 ||, expression2
expression1 or expression2

These operators are defined only for expressions of type *bool*. The first two examples will return the boolean AND of *expression1* and *expression2*. The second two examples will return the boolean OR of *expression1* and *expression2*.

3.4.4 Integer/Floating Point Operations +, -, *, /

expression1 + expression2
expression1 - expression2
*expression1 * expression2*
expression1 / expression2

These operators are defined only for expressions with arithmetic types. Integers will automatically be promoted to floating points if the other operand is a float. These operations will return an `int` if both

operands are of type `int` and will return type `float` otherwise. These operations will return *expression1* plus, minus, times, or divided by *expression2* respectively. All remainders will be ignored in integer division. Use of these operators on expressions with non-Arithmetic types will result in a compile time type error.

3.4.5 Compound Integer/Floating Point Operations +=, -=, *=, /=

id += expression
id -= expression
*id *= expression*
id /= expression

These operators are defined only for expressions with arithmetic types. These operators perform an operation on the value stored in *id* and the value returned by *expression* and stores the value in the variable identified by *id*. If the type of *id* is *int* then the type of *expression* must also be *int*. If the type of *id* is *float* then the type of *expression* can be *int* or *float*.

3.4.6 String Operators

String Concatenation Operator +

expression1 + expression2

This returns a new string that is a concatenation of the strings represented by *expression1* and *expression2*.

Compound String Concatenation Operator +=

id += expression

This concatenates the string stored in *id* and returned by *expression* and stores it in the variable identified by *id*.

3.4.7 Record Operators

Column Access

expr.id
expr.id.id

Accessing the data from a record is done with the . notation. If *expr* refers to a non-record, the compiler reports an error. If the later of the notations is used, the first id after the expression refers to the type the member was defined on. This is redundant for records of a non-tuple type, but is useful for unambiguously referring to members of join products which might be named the same thing, but on different component types. Each user defined type cannot have duplicately named members. The last id after the "." refers to the member name. In RT, members of user defined types (UDTs) are interchangeably called members, elements, or columns. If there is no column of that matches *id*, RT tries to call a *id(expr)* as a function. Note that this does not happen for *expr.id.id* accesses.

3.4.8 Table Operators

Automatic Load

When declaring a table, RT will automatically check whether there is a .csv file in the current working directory where filename in filename.csv matches the declared name of the table. If so, the data from that file is loaded into the table that was declared.

Append Operator +

expr1 + expr2

Returns a table with the contents of the table returned by *expr2* appended to the contents of the table returned by *expr1*. *expr1* or *expr2* can be of Table or Record type.

Head Operator !

!expr1

Returns the first record of the table returned by *expr1*.

Pipe Operator |

expr | id stmt

Iterates over the rows of the table returned by *expr*. In each iteration, the current row is bound to the name *id* inside of stmt.

Filter Operator

expr[bool-expr]

Returns a new table comprised of the Records in the table returned by *expr* for which *bool-expr* evaluates to *true* . There is a compile-time error if *expr* is not a table or if *bool-expr* does bot return boolean value. The @ symbol can be used to represent the current Record being iterated on in *bool-expr*. The *:* can be used to represent equality in the *bool-expr*.

Join Operator

expr1[bool-expr]expr2

Returns a new table which is the cross product of the tables specified by *expr1* and *expr2*, filtered by the boolean predicate bool. There is a compile-time error if *expr1* or *expr2* are not tables or if *bool-expr* does bot return boolean value. Joins between two tables of the same UDT will result in a compile time error. The identifier *a* can be used to reference current Record being joined from the table resulting from *expr1* and *b* can be used to reference current Record being joined from the table resulting from *expr2*. Any variables named *a* or *b*will not be accessible in *bool-expr*. The *:* can be used to represent equality in the *bool-expr*.

Unconditional Join Operator|+|

expr1|+|*expr2*

This equivalent to a Join where the *bool-expr* is always true. This is useful for appending tables "horizontally", or appending the columns of one on another.

Maps

expr1[record-expr]
expr1[record-expr]expr2

Each statement returns a new table comprised of the Records that are returned by *record-expr*. In the first map shown above, *record-expr* is evaluated for every Record in the table returned by *expr1*. In the second map, *record-expr*is evaluated for every Record in the cross product of the tables returned by *expr1* and *expr2*. In the single table Map, the @ symbol can be used to represent the current Record being iterated on in *bool-expr*. in a double table Map, the identifier *a* can be used to reference current Record being joined from the table resulting from *expr1* and *b* can be used to reference current Record being joined from the table resulting from *expr2*. Any variables named *a* or *b*will not be accessible in *bool-expr*.

Conditional Maps

expr1[bool-expr ? record-expr]
expr1[bool-expr ? record-expr]expr2

Conditional Maps are equivalent to Maps except that the Record returned by *bool-expr* are only added to the returned table if *bool-expr* is true. The *:* can be used to represent equality in the *bool-expr*.

Predicate Variable Look-Up Rules

In a filter or join predicate, the @ symbol refers to the current row being tested or mapped, in accordance with the type of the predicate expression. This is sometimes called the default variable. In a join, this has the type *lh#rhs*, where ls is the type of the left hand side table expression of the join, and rhs is the type of the right hand side. Also in a join, the variables *a* and *b* refer to the left hand and right hand rows being tested, respectively. *a* and *b* shadow any other variables declared *a* or *b*, respectively. In member accesses, the @ may be omitted. *@.member* is the same as *.member* in a join or filter predicate. This is only valid in a predicate because @ is only defined in predicates.

Additional syntactic sugar allows *.member* to be referred to simply as *member*. To do this, when the compiler encounters a variable name, it tries to find the variable first, and if that fails, tries to find *@.variable*. Additionally, the normal rules of member lookup apply, so if variable is not a variable in scope and *@.variable* is not defined, *variable(@)*, a function call, is tried. A consequence of is that members may be shadowed by variables declared in an outer scope. This was a design decision to allow unambiguous referencing of all variables, as the member can always be referred to unambiguously as *.member* in a predicate. This additional syntax does not work if the member access is ambiguous because it is on a compound type where multiple component types have a member of that name defined. Additionally, there is no *type.member* syntax to unambiguously refer to a member of the default variable; *.type.member* must be used.

3.5 Casting

3.5.1 Primitive Casting

new-type: expr

This syntax returns the result of *expr* interpreted as type *new-type*. *int* and *float* types can can be casted from one to the other. Any decimal part of a *float* will be truncated. Any type can be converted to the

string type. It is possible to convert a *string* to an *int*, *float* or *bool*. However, if the string is not of the proper format, there will be a runtime error.

3.5.2 Table Casting

table: record-expr

This syntax returns a table with the same type as the record returned by *record-expr* with the that record as its only entry.

3.6 Statements

3.6.1 Expression statements

Most statements are expression statements, which consist of a single expression (which may be empty) followed by a newline. Any side effects of the expression are completed before the next statement is executed.

3.6.2 Compound statements

So that several statements can be used where one is expected, the compound statement (also known as a "block"; see the section on blocks in RT below) is provided. A compound statement is simply comprised of a list of newline-separated statements, with curly braces enclosing the entire list:

{*stmt_list*}

where (insert grammar rule for statement lists here).

The body of a function definition is a compound statement.

3.6.3 If-else statements

If-else statements choose one of several flows of control:

`if` (*expr*) *stmt*
`if` (*expr*) `else` *stmt*

In both forms of the `if` statement, the expression, which must have arithmetic or boolean type, is evaluated, including all side effects, and if it compares unequal to 0 or `false`, then the first substatement is executed. In the `if-else` statement, the second substatement is executed if the expression compares equal to 0 or `false`.

3.6.4 Iteration statements

Iteration statements specify looping. RT has two looping constructs: the `while` loop and the `for` loop.

do...while

```
do stmt while(expr)
do while(expr) stmt
```

In the `do... while` statement, the *stmt* is executed so long as the value of the *expr* remains unequal `false`. In the first example above, the test, including all side-effects from the expression, occurs after each execution of *stmt*. In the second example, the test, including all side-effects from the expression, occurs before each execution of *stmt*.

for

`do`(*expression1; expression2; expression3*) *statement*

In the `for` statement, the first expression is evaluated once, and thus specifies initialization for the loop. There is no restriction on its type. The second expression must have arithmetic or boolean type; it is evaluated before each iteration, and if it becomes unequal to 0 or `false`, the `for` is terminated. The third expression is evaluated after each iteration, and thus specifies a re-initialization for the loop. There is no restriction on its type. Side-effects from each expression are completed immediately after its evaluation. the statement

`for`(*expression1; expression2; expression3*) *statement*

is equivalent to

```
expression1
while (expression2) {
    statement
    expression3
}
```

Any of the three expressions may be dropped. A missing second expression makes the test equivalent to testing a non-zero, non-false element.

3.7 Functions

3.7.1 Function Definitions

Functions are declared as *fsig fbody*. An *fsig* is specified as either of:

rtype fname(formals)
rtype ootype.fname(formals)

where *rtype* is the RT type the function will return, *fname* is the name of the function, and formals are a comma separated list of *vdecls* which declare the argument names and types. In the second form where a *ootype* is given, an argument is prepended to formals of type *ootype* and of name "self".

An *fbody* is either a *block* or = *expr NEWLINE*. In the second form, function declarations are 1 line, and are equivalent to

```
fsig {
  return expr
}
```

Functions may not be named the same as other functions, including built in functions.

3.7.2 Function Calls

Function calls may be of the form *fn(args)* or *obj.fn(args)*. *fn* is the name of the function to be called, and args are the arguments to pass, as comma separated expressions. If *obj* is given, then *obj* is prepended as the first argument. (*obj* is an expression of any type.) Therefore, *a.fn()* is the same as *fn(a)*. Arguments must be of the same type and in the same order as the arguments declared in the function declaration. Functions can be called any where from within the RT program, regardless of where they are declared.

A function which returns void may omit the return statement.

3.8 Scope

3.8.1 Namespaces

Identifiers in RT fall into two namespaces which do not interfere with each other: in addition to the global namespace, every user-defined type creates a separate namespace for its members, so that the same name may appear in several different user-defined types.

3.8.2 Blocks

Blocks in RT are delimited with curly braces and are required as the bodies of function declarations, if statements, loop constructs, and pipe iteration; they can also be declared as wished by the programmer. Any variable declaration inside a block must be initialized. There is an implicit outer block containing the entire program.

3.8.3 Lexical scope of identifiers

The lexical scope of an object or function begins at the end of its declarator and persists to the end of the block in which it appears. The scope of a parameter of a function begins at the start of the block defining the function and persists to the end of the block. The scope of a user-defined type name begins at its appearance in a type specifier and persists to the end of the block.

3.9 Standard Library

3.9.1 print

print(expr)
print()

The print function outputs the results of *(*expr) to the console. Print with no argument puts a newline to the terminal.

3.9.2 stdin

stdin()

The stdin function gets keyboard input from the user and returns the input as a *string*

3.9.3 setWorkingDir

setWorkingDir(string-expr)
setWorkingDir()

The setWorkingDir function changes the path where the program will load or commit tables from and to. Calling setWorkingDir with no argument changes the working directory to original working directory when the program was called.

3.9.4 system

system(string-expr)

The system function executes the command returned by *string-expr*.

3.9.5 argc and argv

argc()
argv(int-expr)

Argc returns an integer equal to number of command line arguments passed to the program. Argv returns the string of the command line argument at the index returned by *(*int-expr).

3.9.6 Table Functions

load

load(type-expr, string-expr)

The load function loads data the of the type returned by *type-expr* in the file named in *string-expr* and returns a table with that information.

commit

commit(table-expr, string-expr)

The commit function stores the table returned by *table-expr* in the file named in *string-expr* in the comma separated value format. If the commit was successful, the table committed is returned.

th

(int-expr).th(table-expr)

The th function returns the record at index equal to *int-expr* from the table returned by *table-expr*. A *null* is returned if the there was no record at the given index.

size

table-expr.size()

The size function returns number of rows in the table returned by *table-expr*.

delete

(table-expr).delete(record-expr)

The delete function removes the record returned by *record-expr* from the table returned by *table-expr*.

tl

(table-expr).tl()

The tl function returns the tail the table returned by *table-expr*.

3.9.7 String Functions

strlen

(string-expr).strlen()

The strlen function returns the length of the string returned by *string-expr*.

substring

(string-expr).substring(int-expr)
(string-expr).substring(int-expr1, int-expr2)

The substring function with one argument returns the substring of the string returned by *string-expr* starting at index equal to *int-expr*. With two arguments, the substring returned starts at *int-expr1* and ending at *int-expr2*.

charAt

(string-expr).chatAt(int-expr))

The charAt function returns a string of a single character that was at index *int-expr* from the string returned by *string-expr*.

Chapter 4

Project Plan

4.1 Team Responsibilities

We each took responsibility for approximately half of the project. Jared was primarily responsible for the scanner, parser and code generation. Michael was primarily responsible for the Java libraries, test suite and documentation. However, since we were a two man team, there was significant overlap and we each had a hand in writing every element of the project. Therefore, we have both signed all code files in the project.

4.2 Programming Conventions

4.2.1 Java

The benefit of compiling to Java is that we could make use of the object-oriented features to limit the amount of redundant code that needed to be written. The java libraries provide all of the functionality for the RTs record and table objects. Since records are viewed by the language as singleton tables, we were able to write Record.java as a subclass of Table.java. Similarly, we were able to take this approach for UDTs and Compound Types. Similarly, we were able to take advantage of Javas interface mechanism to standardize predicates and map statements. As a result, the Java source generated by the RT compiler is concise and consists primarily of function calls to the RT libraries.

4.2.2 OCaml

OCaml code with meaningful variable and function names is typically very readable. Thus, throughout the project, commenting is limited to wherever it is completely necessary. We also attempted to write wrapper functions for more complex function calls so that their meaning and intent would be more clear to the reader.

4.2.3 Time Line

Sept. 28 La Mesa original proposal handed in
Oct. 26 La Mesa original LRM handed in
Nov. 1 La Mesa original scanner and parser started
Dec. 19 Code generation started
Dec. 22 Original due date, 75% of original La Mesa functionality implemented
Dec. 24 Original La Mesa functionality completed

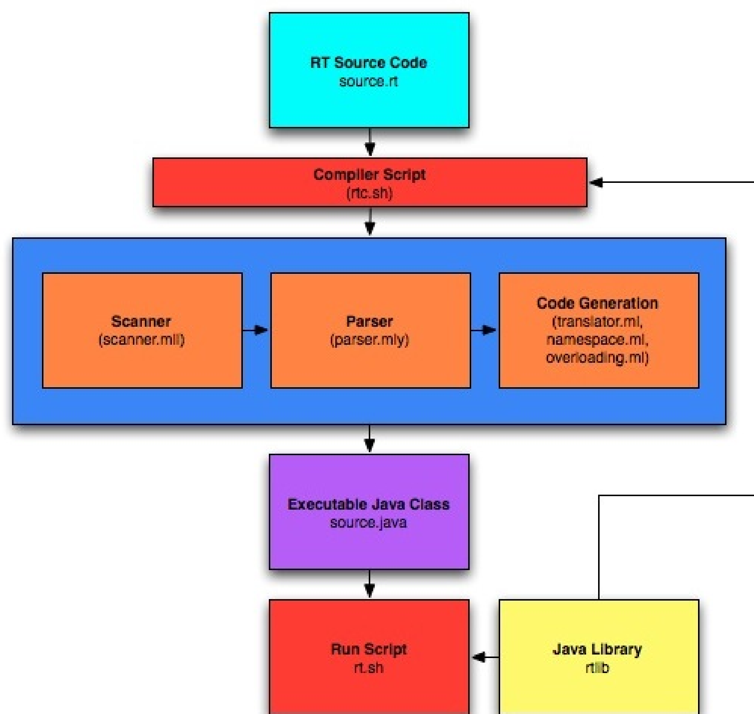
Jan. 2 Full RT functionality completed

Jan. 4 Documentation completed

Chapter 5

Architectural Design

5.1 Block Diagram



5.2 Command Line Interface

To compile an RT program the following command is run:

```
./rtc.sh <source-file> <optional destination> <optional debug>
```

Running the rtc.sh script requires that a source-file is given. If no destination is provided, the compiled .class is placed in the present working directory. If the debug flag is raised, the intermediate .java file is also placed into either the present working directory or the destination depending on whether one is provided. To run the compiled .class, the following command is executed:

```
./rt.sh <class-file> <optional args ... >
```

Running the rt.sh script links the compiled .class file against the RtLib and runs executes it. Command line arguments can be passed to the program by appending them to the end of the call to the script.

Chapter 6

Test Plan

6.1 Example Test Programs

6.1.1 Map, Filter, Join

RT Source:

```
/*
 * map_filter_join.rt
*/


type User{
        string name
        int age
}

User table users = table: User("Michael", 21)
users = users +  User("Jared", 16)

print(users)
print()

type Product{
        string itemName
        float cost
}

Product table products = table: Product("Soap", 1.99)
products = products +  Product("Chocolate", 16.0)

type Purchased{
        string cName
        string pName
}

print(products)
print()

User table michael = users[.name:"Michael"] // filter

print(michael)
print()

User#Product table users_products = users[true]products // join

print(users_products)
print()
```

```
42
43  // single table map
44  Purchased table purchases1 = users_products[Purchased(.name, .itemName)]
45
46  print(purchases1)
47  print()
48
49  // double table map, produces the same result as purchases1
50  Purchased table purchases2 = users[Purchased(name, itemName)]products
51
52  print(purchases2)
53  print()
```

Java Source

```
1
2   /*
3       Generated by the Return of the Table compiler.
4    */
5
6
7   import rtlib.*;
8   import java.util.Scanner;
9
10  class map_filter_join_anonymous_functor_1 implements FilterPredicate {
11
12
13  map_filter_join_anonymous_functor_1(){}
14  public boolean test(Record record) {
15                  return ((((record).entries[0]._String).equals(("Michael"))));
16              }}
17  class map_filter_join_anonymous_functor_2 implements JoinPredicate {
18
19
20  map_filter_join_anonymous_functor_2(){}
21  public boolean test(Record record, Record a, Record b) {
22                  return ((true));
23              }}
24  class map_filter_join_anonymous_functor_3 implements FilterMap {
25
26
27  map_filter_join_anonymous_functor_3(){}
28  public Record map(Record record) {
29                  return ((new Record(map_filter_join.rt2, new Data(((record).entries[0]._String)), new Data(((record).entries[2].
30              }public UserDefinedType getType() {
31                      return map_filter_join.rt2;
32                  }
33  }
34  class map_filter_join_anonymous_functor_4 implements JoinMap {
35
36
37  map_filter_join_anonymous_functor_4(){}
38  public Record map(Record record, Record a, Record b) {
39                  return ((new Record(map_filter_join.rt2, new Data(((record).entries[0]._String)), new Data(((record).entries[2].
40              }public UserDefinedType getType() {
41                      return map_filter_join.rt2;
42                  }
43  }
44
45
46  public class map_filter_join {
47      static UserDefinedType rt0;
48  static UserDefinedType rt1;
49  static UserDefinedType rt2;
50  static Table rt3;
51  static Table rt4;
52  static Table rt5;
53  static Table rt6;
```

```
54 static Table rt7;
55 static Table rt8;
56
57
58     private static Scanner input;
59                 static String globArgs[];
60
61         public static void main(String args[]) {
62         input = new Scanner(System.in);
63                                 globArgs = new String[args.length];
64                                 for(int globArgCounter = 0; globArgCounter < args.length; globArgCounter++)
65                                         globArgs[globArgCounter] = args[globArgCounter];
66
67         map_filter_join.rt0 = new UserDefinedType("User");
68 map_filter_join.rt0.add("User's name", Data.STRING_TYPE);
69 map_filter_join.rt0.add("User's age", Data.INT_TYPE);
70
71 map_filter_join.rt1 = new UserDefinedType("Product");
72 map_filter_join.rt1.add("Product's itemName", Data.STRING_TYPE);
73 map_filter_join.rt1.add("Product's cost", Data.FLOAT_TYPE);
74
75 map_filter_join.rt2 = new UserDefinedType("Purchased");
76 map_filter_join.rt2.add("Purchased's cName", Data.STRING_TYPE);
77 map_filter_join.rt2.add("Purchased's pName", Data.STRING_TYPE);
78
79 {
80 (rt3)=(new Table((new Record(map_filter_join.rt0, new Data(("Michael")), new Data((21))))));
81 (rt3)=((rt3).addRow((new Record(map_filter_join.rt0, new Data(("Jared")), new Data((16))))));
82 System.out.println((rt3));
83 System.out.println("");
84 (rt4)=(new Table((new Record(map_filter_join.rt1, new Data(("Soap")), new Data((1.99))))));
85 (rt4)=((rt4).addRow((new Record(map_filter_join.rt1, new Data(("Chocolate")), new Data((16.))))));
86 System.out.println((rt4));
87 System.out.println("");
88 (rt5)=((rt3).filter(new map_filter_join_anonymous_functor_1()));
89 System.out.println((rt5));
90 System.out.println("");
91 (rt6)=((rt3).join((rt4), new map_filter_join_anonymous_functor_2()));
92 System.out.println((rt6));
93 System.out.println("");
94 (rt7)=((rt6).filterMap(new map_filter_join_anonymous_functor_3()));
95 System.out.println((rt7));
96 System.out.println("");
97 (rt8)=((rt3).joinMap((rt4), new map_filter_join_anonymous_functor_4()));
98 System.out.println((rt8));
99 System.out.println("");
100
101 }
102     }
103 }
104
```

6.1.2 User Purchases

RT Source

```
1  /*
2   * user_purchases.rt
3   */
4
5  //define column names for the user type defined by user.csv
6  //the columns are defined sequentially in left-to-right order
7  type User {
8    string name
9    string addr
10   int age
11 }
```

```
12
13  //ourUsers is a table of users with the data from ourUsers.csv
14  User table ourUsers
15
16  //similar to user
17  type Purchase {
18    string name
19    string item
20  }
21
22  //ourPurchases is a table of users with the data from ourPurchases.csv
23  Purchase table ourPurchases
24
25  //global int
26  int v = 0
27
28  type index {
29      int value
30  }
31
32  void changev() {
33    v = 15
34  }
35
36  void main() {
37    //change global v
38    v = 2
39
40    //user#purchase is obtained by joining a table of type user and a table of
41    //type purchase.
42    //it takes the cross product of ourUsers and ourPurchases and filters
43    //it by the boolean predicate in the brackets
44
45    User#Purchase table user_purchase = ourUsers[a.name:b.name]ourPurchases
46    print(user_purchase)
47
48    //this is a simple filter.  get all rows where users purchased a tank.
49    User#Purchase table tanks = user_purchase[.Purchase.item:"Tank"]
50
51    print("\ncasting to component types:")
52    print(User(tanks))
53
54    print("\nmanual casting:")
55    User table users_who_purchased_tanks = table: User
56    tanks | tank: users_who_purchased_tanks += User(tank.User.name, tank.addr, tank.age)
57    print( users_who_purchased_tanks )
58
59
60    print("\nmanual table print (via iteration)")
61    //pipe iteration
62    tanks | tank: print(tank)
63
64
65    print("\nindexing table...")
66    index#User#Purchase table indexedtanks = index|+|tanks
67    int i = 0
68    tanks|tank {
69      indexedtanks = indexedtanks + index(i)|+|tank
70      i = i + 1
71    }
72    print(indexedtanks)
73
74    //add one more user to our table of users
75    ourUsers = ourUsers + User("Matt", "New York", 23)
76
77    print("\nchecking global variables...")
78    changev()
79    print("$v should be 15... did it work? ${v == 15}!") //should print 15
```

```
80  }
81
82  main()
```

Java Source

```
1
2   /*
3       Generated by the Return of the Table compiler.
4    */
5
6
7   import rtlib.*;
8   import java.util.Scanner;
9
10  class user_purchases_anonymous_functor_1 implements JoinPredicate {
11
12
13  user_purchases_anonymous_functor_1(){}
14  public boolean test(Record record, Record a, Record b) {
15                  return ((((a).entries[0]._String).equals(((b).entries[0]._String))));
16              }}
17  class user_purchases_anonymous_functor_2 implements FilterPredicate {
18
19
20  user_purchases_anonymous_functor_2(){}
21  public boolean test(Record record) {
22                  return ((((record).entries[4]._String).equals(("Tank"))));
23              }}
24  class user_purchases_anonymous_functor_3 implements JoinPredicate {
25
26
27  user_purchases_anonymous_functor_3(){}
28  public boolean test(Record record, Record a, Record b) {
29                  return ((true));
30              }}
31  class user_purchases_anonymous_functor_4 implements JoinPredicate {
32
33
34  user_purchases_anonymous_functor_4(){}
35  public boolean test(Record record, Record a, Record b) {
36                  return ((true));
37              }}
38
39
40  public class user_purchases {
41      static UserDefinedType rt0;
42  static UserDefinedType rt1;
43  static UserDefinedType rt2;
44  static Table rt3;
45  static Table rt4;
46  static int rt5;
47  public static void rt7() {
48  {
49  (rt5)=(2);
50  Table rt8;
51  (rt8)=((rt3).join((rt4), new user_purchases_anonymous_functor_1()));
52  System.out.println((rt8));
53  Table rt9;
54  (rt9)=((rt8).filter(new user_purchases_anonymous_functor_2()));
55  System.out.println((("\n")+("casting to component types:")));
56  System.out.println(((rt9).getReg(user_purchases.rt0)));
57  System.out.println((("\n")+("manual casting:")));
58  Table rt10;
59  (rt10)=(new Table(user_purchases.rt0));
60  for (Record rt11 : (rt9)){(rt10).addRow((new Record(user_purchases.rt0, new Data(((rt11).entries[0]._String)), new Data(((rt11).
61  System.out.println((rt10));
62  System.out.println((("\n")+("manual table print (via iteration)")));
```

```
for (Record rt11 : (rt9)){System.out.println((rt11));}
System.out.println((("\n")+("indexing table...")));
Table rt11;
(rt11)=((new Table(user_purchases.rt2)).join(((rt9)), new user_purchases_anonymous_functor_3()));
int rt12;
(rt12)=(0);
for (Record rt13 : (rt9)){{
(rt11)=((rt11).append(((new Table((new Record(user_purchases.rt2, new Data((rt12)))))).join((new Table((rt13))), new user_purcha
(rt12)=((rt12)+(1));

}}
System.out.println((rt11));
(rt3)=((rt3).addRow((new Record(user_purchases.rt0, new Data(("Matt")), new Data(("New York")), new Data((23))))));
System.out.println((("\n")+("checking global variables...")));
user_purchases.rt6();
System.out.println((((((""+(rt5))+(" should be 15... did it work? "))+(""+((rt5)==(15))))+("!")));

}}

public static void rt6() {
{
(rt5)=(15);

}}

    private static Scanner input;
                static String globArgs[];

        public static void main(String args[]) {
        input = new Scanner(System.in);
                                globArgs = new String[args.length];
                                for(int globArgCounter = 0; globArgCounter < args.length; globArgCounter++)
                                        globArgs[globArgCounter] = args[globArgCounter];

        user_purchases.rt0 = new UserDefinedType("User");
user_purchases.rt0.add("User's name", Data.STRING_TYPE);
user_purchases.rt0.add("User's addr", Data.STRING_TYPE);
user_purchases.rt0.add("User's age", Data.INT_TYPE);

user_purchases.rt1 = new UserDefinedType("Purchase");
user_purchases.rt1.add("Purchase's name", Data.STRING_TYPE);
user_purchases.rt1.add("Purchase's item", Data.STRING_TYPE);

user_purchases.rt2 = new UserDefinedType("index");
user_purchases.rt2.add("index's value", Data.INT_TYPE);

{
(rt3)=(CsvInterpreter.toTable(WorkingDir.getPath(("ourUsers.csv")), user_purchases.rt0));
(rt4)=(CsvInterpreter.toTable(WorkingDir.getPath(("ourPurchases.csv")), user_purchases.rt1));
(rt5)=(0);
user_purchases.rt7();

}
    }
}

```

6.1.3 String Interpolation

RT Source

```
/*
 * string_concat.lm
 */

print("hello
        something\n\"qouted\"\\‘’
```

```
        4 + 5 = ${4 + 5}!
        some more stuff
        ")

string s = ""
int i = 3
do {
        print("${s+="hello, "}i is $i \n")
} while (i < 10) {
        i = i + 1
}
```

Java Source

```

/*
    Generated by the Return of the Table compiler.
 */


import rtlib.*;
import java.util.Scanner;



public class stringinterp {
    static String rt0;
static int rt1;


    private static Scanner input;
                static String globArgs[];

        public static void main(String args[]) {
        input = new Scanner(System.in);
                                globArgs = new String[args.length];
                                for(int globArgCounter = 0; globArgCounter < args.length; globArgCounter++)
                                        globArgs[globArgCounter] = args[globArgCounter];

        {
System.out.println((((((((("hello\n\tsomething")+("\n"))+("\""))+("qouted"))+("\""))+("\\"))+("‘’\n\t4 + 5 = "))+(""+((4)+(5)))
(rt0)=("");
(rt1)=(3);
 while (true) {{
System.out.println(((((((rt0)+=("hello, ")))+("i is "))+(""+(rt1)))+(" "))+("\n")));

} if(!((rt1)<(10))) { break; } {
(rt1)=((rt1)+(1));

}}

}
    }
}

```

6.2 Test Selection

The goal in our test selection process was to ensure coverage of all of the features of our language. While these programs individually do not illustrate the language's full power or produce useful results, they demonstrate that the translator generates the correct code.

6.3 Test Suite

6.3.1 individualTest.sh

```bash
#!/bin/bash

# individualTest.sh

cd "$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

f=$1

RT=".rt"


compile="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )/../compiler/rtc.sh"
run="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )/../compiler/rt.sh"

log="/tmp/RTCLogTemp${f%\/}.txt"


correct="output/correct.txt"
trial="./output/trial.txt"


cd "$f"
if [ ! -e "testScript.sh" ]
        then
        filename=`ls *.rt`
        file=${filename%$RT}
        echo "Testing $file" > "$log"
        "$compile" "$filename" ./build/ debug 1>/dev/null 2>>"$log"
        if [ -e build/"$file".class ]
                then
                "$run" ./build/"$file".class > $trial 2>/dev/null
                 diff -s -q $trial ./$correct >> "$log"
                echo >> "$log"
        else
                echo "$file did not compile" >> "$log"
                echo >> "$log"
        fi
else
        ./testScript.sh >> "$log"
fi
cd ..
```

6.3.2 testAll.sh

```bash
#!/bin/bash

#  testAll.sh

me="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"

compile="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )/../compiler/rtc.sh"
run="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )/../compiler/rt.sh"

cd "$me"

dir=`pwd`
log="$dir/log.txt"

correct="output/correct.txt"
trial="./output/trial.txt"


dateStr=`date`
```

```
20  echo "Test Results from $dateStr" > "$log"
21
22  RT=".rt"
23
24  t=‘ls -d */‘
25  IFS=$’\n’
26
27  maxjob=4
28
29  for f in $t
30  do
31          bash ./individualTest.sh $f &
32  done
33
34  wait
35
36  for i in ‘ls /tmp/RTCLogTemp*.txt‘
37  do
38          cat "$i" >> "$log"
39  done
40
41  rm -f /tmp/RTCLogTemp*.txt
42
43  cat "$log"
```

6.3.3 testClean.sh

```
1   #!/bin/bash
2
3   #  testClean.sh
4
5   cd "$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
6
7
8   t=‘ls -d */‘
9   IFS=$’\n’
10
11  rm -f log.txt
12
13  for f in $t
14  do
15          cd "$f"
16          cd build
17          rm -f *.class
18          rm -f *.java
19          cd ..
20          cd output
21          rm -f trial.txt
22          cd ..
23          cd ..
24
25  done
26
27  rm -f /tmp/RTCLogTemp*.txt
```

6.3.4 Automation

The test process is completely automated. Running the testAll.sh script results in each of our 18 test cases being compiled, run and the output compared against the verified output. Each test is run in parallel to improve the speed of the testing process. The results of the comparison against verified output is stored in log.txt and are outputted to the terminal. Additionally, if a different test mechanism is needed for a particular test case, testAll.sh will look for testScript.sh in the directory where the test resides. If testScript.sh exists, then the results are recorded in log.txt.

Chapter 7

Lessons Learned

7.1 Michael's Lessons

Beyond being an exercise in writing a compiler, this project was the first opportunity for us, as students, to work on the development of a large scale software project in a medium sized group. The most important lesson that I learned from the project related both to compiler writing specifically and also team software development in general. It is of the utmost importance that groups working on a large project create a specification which they want their product to adhere to and stick to it, with as little variation as possible. From the compilers point of view, this is important because it forces the team to think before they act and carefully plan out the languages desired functions and syntax. A specification that is well thought out will both be more useful to the end user and easier for the team to implement. One of the largest difficulties that the team faced was our inability to settle on a specification, both a set of functions and a syntax that represented what we wanted to do with La Mesa. As a result, we spent far too much time discussing what our language would look like rather than implementing functionality. The other aspect of specifying our language that our group could have done a better job with was having a better understanding of the ways in which the different aspects of the project fit together. For example, one of the areas which I was responsible for were the Java libraries that our language would use to generate the Table functionality that our language provided. Due to poor communication amongst the group, I had to rewrite the libraries twice to adhere to the way in which code would be generated. This was time that could have been better spent implementing other core features in the compiler.
In addition, after forking the project, it became clear that the biggest lesson that I've learned while doing this project is the importance of working with people that you both trust to get something done and respect them enough to let them do it. Although we were only half the size of the original group, Jared and I were much more efficient than when working with the larger group because we were able to divide the work and trust that the other would get his part done correctly and on time.

7.2 Jared's Lessons

I learned a lot about working on a group software project in this class, and it was exciting to have the opportunity to work on a difficult, college-level project. Like the others, I wish that we had done more coding earlier in the semester and not put off so many decisions until later. On a positive note, I really enjoyed learning OCaml and plan to continue with functional programming.

I learned especially how important it should have been to have a coherent vision for a language. Without it, we had a difficult time figuring out the details of the language, as we didn't really have an idea of what we wanted it to do.

I learned that making compilers in ocaml, while fun, generally is a fairly standard procedure. Once we had a good AST and translator, adding new features was mostly a matter of adding to the parser, and things that would support the parser, including whatever code was necessary to generate the java. For example, the majority of the AST could be represented as function calls, including casts, unary, and binary operations. Therefore, with a the function translation implemented, most functionality could have been implemented in the parser and built in functions. I very much appreciated this because of the ability to very easily add features in very little time by focusing work on the parser, which was actually a bit addicting.

7.3 Recommendations to Future Groups

Start Early!
Don't be afraid to nail down your language specifications before starting your scanner and parser
Don't just choose the people next to you when looking for a group, find people you will be happy working with
As you get closer to completion, pick an endpoint and stick to it
Become very comfortable with your version control system and commit often to avoid conflicts

Chapter 8

Code Listing

8.1 Translator

8.1.1 ast.ml

```
(*Michael Vitrano and Jared Pochtar*)

type op = Assign
        | Add | Sub | Mult | Div
        | Equal | Neq | Less | Leq | Greater | Geq
        | And | Or | Qmark
        | AddAsn | SubAsn | MultAsn | DivAsn | AndAsn | OrAsn


type t =
    Int
  | Float
  | String
  | Bool
  | Void
  | Udt of string
  | Tuple of string list

type var_decl = {
    vname : string;
    vtype : t;
    isTable : bool;
  }

type type_decl = {
    tname : string;
    members : var_decl list
  }

type col_access_id = string option * string

type expr =
    Id of string
  | Binop of expr * op * expr
  | Attr of expr * col_access_id
  | CurrentVar
  | Call of string * expr list
  | Cast of t * expr
  | CastToTable of expr
  | First of expr
  | Noexpr
  | StringLiteral of string
  | IntLiteral of int
```

```
44    | BoolLiteral of bool
45    | FPLiteral of float
46    | Null of t * bool
47    | Join of expr * expr * expr
48    | Filter of expr * expr
49
50
51  type stmt =
52      Block of block_line list
53    | Expr of expr
54    | Return of expr
55    | If of expr * stmt * stmt
56    | For of expr * expr * expr * stmt
57    | DoWhile of stmt * expr * stmt
58    | Iter of string * expr * stmt
59  and block_line =
60      Statement_line of stmt
61    | Vdecl_line of var_decl
62
63  type func_decl = {
64      ret_table : bool;
65      rettype : t;
66      fname : string;
67      formals : var_decl list;
68      body : stmt;
69    }
70
71  type program = {
72      global_vars : var_decl list;
73      functions : func_decl list;
74      types : type_decl list;
75      global_code : stmt list
76    }
77
78  type table_literal = expr list list
```

8.1.2 main.ml

```
1   (*Michael Vitrano and Jared Pochtar*)
2
3   open Pp
4   open Ast
5
6   type compiler_options = {
7       debug_parser : bool;
8       main_class_name : string;
9     }
10
11  let default_options = {
12      debug_parser = false;
13      main_class_name = "Main"
14    }
15
16  let options_from_flags flags =
17      let rec options last = function
18            [] -> last
19          | "-showparse"::rs ->
20              options { last with debug_parser = true } rs
21          | "-mainclass"::mainclassname::rs ->
22              options { last with main_class_name = mainclassname } rs
23          | _::rs -> options last rs
24      in options default_options flags
25
26  let _ =
27    let lexbuf = Lexing.from_channel stdin in
28    let program = Parser.program (Scanner.buffer_tokens Scanner.token) lexbuf in
29
```

```
30    let compiler_parameters = options_from_flags (Array.to_list Sys.argv) in
31
32    if compiler_parameters.debug_parser then
33      print_endline (Pp.string_of_program program)
34    else
35      print_endline (Translator.java_of_program compiler_parameters.main_class_name program)
```

8.1.3 namespace.ml

```
1   (*Michael Vitrano and Jared Pochtar*)
2
3
4   open Ast
5
6   exception Error of string
7
8   type lm_java_name = string * string (* lmname, jname *)
9
10  type lmprimitive =
11            Int
12          | Float
13          | String
14          | Bool
15
16  type colname = string * string (* first str is typename *)
17  type lmudt = lm_java_name * (colname * lmprimitive) list
18  type lmtuple = lmudt list
19
20  type lmobjecttype =
21            UDT of lmudt
22          | TupleUDT of lmtuple
23
24  type lmtype =
25            Primitive of lmprimitive
26          | Singleton of lmobjecttype
27          | Table of lmobjecttype
28          | Void
29
30  type lmvar = lm_java_name * lmtype
31  type lmfuncsig = lm_java_name * lmtype * lmtype list
32
33  type nsobject =
34            NSVar of lmvar
35          | NSFunc of lmfuncsig
36          | NSUDT of lmudt
37
38  type namespace = {
39          super_scope : namespace option;
40      objects : (string * nsobject) list;
41
42      next_id_index : int;
43      main_class_name : string;
44
45      returntype : lmtype option;
46      default_variable : (string * lmtype) option;    (* for .col in a filter *)
47
48      captures : nsobject list ref;       (* for capuring objs/func captures in filters/joins *)
49      isCapturing : bool
50  }
51
52  let next_id ns = "rt" ^ (string_of_int ns.next_id_index),
53                   { ns with next_id_index = ns.next_id_index + 1 }
54
55  let new_scope mainclass =
56                  { super_scope = None; objects = [];
57                    next_id_index = 0; main_class_name = mainclass;
58                    returntype = None; default_variable = None;
```

```
59                   captures = ref []; isCapturing = false }
60
61  let nest_scope scope = { scope with
62      super_scope = Some(scope);
63      objects = [];
64
65      isCapturing = false;
66      captures = ref [];
67  }
68
69
70
71                 (* ocaml utils *)
72
73  let qt str = "\"" ^ str ^ "\""
74
75  let list_of_unique_elements l =
76      let rec uniq accum = function
77            [] -> accum
78          | e::rs -> uniq (if List.exists (fun x -> x = e) accum then accum else e::accum) rs
79      in List.rev (uniq [] l)
80
81  let rec list_elements_are_unique = function
82        [] -> true
83      | e::rs ->
84          if (List.mem e rs) then false else
85          list_elements_are_unique rs
86
87  let index_of fn list =  (* returns (index, element, isunique) option *)
88      let rec index fn i = function
89            [] -> None
90          | e::r ->
91                  if fn(e) then
92                      Some(i, e, (index fn 0 r) = None)
93                  else
94                      index fn (i + 1) r
95      in index fn 0 list
96
97
98
99                 (* java signatures *)
100
101 let java_type_of_primitive_type = function
102       Int -> "int"
103     | Float -> "double"
104     | String -> "String"
105     | Bool -> "boolean"
106
107 let java_signature = function
108           Primitive(p) -> java_type_of_primitive_type p
109         | Singleton(_) -> "Record"
110         | Table(_) -> "Table"
111         | Void -> "void"
112
113
114                (* Scope functions *)
115
116 let rec flatten_scope ns =
117         let objs = ns.objects in
118         match ns.super_scope with
119                   None -> objs
120                 | Some(ns') -> objs @ flatten_scope ns'
121
122 let string_of_scope ns =
123         let flat = List.map fst (flatten_scope ns) in
124         "[" ^ (String.concat "; " flat) ^ "]"
125
126 (* get object *)
```

```
127 let rec obj_named ns id =       (* returns the obj, if it was captured *)
128         try
129                 let _, obj = List.find (fun (name, obj) -> name = id) ns.objects
130                 in Some(obj, false)
131         with Not_found -> match ns.super_scope with
132                   None -> None
133                 | Some(super) ->
134             if not ns.isCapturing
135             then obj_named super id
136             else match obj_named super id with
137                   None -> None
138                 | Some(ssobj, _) ->
139                     ns.captures := (ssobj)::!(ns.captures);
140                     Some(ssobj, true)
141
142 (* add object *)
143 let scope_with_obj_of_name_ignoring_shadowing scope obj name =
144     { scope with objects = (name, obj)::(scope.objects) }
145
146 let builtin_function_names = ["print"; "commit"; "th"; "size"; "stdin"; "setWorkingDir"; "substring"; "strlen"; "charAt"; "syste
147
148 let scope_with_obj_of_name scope obj (name : string) =
149         if not ((obj_named scope name) = None) then
150         raise (Error ("redeclaring " ^ name ^ " where there's " ^ (string_of_scope scope)))
151     else if (List.mem name builtin_function_names) then
152         raise (Error ("redeclaring built in function "^name))
153         else scope_with_obj_of_name_ignoring_shadowing scope obj name
154
155
156
157                 (* UDT functions: get index+type of column by name. *)
158
159 (* use the last one, col_index_of_obj with udt, col type qualifier option, colname *)
160
161 let col_index_of_tuple tuple col_on_type colname =
162         let allcols = List.flatten (List.map snd tuple) in
163     match index_of (fun ((col_on_type', colname'), _) ->
164         col_on_type = col_on_type' && colname = colname')
165         allcols
166         with
167           Some(index, (_, coltype), true) -> Some(index, coltype)
168     | Some(_, _, false) -> raise (Error "internal error: duplicate unambiguous member access")
169         | None -> None
170
171 let unqualified_col_index_of_tuple tuple colname =
172         let allcols = List.flatten (List.map snd tuple) in
173     match index_of
174         (fun ((_, colname'), _) -> colname = colname')
175         allcols
176         with
177           Some(index, (_, coltype), true) -> Some(index, coltype)
178     | Some(_, _, false) -> raise (Error "ambiguous member access")
179         | None -> None
180
181 let col_index_of_obj obj col_on_type colname =
182     let colindex = match obj, col_on_type with
183           UDT(x), Some(coltype) ->      col_index_of_tuple [x] coltype colname
184         | TupleUDT(x), Some(coltype) -> col_index_of_tuple x coltype colname
185         | UDT(x), None ->               unqualified_col_index_of_tuple [x] colname
186         | TupleUDT(x), None ->          unqualified_col_index_of_tuple x colname
187     in match colindex with
188           None -> raise (Error "accessing member that doesn't exist")
189         | Some(index) -> index
190
191 let combine_types lhs rhs =
192     let udt_list_of_lmobject = function
193           UDT(lmudt) -> [lmudt]
194         | TupleUDT(lmudts) -> lmudts
```

```
195     in
196     let combined =
197         (udt_list_of_lmobject lhs)@(udt_list_of_lmobject rhs)
198     in
199     TupleUDT(combined), list_elements_are_unique combined
200
201 let java_of_udt = function
202       UDT((_, jname), _) -> jname
203     | TupleUDT(udts) ->
204         let lm_java_udt_names = List.map fst udts in
205         let compound_name = String.concat "#" (List.map fst lm_java_udt_names) in
206         let java_of_udts = String.concat ", " (List.map snd lm_java_udt_names) in
207         "new TupleType("^qt(compound_name)^", "^java_of_udts^")"
208
209 (* NSUDT functions *)
210 (* get UDT *)
211 let udt_of_name scope udtname =
212         match obj_named scope udtname with
213                   None -> raise (Error ("no type named "^udtname))
214                 | Some(NSUDT(udt), _) -> udt
215                 | Some(_) -> raise (Error (udtname^" is not a type"))
216
217 (* exists *)
218 let exists_udt_named ns udtname =
219     match obj_named ns udtname with
220           None -> false
221         | Some(NSUDT(_), _) -> true
222         | Some(_) -> false
223
224
225                     (* type functions *)
226
227 (* AST TYPE -> LMTYPE *)
228 let lmtype_of_ast_type ns ast_type istable =
229     match ast_type with
230           Ast.Int -> Primitive(Int)
231         | Ast.Float -> Primitive(Float)
232         | Ast.String -> Primitive(String)
233         | Ast.Bool -> Primitive(Bool)
234         | Ast.Void -> Void
235         | Ast.Udt(name) ->
236             let udt = udt_of_name ns name in
237             if istable then Table(UDT(udt)) else Singleton(UDT(udt))
238         | Ast.Tuple(udt_names) ->
239             let udts = List.map (udt_of_name ns) udt_names in
240             if istable then Table(TupleUDT(udts)) else Singleton(TupleUDT(udts))
241
242
243
244
245                     (* NSUDT functions *)
246 (* add UDT *)
247 let scope_with_ast_udt_decl ns (udt_decl : Ast.type_decl) =
248     let member_of_ast_vdecl vdecl =
249         match (lmtype_of_ast_type ns vdecl.vtype vdecl.isTable) with
250               Primitive(p) -> ((udt_decl.tname, vdecl.vname), p)
251             | _ -> raise (Error "improper type in a UDT member")
252     in
253     let members = List.map member_of_ast_vdecl udt_decl.members in
254
255     if not (list_elements_are_unique (List.map fst members)) then
256         raise (Error "duplicate member in udt declaration")
257     else
258
259     let name, ns = next_id ns in
260         let udt = (udt_decl.tname, ns.main_class_name^"."^name), members in
261     (scope_with_obj_of_name ns (NSUDT(udt)) udt_decl.tname), udt, name
262
```

```
263
264
265
266                     (* NSVar functions *)
267 (* add var *)
268 let scope_with_var_name_type ns varname vartype =
269     if vartype = Void then raise (Error "variable cannot have type void") else
270     let name, ns = next_id ns in
271         let lmv = (varname, name), vartype in
272     (scope_with_obj_of_name ns (NSVar lmv) varname), lmv
273
274 let scope_with_jvar_passthrough scope varname vartype =
275     scope_with_obj_of_name_ignoring_shadowing
276         scope
277         (NSVar((varname, varname), vartype))
278         varname
279
280 let scope_with_var ns var =
281     let vartype = lmtype_of_ast_type ns var.vtype var.isTable in
282     scope_with_var_name_type ns var.vname vartype
283
284 (* get var *)
285 let var_of_name ns name =
286          match obj_named ns name with
287                   None -> raise (Error ("no var named "^name^" in scope "^(string_of_scope ns)))
288                 | Some(NSVar(v), was_captured) -> v, was_captured
289                 | Some(_) -> raise (Error (name^" is not a variable"))
290
291
292
293
294                     (* NSFunc functions *)
295
296 (* add fsig *)
297 let scope_with_ast_func ns func_decl =
298     let rettype = lmtype_of_ast_type ns func_decl.rettype func_decl.ret_table in
299     let formals = List.map
300         (fun v ->
301             let ft = lmtype_of_ast_type ns v.vtype v.isTable in
302             if ft = Void then raise (Error "function argument cannot be of type void")
303             else ft)
304         func_decl.formals
305     in
306     let name, ns = next_id ns in
307     let fsig = ((func_decl.fname, ns.main_class_name^"."^name), rettype, formals) in
308     (scope_with_obj_of_name ns (NSFunc fsig) func_decl.fname), fsig, name
309
310 (* get fsig *)
311 let funcsig_of_name ns name =
312          match obj_named ns name with
313                   None -> raise (Error ("no function named "^name^" in scope "^(string_of_scope ns)))
314                 | Some(NSFunc(f), _) -> f
315                 | Some(_) -> raise (Error (name^" is not a variable"))
316
317
318
319                     (* functor handling *)
320
321 let functor_declarations = ref ""
322 let functor_next_id = ref 0
323
324 let reset_functors u =
325     functor_declarations := "";
326     functor_next_id := 0
327
328 let get_new_functor_name ns =
329     functor_next_id := !functor_next_id + 1;
330     ns.main_class_name^"_anonymous_functor_" ^ (string_of_int !functor_next_id)
```

```
331
332 let add_functor ftor =
333     functor_declarations := (!functor_declarations ^ ftor)
334
335 let getCaptures capturing_scope =
336     List.fold_left (fun accum v -> match v with
337           NSVar((_, jname), vtype) -> (jname, (java_signature vtype))::accum
338         | _ -> accum
339     ) [] (list_of_unique_elements !(capturing_scope.captures))
340
341 let functor_call pred_capturing_scope test_java proto_name test_sig getType =
342     let captures = getCaptures pred_capturing_scope in
343     let ft_name = get_new_functor_name pred_capturing_scope in
344     let ftor =
345         "class "^ft_name^" implements "^proto_name^" {\n"
346             (* ivars = captured variables *)
347             ^ (String.concat "\n" (List.map
348                     (fun (jid, jsig) -> jsig^" "^jid^";")
349                     captures))
350             ^ "\n\n"
351
352             (* ctor to initialize captures *)
353             ^ ft_name^"("
354             ^ (String.concat ", " (List.map
355                     (fun (jid, jsig) -> jsig^" _"^jid)
356                     captures))
357             ^ "){"
358             ^ (String.concat "\n" (List.map
359                     (fun (jid, jsig) -> jid^" = _"^jid^";")
360                     captures))
361             ^ "}\n"
362
363             (* code captured *)
364             ^test_sig^" {
365                 return "^test_java^";
366             }"
367             ^ (match getType with None -> "" | Some(udt) ->
368                 "public UserDefinedType getType() {
369                     return "^java_of_udt udt^";
370                 }\n"
371             )
372             ^"}\n"
373         in add_functor ftor;
374
375     "new "^ft_name^"("^(String.concat ", " (List.map fst captures))^")"
376
377
```

8.1.4 overloading.ml

```
1  (*Michael Vitrano and Jared Pochtar*)
2
3  open Ast
4  open Namespace
5  open Namespace_pp
6  exception Error of string
7
8  (*
9     operator overloading goes here! first get the types of the subexprs,
10    then match on (lhstype, op, rhstype)
11    (Int, Add, Int) -> something
12    (Table, Add, Table) -> something different
13    and so on
14 *)
15
16 let requires_assignable assignable =
17     if not assignable then
```

```
18          raise (Error ("lhs of assign is not an lvalue"))
19      else ()
20
21  let java_of_binop ns isAssignable lhs_java rhs_java = function
22          | Primitive(String), Add, Primitive(String) ->
23                                lhs_java ^ "+" ^ rhs_java, Primitive(String)
24
25          | Primitive(Int), Add, Primitive(Int) -> lhs_java ^ "+" ^ rhs_java, Primitive(Int)
26          | Primitive(Float), Add, Primitive(Float)
27          | Primitive(Int), Add, Primitive(Float)
28          | Primitive(Float), Add, Primitive(Int) -> lhs_java ^ "+" ^ rhs_java, Primitive(Float)
29
30      | Table(ttype), Add, Singleton(rtype) when ttype=rtype ->
31                  lhs_java^".addRow("^rhs_java^")", Table(ttype)
32      | Table(ttype), Add, Table(rtype) when ttype=rtype ->
33                  lhs_java^".append("^rhs_java^")", Table(ttype)
34      | Singleton(ttype), Add, Table(rtype) when ttype=rtype ->
35                  rhs_java^".prepend("^lhs_java^")", Table(ttype)
36
37          | Primitive(Int), Sub, Primitive(Int) -> lhs_java ^ "-" ^ rhs_java, Primitive(Int)
38          | Primitive(Float), Sub, Primitive(Float)
39          | Primitive(Int), Sub, Primitive(Float)
40          | Primitive(Float), Sub, Primitive(Int) -> lhs_java ^ "-" ^ rhs_java, Primitive(Float)
41
42          | Primitive(Int), Mult, Primitive(Int) -> lhs_java ^ "*" ^ rhs_java, Primitive(Int)
43          | Primitive(Float), Mult, Primitive(Float)
44          | Primitive(Int), Mult, Primitive(Float)
45          | Primitive(Float), Mult, Primitive(Int) -> lhs_java ^ "*" ^ rhs_java, Primitive(Float)
46
47          | Primitive(Int), Div, Primitive(Int) -> lhs_java ^ "/" ^ rhs_java, Primitive(Int)
48          | Primitive(Float), Div, Primitive(Float)
49          | Primitive(Int), Div, Primitive(Float)
50          | Primitive(Float), Div, Primitive(Int) -> lhs_java ^ "/" ^ rhs_java, Primitive(Float)
51
52      | x, Assign, y when x=y -> requires_assignable isAssignable; lhs_java ^ "=" ^ rhs_java, x
53
54          | Primitive(String), AddAsn, Primitive(String) ->
55                  requires_assignable isAssignable;
56                                lhs_java ^ "+=" ^ rhs_java, Primitive(String)
57
58      | Table(ttype), AddAsn, Singleton(rtype) when ttype=rtype ->
59                  requires_assignable isAssignable;
60                  lhs_java^".addRow("^rhs_java^")", Table(ttype)
61      | Table(ttype), AddAsn, Table(rtype) when ttype=rtype ->
62                  requires_assignable isAssignable;
63                  lhs_java^".append("^rhs_java^")", Table(ttype)
64
65          | Primitive(Int), AddAsn, Primitive(Int) ->
66                  requires_assignable isAssignable;
67                  lhs_java ^ "+=" ^ rhs_java, Primitive(Int)
68          | Primitive(Float), AddAsn, Primitive(Float)
69          | Primitive(Int), AddAsn, Primitive(Float)
70          | Primitive(Float), AddAsn, Primitive(Int) ->
71                  requires_assignable isAssignable;
72                  lhs_java ^ "+=" ^ rhs_java, Primitive(Float)
73
74          | Primitive(Int), SubAsn, Primitive(Int) ->
75                  requires_assignable isAssignable;
76                  lhs_java ^ "-=" ^ rhs_java, Primitive(Int)
77          | Primitive(Float), SubAsn, Primitive(Float)
78          | Primitive(Int), SubAsn, Primitive(Float)
79          | Primitive(Float), SubAsn, Primitive(Int) ->
80                  requires_assignable isAssignable;
81                  lhs_java ^ "-=" ^ rhs_java, Primitive(Float)
82
83          | Primitive(Int), MultAsn, Primitive(Int) ->
84                  requires_assignable isAssignable;
85                  lhs_java ^ "*=" ^ rhs_java, Primitive(Int)
```

```
86          | Primitive(Float), MultAsn, Primitive(Float)
87          | Primitive(Int), MultAsn, Primitive(Float)
88          | Primitive(Float), MultAsn, Primitive(Int) ->
89                requires_assignable isAssignable;
90                lhs_java ^ "*=" ^ rhs_java, Primitive(Float)
91
92          | Primitive(Int), DivAsn, Primitive(Int) ->
93                requires_assignable isAssignable;
94                lhs_java ^ "/=" ^ rhs_java, Primitive(Int)
95          | Primitive(Float), DivAsn, Primitive(Float)
96          | Primitive(Int), DivAsn, Primitive(Float)
97          | Primitive(Float), DivAsn, Primitive(Int) ->
98                requires_assignable isAssignable;
99                lhs_java ^ "/=" ^ rhs_java, Primitive(Float)
100
101     | Primitive(Bool), AndAsn, Primitive(Bool) ->
102               requires_assignable isAssignable;
103               lhs_java ^ "&=" ^ rhs_java, Primitive(Bool)
104         | Primitive(Bool), OrAsn, Primitive(Bool) ->
105               requires_assignable isAssignable;
106               lhs_java ^ "|=" ^ rhs_java, Primitive(Bool)
107
108
109         | Primitive(Int), Equal, Primitive(Int)
110         | Primitive(Float), Equal, Primitive(Int)
111         | Primitive(Int), Equal, Primitive(Float)
112         | Primitive(Float), Equal, Primitive(Float)
113         | Primitive(Bool), Equal, Primitive(Bool)         -> lhs_java ^ "==" ^ rhs_java, Primitive(Bool)
114
115         | Primitive(String), Equal, Primitive(String) -> lhs_java ^ ".equals(" ^ rhs_java ^ ")", Primitive(Bool)
116
117
118         | Primitive(Int), Neq, Primitive(Int)
119         | Primitive(Float), Neq, Primitive(Int)
120         | Primitive(Int), Neq, Primitive(Float)
121         | Primitive(Float), Neq, Primitive(Float)
122         | Primitive(Bool), Neq, Primitive(Bool)         -> lhs_java ^ "!=" ^ rhs_java, Primitive(Bool)
123
124         | Primitive(String), Neq, Primitive(String) -> "!(" ^ lhs_java ^ ".equals(" ^ rhs_java ^ "))", Primitive(Bool)
125
126         | Primitive(Int), Less, Primitive(Int)
127         | Primitive(Float), Less, Primitive(Int)
128         | Primitive(Int), Less, Primitive(Float)
129         | Primitive(Float), Less, Primitive(Float) -> lhs_java ^ "<" ^ rhs_java, Primitive(Bool)
130
131         | Primitive(Int), Leq, Primitive(Int)
132         | Primitive(Float), Leq, Primitive(Int)
133         | Primitive(Int), Leq, Primitive(Float)
134         | Primitive(Float), Leq, Primitive(Float) -> lhs_java ^ "<=" ^ rhs_java, Primitive(Bool)
135
136         | Primitive(Int), Geq, Primitive(Int)
137         | Primitive(Float), Geq, Primitive(Int)
138         | Primitive(Int), Geq, Primitive(Float)
139         | Primitive(Float), Geq, Primitive(Float) -> lhs_java ^ ">=" ^ rhs_java, Primitive(Bool)
140
141         | Primitive(Int), Greater, Primitive(Int)
142         | Primitive(Float), Greater, Primitive(Int)
143         | Primitive(Int), Greater, Primitive(Float)
144         | Primitive(Float), Greater, Primitive(Float) -> lhs_java ^ ">" ^ rhs_java, Primitive(Bool)
145
146         | Primitive(Bool), And, Primitive(Bool) -> lhs_java ^ "&&" ^ rhs_java, Primitive(Bool)
147         | Primitive(Bool), Or, Primitive(Bool) -> lhs_java ^ "||" ^ rhs_java, Primitive(Bool)
148
149
150     | Primitive(Bool), Qmark, Singleton(udt) ->
151               lhs_java ^ " ? " ^ rhs_java ^ " : null", Singleton(udt)
152     | Primitive(Bool), Qmark, Table(udt) ->
153               lhs_java ^ " ? " ^ rhs_java ^ " : null", Table(udt)
```

```
154
155         | lhs, op, rhs -> raise (Error ("Type mismatch: "
156         ^ (string_of_lmtype lhs)
157         ^ " and "
158         ^ (string_of_lmtype rhs)
159         ))
160
161 let java_of_function_call ns fn args = match (fn, args) with
162
163       "print", [(printable_java, _)] ->
164             "System.out.println("^printable_java^")", Void
165                 | "print", [] -> "System.out.println(\"\")", Void
166     | "print", _ -> raise (Error "Print needs exactly one argument")
167
168     | "commit", [(object_java, Table(t)); (filename_java, Primitive(String))] ->
169                                         "CsvInterpreter.toFile(WorkingDir.getPath("^filename_java^"), "^object_java^")",
170     | "commit", [(object_java, Singleton(t)); (filename_java, Primitive(String))] ->
171             "CsvInterpreter.toFile(WorkingDir.getPath("^filename_java^"), "^object_java^")", Singleton(t)
172     | "commit", _ -> raise (Error "Commit needs a Table or Singleton and filename")
173
174     (* th of, as in 4.th(users), or, index.th(users) *)
175     | "th", [(index_java, Primitive(Int)); (table_java, Table(udt))] ->
176             table_java ^ ".nth(" ^ index_java ^ ")", Singleton(udt)
177     | "th", _ -> raise (Error "nth needs a Table and index")
178
179     | "size", [table_java, Table(_)] -> table_java ^ ".mySize()", Primitive(Int)
180     | "size", _ -> raise (Error "size needs exactly one Table argument")
181
182     | "stdin", [] -> "input.nextLine()", Primitive(String)
183     | "stdin", _ -> raise (Error "Stdin takes no arguments")
184
185     | "setWorkingDir", [dir_java, Primitive(String)] ->        "WorkingDir.setPath("^dir_java^")", Void
186     | "setWorkingDir", [] -> "WorkingDir.defaultPath()", Void
187     | "setWorkingDir", _ -> raise (Error "WorkingDir takes atmost one String argument")
188
189     | "substring", [(jstring, Primitive(String)); (jstart, Primitive(Int))] ->
190             jstring^".substring("^jstart^")", Primitive(String)
191     | "substring", [(jstring, Primitive(String)); (jstart, Primitive(Int)); (jend, Primitive(Int))] ->
192             jstring^".substring("^jstart^", "^jend^")", Primitive(String)
193     | "substring", _ ->
194             raise (Error "Substring takes a string, a start index and optional end index")
195
196     | "strlen", [string_java, Primitive(String)] ->
197             string_java^".length()", Primitive(Int)
198     | "strlen", _ -> raise (Error "Strlen takes exactly one string argument")
199
200     | "charAt", [(string_java, Primitive(String)); (index_java, Primitive(Int))] ->
201             "String.valueOf("^string_java^".charAt("^index_java^"))", Primitive(String)
202     | "charAt", _ -> raise (Error "StrCharAt takes one string and index argument")
203
204                 | "system", [(string_java, Primitive(String))] ->
205                                         "RtUtil.System("^string_java^")", Primitive(String)
206                 | "system", _ -> raise (Error "System takes one string argument")
207
208                 | "delete", [(table_java, Table(ttype)); (record_java, Singleton(rtype))] when ttype = rtype ->
209                                                 table_java^".delete("^record_java^")", Table(ttype)
210                 | "delete", _ -> raise (Error "Delete takes one table and one record argument")
211
212                 | "argc", [] -> "globArgs.length", Primitive(Int)
213
214                 | "argc", _ -> raise (Error "argc takes no arguments")
215
216                 | "argv", [(jindex, Primitive(Int))] -> "globArgs["^jindex^"]", Primitive(String)
217
218                 | "argv", _ -> raise (Error "argv takes one integer argument")
219
220                 | "tl", [(object_java, Table(t))] -> object_java^".tail()", Table(t)
221
```

```
222                 | "tl", _ -> raise (Error "tail takes one table as an argument")
223
224
225         (* this needs to be last, it's a catch all if the function is not any of the builtins *)
226     | fn, args ->
227         let argtypes = (List.map snd args) in
228         try
229             let (_, javaname), rettype, fn_args = funcsig_of_name ns fn in
230             if not ( fn_args = argtypes ) then
231                 raise (Error ("wrong argument types for "^fn))
232             else
233             let jcall = javaname^"("^(String.concat ", " (List.map fst args))^")" in
234             jcall, rettype
235         with Namespace.Error(_) -> try (* if there isn't a function fn, maybe it's a ctor *)
236             let udt = udt_of_name ns fn in
237             let (_, udt_jname), members = udt in
238             let fsig = (List.map (fun (colname, coltype) -> Primitive(coltype)) members) in
239             if fsig = argtypes then
240                 "new Record("^udt_jname^", "^
241                     (String.concat ", " (List.map
242                         (fun (argjava, argtype) -> "new Data("^argjava^")")
243                         args)
244                 ^")"), Singleton(UDT(udt))
245             else (match args with
246               [tuple_expr_java, Table(TupleUDT(udtlist))] when List.mem udt udtlist ->
247
248                 tuple_expr_java^".getReg("^udt_jname^")", Table(UDT(udt))
249
250             | [tuple_expr_java, Singleton(TupleUDT(udtlist))] when List.mem udt udtlist ->
251
252                 tuple_expr_java^".getReg("^udt_jname^")", Singleton(UDT(udt))
253
254             | _ -> raise (Error ("wrong parameters for constructor."
255                 ^ (string_of_list string_of_lmtype argtypes)
256                 ^ "\nvs actual constructor\n"
257                 ^ (string_of_lmudt udt)
258                 ))
259             )
260         with Namespace.Error(_) -> raise (Error ("no function named "^fn))
261
262 let java_of_cast expr_java from_type to_type =
263     (match (to_type, from_type) with
264           Primitive(x), Primitive(y) when x=y -> expr_java
265         | Primitive(Int), Primitive(Float) -> "(int)"^expr_java
266         | Primitive(Float), Primitive(Int) -> "(double)"^expr_java
267
268         | Primitive(Int), Primitive(String) -> "Integer.parseInt("^expr_java^")"
269         | Primitive(Float), Primitive(String) -> "Double.parseDouble("^expr_java^")"
270         | Primitive(Bool), Primitive(String) -> "Boolean.getBoolean("^expr_java^")"
271
272         | Primitive(String), Primitive(_) -> "\"\"+"^expr_java
273
274         | Table(x), Singleton(y) when x=y -> "new Table("^expr_java^")"
275
276         | _, _ -> raise (Error "cast between two incompatible types")
277     ), to_type
278
279
```

8.1.5 parser.mly

```
1
2 %{ open Ast %}
3
4 /* Michael Vitrano and Jared Pochtar */
5
6 %token SEMI LPAREN RPAREN LBRACE RBRACE COMMA
```

```
7   %token VOID NULL
8   %token LBRACK RBRACK HASH COLUMNAPPEND QMARK
9   %token PLUS MINUS TIMES DIVIDE
10  %token ASSIGN PLUSEQ MINUSEQ TIMESEQ DIVIDEEQ ANDEQ OREQ
11  %token EQ NEQ LT LEQ GT GEQ AND OR
12  %token ITER
13  %token COLON DOT NEWLINE AT BANG
14  %token STRING INT TABLE TYPE BOOL
15  %token RETURN IF ELSE FOR DO WHILE FLOAT
16  %token <float> FPLITERAL
17  %token <int> INTLITERAL
18  %token <bool> BOOLLITERAL
19  %token <string> ID
20
21  %token <string> STRINGLITERAL
22  %token <Ast.expr> INTERPOLATION
23
24  %token <token list> TOKENBUFFER /* do not use inside parser! */
25
26  %token EOF
27
28  %nonassoc NOELSE
29  %nonassoc ELSE
30
31  %nonassoc QMARK
32
33  %left AND OR
34  %left EQ NEQ LT GT LEQ GEQ COLON
35  %right ASSIGN PLUSEQ MINUSEQ TIMESEQ DIVIDEEQ ANDEQ OREQ
36  %left PLUS MINUS
37  %left TIMES DIVIDE
38
39  %left LBRACK RBRACK
40  %left COLUMNAPPEND
41
42  %nonassoc NO_DOT
43  %nonassoc DOT
44
45  %nonassoc BANG
46
47  %start program
48  %type <Ast.program> program
49
50  %start expr
51  %type <Ast.expr> expr
52
53  %%
54
55  program:
56    program_  { {
57                  global_vars = List.rev $1.global_vars;
58                  functions = List.rev $1.functions;
59                  types = List.rev $1.types;
60                  global_code = List.rev $1.global_code
61    } }
62
63  program_:
64     /* nothing */        { {global_vars = []; functions = []; types = []; global_code = [] } }
65   | program_ NEWLINE        { $1 }
66   | program_ fdecl        { {$1 with functions = ($2 :: $1.functions) } }
67   | program_ tdecl        { {$1 with types = ($2 :: $1.types) } }
68   | program_ stmt        { {$1 with global_code = ($2 :: $1.global_code) } }
69   | program_ var_declaration NEWLINE
70      { {$1 with global_vars = ($2 :: $1.global_vars);
71                 (* automatically load tables from same-named csv file *)
72                 global_code = match $2.vtype, $2.isTable with
73                      | Udt(udt_name), true ->
74                          let defaultfileexpr = StringLiteral($2.vname^".csv") in
```

```
75                          let load =  Call("load", [Id(udt_name); defaultfileexpr]) in
76                          Expr(Binop(Id($2.vname), Assign, load))::$1.global_code
77                      | _ -> $1.global_code
78      } }
79   | program_ var_declaration ASSIGN expr NEWLINE
80      { {$1 with global_vars = ($2 :: $1.global_vars);
81                 global_code = Expr(Binop(Id($2.vname), Assign, $4))::$1.global_code } }
82
83  tdecl:
84      TYPE ID LBRACE NEWLINE member_declaration_list RBRACE
85          { { tname = $2; members = List.rev $5 } }
86
87  member_declaration_list:
88      /* nothing */       { [] }
89    | member_declaration_list var_declaration NEWLINE { ($2 :: $1) }
90
91  var_declaration:
92      fulltype ID        { { vname = $2; vtype = fst $1; isTable = snd $1 } }
93
94  fulltype:
95      type_name           { $1, false }
96    | type_name TABLE     { $1, true }
97
98  primitive_type_name:
99      INT     { Int }
100   | FLOAT   { Float }
101   | STRING  { String }
102   | BOOL    { Bool }
103   | VOID    { Void }
104
105 type_name:
106     primitive_type_name { $1 }
107   | ID                  { Udt ($1) }
108   | tuple_type          { Tuple(List.rev $1) }
109
110 tuple_type:
111         ID HASH ID                        { [$3; $1] }        //we're reversing it later
112   | tuple_type HASH ID  { $3::$1 }
113
114
115 fdecl:
116     fulltype ID LPAREN formals_opt RPAREN fdecl_body
117      { { ret_table = snd $1;
118                  rettype = fst $1;
119          fname = $2;
120                  formals = $4;
121                  body = $6 } }
122   | fulltype type_name DOT ID LPAREN formals_opt RPAREN fdecl_body
123      { { ret_table = snd $1;
124                  rettype = fst $1;
125          fname = $4;
126                  formals = {vname = "self"; vtype = $2; isTable = false} :: $6;
127                  body = $8 } }
128
129 formals_opt:
130     /* nothing */ { [] }
131   | formal_list   { List.rev $1 }
132
133 formal_list:
134     var_declaration                                             { [$1] }
135   | formal_list COMMA var_declaration       { $3 :: $1 }
136
137 fdecl_body:
138         block                               { $1 }
139   | ASSIGN expr NEWLINE       { Return($2) }
140
141 block:
142     LBRACE block_line_list RBRACE       { Block(List.rev $2) }
```

```
143
144 block_line_list:
145     /* nothing */  { [] }
146   | block_line_list stmt                { Statement_line($2) :: $1 }
147   | block_line_list        NEWLINE               { $1 }
148   | block_line_list var_declaration ASSIGN expr NEWLINE
149                 { Statement_line(Expr(Binop(Id($2.vname), Assign, $4))) :: Vdecl_line ($2) :: $1 }
150
151 stmt:
152     expr NEWLINE                                                               { Expr($1) }
153   | RETURN expr NEWLINE                                                        { Return($2) }
154   | block                                                                               { $1 }
155
156   | IF LPAREN expr RPAREN stmt %prec NOELSE               { If($3, $5, Block([])) }
157   | IF LPAREN expr RPAREN stmt ELSE stmt               { If($3, $5, $7) }
158
159   | DO stmt WHILE LPAREN expr RPAREN stmt               { DoWhile($2, $5, $7) }
160   | DO WHILE LPAREN expr RPAREN stmt                        { DoWhile(Block([]), $4, $6) }
161   | DO stmt WHILE LPAREN expr RPAREN NEWLINE       { DoWhile($2, $5, Block([])) }
162
163   | expr ITER ID COLON stmt                                                { Iter($3, $1, $5) }
164   | expr ITER ID LBRACE block_line_list RBRACE
165                                                   { Iter($3, $1, Block(List.rev $5)) }
166   | FOR LPAREN expr_opt SEMI expr_opt SEMI expr_opt RPAREN stmt
167                                                                                                        { For($3, $5, $7, $9) }
168
169 expr_opt:
170     /* nothing */ { Noexpr }
171   | expr         { $1 }
172
173 str_literal:
174     STRINGLITERAL                          { StringLiteral($1) }
175   | INTERPOLATION               { Cast(String, $1) }
176   | str_literal INTERPOLATION        { Binop($1, Add, Cast(String, $2)) }
177   | str_literal STRINGLITERAL        { Binop($1, Add, StringLiteral($2)) }
178
179
180 expr:
181     INTLITERAL                { IntLiteral($1) }
182   | FPLITERAL               { FPLiteral($1) }
183   | BOOLLITERAL             { BoolLiteral($1) }
184   | str_literal                   { $1 }
185
186   | NULL fulltype       { Null(fst $2, snd $2) }
187
188   | ID                  { Id($1) }
189   | expr DOT col_name   { Attr($1, $3) }
190   | AT                  { CurrentVar }
191   | DOT col_name        { Attr(CurrentVar, $2) }
192   | COLON ID            {
193         let lhs = Attr(Id("a"), (None, $2)) in
194         let rhs = Attr(Id("b"), (None, $2)) in
195         Binop(lhs, Equal, rhs)
196     }
197
198   | BANG expr                              { First($2) }
199   | TABLE COLON expr %prec NO_DOT              { CastToTable($3) }
200   | primitive_type_name COLON expr %prec NO_DOT { Cast($1, $3) }
201
202   | expr QMARK expr     { Binop($1, Qmark, $3) }
203   | expr AND    expr        { Binop($1, And,   $3) }
204   | expr OR     expr        { Binop($1, Or,    $3) }
205   | expr PLUS   expr        { Binop($1, Add,   $3) }
206   | expr MINUS  expr        { Binop($1, Sub,   $3) }
207   | expr TIMES  expr        { Binop($1, Mult,  $3) }
208   | expr DIVIDE expr        { Binop($1, Div,   $3) }
209
210   | expr ANDEQ    expr        { Binop($1, AndAsn,   $3) }
```

```
211    | expr OREQ     expr         { Binop($1, OrAsn,    $3) }
212    | expr PLUSEQ   expr         { Binop($1, AddAsn,   $3) }
213    | expr MINUSEQ  expr         { Binop($1, SubAsn,   $3) }
214    | expr TIMESEQ  expr         { Binop($1, MultAsn,  $3) }
215    | expr DIVIDEEQ expr         { Binop($1, DivAsn,   $3) }
216
217    | expr EQ     expr        { Binop($1, Equal, $3) }
218    | expr COLON        expr         { Binop($1, Equal, $3) }
219    | expr NEQ    expr         { Binop($1, Neq,   $3) }
220    | expr LT     expr         { Binop($1, Less,  $3) }
221    | expr LEQ    expr         { Binop($1, Leq,   $3) }
222    | expr GT     expr         { Binop($1, Greater,  $3) }
223    | expr GEQ    expr         { Binop($1, Geq,   $3) }
224    | expr ASSIGN expr         { Binop($1, Assign, $3) }
225
226    | ID LPAREN actuals_opt RPAREN { Call($1, $3) }
227    | expr DOT ID LPAREN actuals_opt RPAREN { Call($3, $1::$5) }
228
229    | LPAREN expr RPAREN { $2 }
230
231    | expr LBRACK expr RBRACK                        { Filter($1, $3) }
232    | expr LBRACK expr RBRACK expr        { Join($1, $3, $5) }
233    | expr COLUMNAPPEND expr
234            { Join(CastToTable($1), BoolLiteral(true), CastToTable($3)) }
235
236  col_name:
237      ID %prec NO_DOT     { None, $1 }
238    | ID DOT ID           { Some($1), $3 }
239
240  actuals_opt:
241      /* nothing */ { [] }
242    | actuals_list  { List.rev $1 }
243
244  actuals_list:
245      expr                   { [$1] }
246    | actuals_list COMMA expr { $3 :: $1 }
```

8.1.6 scanner.mll

```
1   (*Michael Vitrano and Jared Pochtar*)
2
3   {
4
5   open Parser
6
7   type string_interpolation_buffer_item =
8       StrLiteral of string
9     | Interpolation of token list
10
11  let buffer_tokens lexer =
12          let tbuf = ref [] in
13          let rec scanner_wrapper lexbuf = (match !tbuf with
14                       [] ->
15                              (match lexer lexbuf with
16                                      TOKENBUFFER(tokens) ->
17                                              tbuf := tokens;
18                                              scanner_wrapper lexbuf
19                                | t -> t
20                              )
21                     | t::rs ->
22                              tbuf := rs;
23                              t
24          ) in scanner_wrapper
25
26  let lex_to_buffer lexer lexbuf =
27          let rec accum_lex prev = (
28                  match lexer lexbuf with
```

```
29                          None -> prev
30                    | Some(tok) -> accum_lex (tok::prev)
31          ) in List.rev (accum_lex [])
32
33  let lex_ending_on endtok lexer =
34      fun lexbuf ->
35          let next_tok = lexer lexbuf in
36          if next_tok = endtok then EOF
37          else next_tok
38
39  }
40
41  let id = ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_' '\'']*
42
43  rule string_literal = parse
44    '"'                                                                          { None }
45  | ([^'"' '\\' '$'])* as lxm             { Some(STRINGLITERAL(lxm))  }
46  | "\\$"                                                                          { Some(STRINGLITERAL("$"))  }
47  | "\\n"                                                                          { Some(STRINGLITERAL("\n")) }
48  | "\\\""                                                                 { Some(STRINGLITERAL("\"")) }
49  | "\\\\"                                                                 { Some(STRINGLITERAL("\\")) }
50  | '\\'(_ as escapechar)                               {
51                  raise (Failure("illegal escape code " ^ Char.escaped escapechar))
52          }
53  | '$'(id as v)                          { Some(INTERPOLATION(Ast.Id(v))) }
54  | "${"                                                                         {
55          try
56              let expr = Parser.expr (buffer_tokens (lex_ending_on RBRACE token)) lexbuf
57              in Some(INTERPOLATION(expr))
58          with e -> raise (Failure "error parsing a string interpolation")
59          }
60
61
62  and token = parse
63    [' ' '\t' '\r']         { token lexbuf } (* Whitespace *)
64  | "/*"                                { multiComment 0 lexbuf }      (* Multi Line Comments *)
65  | "//"                                { singleComment lexbuf }     (*Single Line Comments *)
66
67  | '\n'     { NEWLINE }
68  | ",\n"           { token lexbuf }
69
70  | '"'            { TOKENBUFFER(lex_to_buffer string_literal lexbuf) }
71  | "\"\""   { STRINGLITERAL("") }
72
73  | "true"   { BOOLLITERAL(true) }
74  | "false"  { BOOLLITERAL(false) }
75  | "void"   { VOID }
76  | "null"   { NULL }
77
78  | '('      { LPAREN }
79  | ')'      { RPAREN }
80  | '{'      { LBRACE }
81  | '}'      { RBRACE }
82  | '['      { LBRACK }
83  | ']'      { RBRACK }
84  | '#'      { HASH }
85  | ','      { COMMA }
86
87  | '+'      { PLUS }
88  | '-'      { MINUS }
89  | '*'      { TIMES }
90  | '/'      { DIVIDE }
91
92  | "&&"     { AND }
93  | "||"     { OR }
94  | "and"    { AND }
95  | "or"     { OR }
96
```

```
97  | '='      { ASSIGN }
98  | "+="     { PLUSEQ }
99  | "-="     { MINUSEQ }
100 | "*="     { TIMESEQ }
101 | "/="     { DIVIDEEQ }
102 | "&&="    { ANDEQ }
103 | "||="    { OREQ }
104
105 | "=="     { EQ }
106 | "!="     { NEQ }
107 | '<'      { LT }
108 | "<="     { LEQ }
109 | ">"      { GT }
110 | ">="     { GEQ }
111 | '|'      { ITER }
112 | ':'      { COLON }
113 | ';'           { SEMI }
114 | '.'      { DOT }
115 | '@'      { AT }
116 | '!'      { BANG }
117 | "|+|"           { COLUMNAPPEND }
118 | '?'      { QMARK }
119
120 | "string" { STRING }
121 | "float"  { FLOAT }
122 | "bool"   { BOOL }
123 | "table"  { TABLE }
124 | "type"   { TYPE }
125 | "if"     { IF }
126 | "else"   { ELSE }
127 | "for"    { FOR }
128 | "do"           { DO }
129 | "while"  { WHILE }
130 | "return" { RETURN }
131 | "int"    { INT }
132 | ['0'-'9']+'.'['0'-'9']* as lxm { FPLITERAL(float_of_string lxm) }
133 | ['0'-'9']+ as lxm { INTLITERAL(int_of_string lxm) }
134 | id as lxm { ID(lxm) }
135 | eof { EOF }
136 | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
137
138 and multiComment level = parse
139   "*/" { if level = 0 then token lexbuf
140                 else multiComment (level-1) lexbuf }
141 | "/*" { multiComment (level+1) lexbuf }
142 | eof  { raise (Failure "file ends in a comment") }
143 | _    { multiComment level lexbuf }
144
145 and singleComment = parse
146   "\n" { NEWLINE }
147 | _    { singleComment lexbuf }
```

8.1.7 translator.ml

```
1  (*Michael Vitrano and Jared Pochtar*)
2
3  open Ast
4  open Namespace
5  open Namespace_pp
6
7  exception Error of string
8
9  (* variable declarations; these go in the global scope of the java class *)
10 let java_of_vdecl ns vdecl =
11         let newScope, ((_, jname), lmtype) = scope_with_var ns vdecl in
12         let declcode = (java_signature lmtype) ^ " " ^ jname in
13         (newScope, declcode)
```

```
14
15
16  (* Just to be safe, parenthesise every expr.  Java does this weird thing
17  where you can't ...; (expr); so to get around that I expose the one that
18  is unparened, for stmts to use that, but then it uses recursively the parened one. *)
19
20  let rvalue (j, t) = j, t, false
21
22  let paren str = "("^str^")"
23
24  let rec java_of_expr ns expr =
25          let code, ltype, assignable = java_of_unparenthesized_expr ns expr in
26      "(" ^ code ^ ")", ltype
27  and java_of_unparenthesized_expr ns = function      (* returns java, type, isLvalue *)
28        Id(var) ->
29                  (try
30                            let ((vname, javaname), vtype), was_captured = var_of_name ns var in
31                          javaname, vtype, not was_captured
32                   with e -> try                (* maybe it's a column on the default var? *)
33                           let defaultVarAttr = Attr(CurrentVar, (None, var)) in
34                             java_of_unparenthesized_expr ns defaultVarAttr
35           with
36              (* hack to allow certain error messages through *)
37                e' when e' = (Namespace.Error "ambiguous member access") -> raise e'
38              | e' -> raise e
39                   )
40      | IntLiteral(i) -> string_of_int i, Primitive(Int), false
41      | FPLiteral(f) -> string_of_float f, Primitive(Float), false
42      | BoolLiteral(b) -> (if b then "true" else "false"), Primitive(Bool), false
43      | StringLiteral(s) -> qt (String.escaped s), Primitive(String), false
44      | Null(tname, isTable) -> "null", lmtype_of_ast_type ns tname isTable, false
45      | Noexpr -> "", Void, false
46
47      | Cast(ast_t, expr) ->
48              let java_of_expr, expr_type = java_of_expr ns expr in
49              let to_type = lmtype_of_ast_type ns ast_t false in
50              rvalue (Overloading.java_of_cast java_of_expr expr_type to_type)
51
52      | CastToTable(Id(udtname)) when exists_udt_named ns udtname ->
53              let udt = udt_of_name ns udtname in
54              let ((lmname, jname), _) = udt in
55              "new Table("^jname^")", Table(UDT(udt)), false
56
57      | CastToTable(expr) ->
58              let java_of_expr, expr_type = java_of_expr ns expr in (
59              match expr_type with
60                    Singleton(udt) -> "new Table("^java_of_expr^")", Table(udt), false
61                  | Table(udt) -> java_of_expr, Table(udt), false
62                  | _ -> raise (Error "can't cast non-singletons to tables")
63              )
64
65      | Call("load", [Id(udt_name); filename_expr]) ->
66              let udt = udt_of_name ns udt_name in
67              let ((_, udt_jname), _) = udt in
68
69              let filename_java, filename_type = java_of_expr ns filename_expr in
70              if filename_type = Primitive(String) then
71                  "CsvInterpreter.toTable(WorkingDir.getPath("^filename_java^"), "^udt_jname^")", Table(UDT(udt)), false
72              else
73                  raise (Error "load needs a udt and a string")
74      | Call("load", _) -> raise (Error "loadtable needs a udt and a string")
75
76      | Call(fn, args) ->
77              rvalue (Overloading.java_of_function_call ns fn (List.map (java_of_expr ns) args))
78
79      | Binop(lhs, op, rhs) ->
80              let lhs_java, lhs_type, isAssignable = java_of_unparenthesized_expr ns lhs in
81              let rhs_java, rhs_type = java_of_expr ns rhs in
```

```
82              rvalue (Overloading.java_of_binop ns
83                  isAssignable (paren lhs_java) rhs_java
84                  (lhs_type, op, rhs_type)
85              )
86
87      | First(expr) ->
88              let java_of_expr, expr_type = java_of_expr ns expr in (
89              match expr_type with
90                  Table(udt) -> java_of_expr^".first()", Singleton(udt), false
91                  | _ -> java_of_expr, expr_type, false
92              )
93
94      | Attr(singleton, (attrtype, attr)) -> (
95              let java_of_obj, obj_type = java_of_expr ns singleton in
96              try (
97              match obj_type with
98                  | Singleton(obj_type) ->
99                      let index, coltype = col_index_of_obj obj_type attrtype attr in
100                     let data_getter = (match coltype with
101                            Int -> "._Integer"
102                          | Float -> "._Float"
103                          | String -> "._String"
104                          | Bool -> "._Bool"
105                     )
106                     in let getter = java_of_obj ^ ".entries["^(string_of_int index)^"]"^data_getter
107                     in getter, Primitive(coltype), true
108                 | _ -> raise (Error "access of members of type that doesn't have members")
109             )
110             with e -> (try
111                 (* if there's no actual attribute, maybe it's a function call.  fn(e) == e.fn *)
112                 if attrtype = None then
113                     rvalue (Overloading.java_of_function_call ns attr [(java_of_obj, obj_type)])
114                 else raise e
115             with e' -> raise e)
116             )
117
118     | CurrentVar ->
119         (match ns.default_variable with
120             | None -> raise (Error "can't address default variable here; there's none set")
121             | Some(obj_jname, obj_type) -> obj_jname, obj_type, false
122         )
123
124     | Filter(table_expr, predicate) ->
125             (match java_of_expr ns table_expr with
126                 | table_java, Table(udt) ->
127
128                     let capturing = { (nest_scope ns) with
129                         default_variable = Some("record", Singleton(udt));
130                         isCapturing = true
131                     } in
132
133                     let pred_java, pred_type = java_of_expr capturing (First predicate) in
134                     let pred_call = functor_call capturing pred_java in
135
136                     (match pred_type with
137                         | Primitive(Bool) ->
138                             let pred_call_java = pred_call
139                                 "FilterPredicate"
140                                 "public boolean test(Record record)"
141                                 None
142                             in table_java^".filter("^pred_call_java^")", Table(udt), false
143
144                         | Singleton(map_udt) ->
145                             let pred_call_java = pred_call
146                                 "FilterMap"
147                                 "public Record map(Record record)"
148                                 (Some(map_udt))
149                             in table_java^".filterMap("^pred_call_java^")", Table(map_udt), false
```

```
150
151                         | _ -> raise (Error "filter predicate is invalid")
152                     )
153
154                 | _, _ -> raise (Error "filtering on a non-table")
155             )
156
157         | Join(lhs, predicate, rhs) ->
158             (match java_of_expr ns lhs, java_of_expr ns rhs with
159                 | (lhs_java, Table(lhs_udt)), (rhs_java, Table(rhs_udt)) ->
160
161                     let combined_udt, combined_udt_is_stable =
162                         combine_types lhs_udt rhs_udt
163                     in
164
165                     let default_var =
166                         if combined_udt_is_stable then
167                             Some("record", Singleton(combined_udt))
168                         else None
169                     in
170
171                     let capturing = { (nest_scope ns) with
172                         default_variable = default_var;
173                         isCapturing = true
174                     } in
175
176                     let capturing = scope_with_jvar_passthrough capturing "a" (Singleton(lhs_udt)) in
177                     let capturing = scope_with_jvar_passthrough capturing "b" (Singleton(rhs_udt)) in
178
179                     let pred_java, pred_type = java_of_expr capturing (First predicate) in
180                     let pred_call = functor_call capturing pred_java in
181
182                     (match pred_type with
183                         | Primitive(Bool) ->
184                             if not combined_udt_is_stable then
185                                 raise (Error ("cannot (non-map) join tables with shared components"))
186                             else
187                             let pred_call_java = pred_call
188                                 "JoinPredicate"
189                                 "public boolean test(Record record, Record a, Record b)"
190                                 None
191                             in lhs_java^".join("^rhs_java^", "^pred_call_java^")", Table(combined_udt), false
192
193                         | Singleton(udt) ->
194                             let pred_call_java = pred_call
195                                 "JoinMap"
196                                 "public Record map(Record record, Record a, Record b)"
197                                 (Some(udt))
198                             in lhs_java^".joinMap("^rhs_java^", "^pred_call_java^")", Table(udt), false
199
200                         | _ -> raise (Error "join predicate is invalid")
201                     )
202                 | (_, _), (_, _) -> raise (Error "joining on two things which are not both tables")
203             )
204
205
206 let java_of_expr' ns expr = let j, _, _ = (java_of_unparenthesized_expr ns expr) in j
207 let java_of_pred ns pred = match java_of_unparenthesized_expr ns pred with
208       java, Primitive(Bool), _ -> java
209     | _ -> raise (Error "used a non-bool where a boolean expression was expected")
210
211
212 let rec java_of_block ns b =
213     let finalenv, blockcode = List.fold_left
214         (fun (env, code) lmline -> match lmline with
215               Statement_line(s) -> (env, code ^ java_of_stmt env s ^ "\n")
216             | Vdecl_line(v) ->  let newenv, declcode = java_of_vdecl env v in
217                                 (newenv, code ^ declcode ^ ";\n")
```

```
218         ) (nest_scope ns, "") b
219     in "{\n" ^ blockcode ^ "\n}"
220
221 and java_of_stmt ns = function
222       Block(body) -> java_of_block ns body
223     | Expr(e) -> java_of_expr' ns e ^ ";"
224     | Return(expr) ->
225                 let retexpr_java, retexprtype = java_of_expr ns expr in
226                 if ns.returntype = Some(retexprtype) then
227                 " return " ^ retexpr_java ^ ";"
228                 else raise (Error "returning wrong type")
229
230     | If(pred, t, e) -> " if (" ^ java_of_pred ns pred ^ ") {"
231                         ^ java_of_stmt ns t
232                         ^ "} else {"
233                         ^ java_of_stmt ns e
234                         ^ "}"
235     | For(init, pred, post, body) ->
236         " for (" ^ java_of_expr' ns init ^ "; "
237                  ^ java_of_pred ns pred ^ "; "
238                  ^ java_of_expr' ns post ^ "){"
239             ^ java_of_stmt ns body ^
240         "}"
241     | DoWhile(pre, pred, post) ->
242                 " while (true) {"
243                         ^ java_of_stmt ns pre
244                         ^ " if(!("^ java_of_pred ns pred ^ ")) { break; } "
245                         ^ java_of_stmt ns post
246                 ^ "}"
247     | Iter(newvar, table, body) ->
248                 let ns = nest_scope ns in
249                 let java_of_table, table_type = java_of_expr ns table in
250                 match table_type with
251                           Table(udt) ->
252                 let ns, ((_, itervar_jname), _) =
253                     scope_with_var_name_type ns newvar (Singleton(udt))
254                                 in
255                 "for (Record " ^ itervar_jname ^ " : " ^ java_of_table ^ "){"
256                           ^ java_of_stmt ns body ^
257                 "}"
258                         | _ -> raise (Error "itterating on something not a table")
259
260
261 let java_of_program main_class_name program =
262     reset_functors ();
263         let ns = new_scope main_class_name in
264
265         let ns, udt_decls, udt_builders = List.fold_left
266         (fun (env, decls, builders) udtdecl ->
267                                 let newenv, ((lmname, jname), members), declname = scope_with_ast_udt_decl env udtdecl
268                                 in let declcode = "static UserDefinedType " ^ declname ^ ";\n"
269                                 in let buildercode =
270                                         jname ^ " = new UserDefinedType("^qt lmname^");\n"
271                                   ^ (String.concat "\n" (List.map
272                                                   (fun ((coloftype, colname), colprimtype) ->
273                                                         jname^".add("^(qt(coloftype^"'s "^colname))^", "
274                                                         ^ (match colprimtype with
275                                                                   Int -> "Data.INT_TYPE"
276                                                                 | Float -> "Data.FLOAT_TYPE"
277                                                                 | String -> "Data.STRING_TYPE"
278                                                                 | Bool -> "Data.BOOL_TYPE")
279                                                         ^ ");") members))
280                                 in
281                                 (newenv, decls ^ declcode, builders ^ buildercode ^ "\n\n")
282         ) (ns, "", "") program.types
283         in
284
285         let ns, global_var_decls = List.fold_left
```

```
286         (fun (env, alldeclcode) vdecl ->
287                             let newenv, vdeclcode = java_of_vdecl env vdecl in
288                             (newenv, alldeclcode ^ "static " ^ vdeclcode ^ ";\n")
289         ) (ns, "") program.global_vars
290         in
291 
292         let ns, function_decls = List.fold_left
293         (fun (env, function_decls) fdecl ->
294             let newenv, (_, rettype, _), jname = scope_with_ast_func env fdecl in
295             newenv, (jname, rettype, fdecl)::function_decls
296         ) (ns, []) program.functions
297     in
298 
299     let java_of_functions = String.concat "\n\n" (List.map (fun (fjname, frettype, fdecl) ->
300         let fenv = { (nest_scope ns) with returntype = Some(frettype) } in
301         let fenv', formals_decl_code = List.fold_left
302             (fun (env, alldeclcode) vdecl ->
303                 let newenv, vdeclcode = java_of_vdecl env vdecl in
304                 (newenv, vdeclcode :: alldeclcode)
305             ) (fenv, []) fdecl.formals
306         in
307         "public static " ^ (java_signature frettype) ^ " " ^ fjname
308         ^"("^(String.concat ", " (List.rev formals_decl_code))^") {\n"
309             ^(java_of_stmt fenv' fdecl.body)
310         ^ "}"
311         ) function_decls)
312         in
313 "
314 /*
315     Generated by the Return of the Table compiler.
316  */
317 
318 
319 import rtlib.*;
320 import java.util.Scanner;
321 
322 " ^ !functor_declarations ^ "
323 
324 public class "^main_class_name^" {
325     " ^ udt_decls ^ global_var_decls ^ java_of_functions ^ "
326 
327     private static Scanner input;
328                 static String globArgs[];
329 
330         public static void main(String args[]) {
331         input = new Scanner(System.in);
332                             globArgs = new String[args.length];
333                             for(int globArgCounter = 0; globArgCounter < args.length; globArgCounter++)
334                                     globArgs[globArgCounter] = args[globArgCounter];
335 
336         "
337 
338         (* type constructions *)
339         ^ udt_builders
340 
341         (* program statements *)
342         ^ java_of_block ns (List.map (fun s->Statement_line(s)) program.global_code)
343         ^ "
344     }
345 }
346 "
347 
348 
349 
350 
```

8.2 Framework

8.2.1 WorkingDir.java

```
1   // Michael Vitrano and Jared Pochtar
2
3
4   package rtlib;
5
6   public class WorkingDir {
7
8           private static String working;
9
10          static {
11          working = "./";
12      }
13
14          public static String getPath(String aPath){
15                  if(aPath.startsWith("/")) {
16                          return aPath;
17                  } else {
18                          return working + aPath;
19          }
20          }
21
22          public static void setPath(String aPath){
23          working = aPath;
24      }
25
26          public static void defaultPath(){
27          working = "./";
28      }
29  }
```

8.2.2 UserDefinedType.java

```
1   // Michael Vitrano and Jared Pochtar
2
3   package rtlib;
4
5   import java.util.ArrayList;
6   import java.lang.Math;
7
8
9   public class UserDefinedType {
10
11          ArrayList<String> names;
12          ArrayList<Integer> types;
13          String name;
14          boolean isTuple;
15
16          public UserDefinedType(String aName){
17                  name = aName;
18                  types = new ArrayList<Integer>();
19                  names = new ArrayList<String>();
20                  isTuple = false;
21          }
22          public void add(String name, int type){
23                  types.add(type);
24                  names.add(name);
25          }
26
27          public void addCompType(ArrayList<String> newNames, ArrayList<Integer> newTypes){
28                  names.addAll(newNames);
29                  types.addAll(newTypes);
30          }
31
```

```
32          public int[] typeList(){
33                  int a[] = new int[types.size()];
34
35                  for(int i = 0; i < types.size(); i++){
36                          a[i] = types.get(i);
37                  }
38                  return  a;
39          }
40          public String[] nameList(){
41                  String a[] = new String[names.size()];
42
43                  for(int i = 0; i < names.size(); i++){
44                          a[i] = names.get(i);
45                  }
46                  return  a;
47          }
48
49          public int getSize(){
50               return names.size();
51      }
52
53          public static TupleType combineTypes(UserDefinedType a, UserDefinedType b){
54                           return new TupleType(a.name + b.name, a, b);
55          }
56
57          public String toString(){
58                  String out = "| ";
59                  for(String name : names){
60                          out += name + "\t| ";
61                  }
62                  return out;
63          }
64
65          public String toString(int[] size){
66                  String out = "| ";
67                  for(int i = 0; i < names.size(); i++){
68                          String name = names.get(i);
69                          int frontPad, backPad;
70                          if(name.length() < size[i]){
71                                  frontPad = (int) Math.floor((size[i]-name.length())/2.0);
72                                  backPad = size[i]-name.length() - frontPad;
73                          }else{
74                                  frontPad = 0;
75                                  backPad = 0;
76                          }
77                          for(int j = 0; j < frontPad; j++)
78                                  out += " ";
79                          out += name;
80                          for(int j = 0; j < backPad; j++)
81                                  out += " ";
82                          out += " | ";
83                  }
84                  return out;
85          }
86
87      public String toFile(){
88       String out = "";
89                  for(int i = 0; i < names.size(); i++){
90                    if( i != names.size() - 1 )
91                out += names.get(i) + ",";
92              else
93                out += names.get(i);
94                  }
95                  return out;
96      }
97
98  }
```

8.2.3 TupleType.java

```
1   // Michael Vitrano and Jared Pochtar
2
3   package rtlib;
4
5   import java.util.ArrayList;
6   import java.lang.Math;
7
8   public class TupleType extends UserDefinedType {
9
10          ArrayList<UserDefinedType> compTypes;
11
12          public TupleType(String aName, UserDefinedType ... components){
13                  super(aName);
14                  compTypes = new ArrayList<UserDefinedType>();
15                  for(int i = 0; i < components.length; i++){
16                          if(components[i].isTuple){
17                                  @SuppressWarnings("unchecked")
18                                  TupleType comp = (TupleType) components[i];
19                                  this.compTypes.addAll(comp.compTypes);
20                          }else{
21                                  compTypes.add(components[i]);
22                          }
23                          super.addCompType(components[i].names, components[i].types);
24                  }
25                  super.isTuple = true;
26          }
27
28          public int[] getTypeBounds(UserDefinedType type){
29                  int start, end, last;
30                  last = 0;
31                  //System.out.println("looking for " + type.name);
32
33                  for(int i = 0; i < compTypes.size(); i++){
34                          start = last;
35                          end = start + compTypes.get(i).getSize();
36                          last = end + 1;
37                          //System.out.println("examining " + compTypes.get(i).name);
38                          if (compTypes.get(i).name.equals(type.name) ){
39                                  int ret[] = {start, end};
40                                  //System.out.println("returning " + ret[0] +" , "+ret[1] );
41                                  return ret;
42                          }
43                  }
44                  //System.out.println("returning null" );
45                  return null;
46          }
47
48          public void printStat(){
49                  System.out.println("name:" + super.name);
50                  System.out.println("col names:");
51                  for(String cName : super.names)
52                          System.out.println(cName);
53
54                  System.out.println("comp type names:");
55                  for(UserDefinedType cType : compTypes)
56                          System.out.println(cType.name);
57
58
59          }
60
61
62
63  }
64
65
```

8.2.4 Table.java

```java
// Michael Vitrano and Jared Pochtar

package rtlib;

import java.util.ArrayList;
import java.util.Iterator;


public class Table implements Iterable<Record>{

    ArrayList<Record> rows;
                UserDefinedType type;
                ArrayList<IteratorHelper> iters;
                int sizes[];
                boolean isTupleTable;
        public Table(UserDefinedType aType){
                type = aType;
                rows = new ArrayList<Record>();
                iters = new ArrayList<IteratorHelper>();
                sizes = new int[type.names.size()];
                for(int i=0; i<type.names.size(); i++)
                        sizes[i] = type.names.get(i).length();
        }

    public Record first() {
        if (rows.size() > 0) {
            return rows.get(0);
        } else {
            return null;
        }
    }

    public Table(Record r){
                        type = r.type;
                        rows = new ArrayList<Record>();
                        iters = new ArrayList<IteratorHelper>();
                        sizes = new int[type.names.size()];
                        for(int i=0; i<type.names.size(); i++)
                                sizes[i] = type.names.get(i).length();
                        addRow(r);
        }

        public int mySize(){
                return rows.size();
        }

        public Iterator<Record> iterator() {
                TableIterator iter = new TableIterator(this);
                iters.add(new IteratorHelper(iter));
                return iter;

        }

        public Table addRow(Record newRow){
                for(int i = 0; i < newRow.sizes.length; i++){
                        if(this.sizes[i]<newRow.sizes[i])
                                this.sizes[i] = newRow.sizes[i];
                }
                rows.add(newRow);
                return this;
        }

    public Table append(Table other){
                Table newTable = new Table(type);
                for(Record row : this ){
                        newTable.addRow(row);
```

```
67                  }
68                  for(Record row : other ){
69                          newTable.addRow(row);
70                  }
71                  return newTable;
72          }
73
74          public void delete(Record a){
75                  int i = rows.indexOf(a);
76                  if( i != -1){
77                          rows.remove(i);
78                          for(IteratorHelper help : iters){
79                                  if(i < help.getLoc()){
80                                          help.getIter().decLoc();
81                                  }
82                          }
83                  }
84          }
85
86          public void deleteIter(TableIterator iter){
87                  Iterator<IteratorHelper> arrayIter = iters.iterator();
88                  while(arrayIter.hasNext()){
89                          IteratorHelper next = arrayIter.next();
90                          if(next.getIter() == iter)
91                                  arrayIter.remove();
92                  }
93          }
94
95          public Table tail(){
96                  if(rows.size() > 1){
97                          Table ret = new Table(type);
98                          for(int i = 1; i < rows.size(); i++){
99                                  ret.addRow(rows.get(i));
100                         }
101                         return ret;
102                 }else
103                         return null;
104         }
105
106
107         public Table copyOf(){
108                 Table newTable = new Table(type);
109                 for(Record row : rows)
110                         newTable.addRow(row.copyOf());
111                 return newTable;
112         }
113
114         public Table prepend(Record r) {
115                 Table newTable = new Table(type);
116                 newTable.addRow(r);
117                 for(Record row : this){
118                         newTable.addRow(row);
119                 }
120                 return newTable;
121          }
122
123         public String toString(){
124                 String out = type.toString(sizes) + "\n";
125                 for(int i = 0; i < rows.size(); i++){
126                         if( i != rows.size() - 1)
127                                 out += rows.get(i).toString(sizes) + "\n";
128                         else
129                                 out += rows.get(i).toString(sizes) ;
130                 }
131                 return out;
132         }
133
134     public String toFile(){
```

```
135     String out = "";
136                 for(int i = 0; i < rows.size(); i++){
137                         if(i != rows.size() -1 )
138                                 out += rows.get(i).toFile() + "\n";
139                         else
140                                 out += rows.get(i).toFile();
141                 }
142                 return out;
143         }
144 
145         public ArrayList<Record> getRows(){return rows;}
146 
147         public UserDefinedType getType(){
148                 return type;
149         }
150 
151         public Table filter(FilterPredicate pred){
152                 Table newTable = new Table(type);
153                 for(Record row : rows){
154                         if(pred.test(row))
155                                 newTable.addRow(row);
156                 }
157                 return newTable;
158         }
159 
160         public Table filterMap(FilterMap map){
161                 Table newTable = new Table(map.getType());
162                 for(Record row : rows){
163                         Record toAdd = map.map(row);
164                         if(toAdd != null)
165                                 newTable.addRow(toAdd);
166                 }
167                 return newTable;
168         }
169 
170         public Table joinMap(Table rhs, JoinMap map){
171         UserDefinedType newType = UserDefinedType.combineTypes(this.type, rhs.type);
172         int sizeof_a = this.type.types.size();
173         int sizeof_b = rhs.type.types.size();
174 
175                 Table newTable = new Table(map.getType());
176 
177                 for(Record thisRow : this.rows){
178                         for(Record rhsRow : rhs.rows){
179                 Record newRec = new Record(newType);
180 
181                 for(int k = 0; k < sizeof_a; k++){
182                     newRec.addData(thisRow.getData(k));
183                 }
184 
185                 for(int k = 0; k <  sizeof_b; k++){
186                     newRec.addData(rhsRow.getData(k));
187                 }
188 
189                                 Record toAdd = map.map(newRec, thisRow, rhsRow);
190                 if(toAdd != null) {
191                                         newTable.addRow(toAdd);
192                 }
193             }
194                 }
195                 return newTable;
196         }
197 
198         public Table join(Table rhs, JoinPredicate pred){
199                 UserDefinedType newType = UserDefinedType.combineTypes(this.type, rhs.type);
200         int sizeof_a = this.type.types.size();
201         int sizeof_b = rhs.type.types.size();
202 
```

```
203                 Table newTable = new Table(newType);
204
205         for (Record a : this) {
206             for (Record b : rhs) {
207                 Record newRec = new Record(newType);
208
209                 for(int k = 0; k < sizeof_a; k++){
210                     newRec.addData(a.getData(k));
211                 }
212
213                 for(int k = 0; k <  sizeof_b; k++){
214                     newRec.addData(b.getData(k));
215                 }
216
217                 if (pred.test(newRec, a, b)) {
218                     newTable.addRow(newRec);
219                 }
220             }
221         }
222
223         return newTable;
224         }
225
226         public Table getReg(UserDefinedType t){
227                 @SuppressWarnings("unchecked")
228                 TupleType tType = (TupleType) type;
229                 int bounds[] = tType.getTypeBounds(t);
230                 Table ret = new Table(t);
231                 //System.out.println("bounds are " + bounds[0] + " and " + bounds[1]);
232                 for(Record toAdd : rows){
233                         Record newRec = new Record(t);
234                         for(int i = bounds[0]; i < bounds[1]; i++)
235                                 newRec.addData(toAdd.entries[i]);
236                         ret.addRow(newRec);
237                 }
238                 return ret;
239         }
240
241         private class TableIterator implements Iterator<Record>{
242                 private Table t;
243                 int i;
244                 public TableIterator(Table aTable){
245                         t = aTable;
246                         i = 0;
247                 }
248                 public boolean hasNext(){
249                         boolean next = i < t.rows.size();
250                         if(next)
251                                 return true;
252                         else{
253                                 t.deleteIter(this);
254                                 return false;
255                         }
256                 }
257                 public Record next(){
258                         i++;
259                         return t.rows.get(i-1);
260                 }
261
262                 public void decLoc(){i--;}
263
264                 public void remove () {}
265                 public int getLoc(){return i;}
266         }
267
268         private class IteratorHelper{
269                 private TableIterator iter;
270
```

```
271                public IteratorHelper(TableIterator aIter){
272                        iter = aIter;
273                }
274                public int getLoc(){return iter.getLoc();}
275
276                public TableIterator getIter(){return iter;}
277
278        }
279
280 }
```

8.2.5 RtUtil.java

```
1  // Michael Vitrano and Jared Pochtar
2
3  package rtlib;
4  import java.io.*;
5
6  public class RtUtil{
7
8
9          public static String System(String command){
10                 try{
11                 Process p=Runtime.getRuntime().exec(command);
12                 p.waitFor();
13                 BufferedReader reader=new BufferedReader(new InputStreamReader(p.getInputStream()));
14                 String test = reader.readLine();
15                 String out = "";
16                 while(test != null) {
17                         out += test + "\n";
18                         test=reader.readLine();
19                 }
20                 return out;
21         }catch(Exception e){
22                 return null;
23         }
24         }
25
26
27 }
```

8.2.6 Record.java

```
1  // Michael Vitrano and Jared Pochtar
2
3  package rtlib;
4
5  import java.util.Iterator;
6
7
8  public class Record extends Table implements Iterable<Record> {
9
10         public Data[] entries;
11         private int next;
12     UserDefinedType type;
13         int sizes[];
14         public Record(UserDefinedType aType){
15                 super(aType);
16                 type = aType;
17                 entries = new Data[type.getSize()];
18                 next = 0;
19                 sizes = new int[type.names.size()];
20         }
21
22         public Record(UserDefinedType aType, Data ... dataMembers){
23                 super(aType);
```

```
24                  type = aType;
25                  entries = new Data[type.getSize()];
26                  next = 0;
27                  sizes = new int[type.names.size()];
28                  for(int i = 0; i < dataMembers.length; i++){
29                          entries[i] = dataMembers[i];
30                          sizes[i] = dataMembers[i].toString().length();
31                          next++;
32                  }
33
34          }
35
36          public Record copyOf(){
37                  Record copy = new Record(type);
38                  for(int i = 0; i < entries.length; i++)
39                          copy.addData(entries[i].copyOf());
40                  return copy;
41          }
42
43          public Table addRow(Record newRow){
44          return null;
45      }
46
47          public Iterator<Record> iterator() {
48                  return this.new RecordIterator(this);
49          }
50
51          public void addData(Data newData){
52                  entries[next] = newData;
53                  sizes[next] = newData.toString().length();
54                  next++;
55          }
56
57          public Data getData(int i){
58                  return entries[i];
59          }
60
61          public void SetData(int i, Data d){
62                  entries[i] = d;
63          }
64
65          public String toString(){
66                  String out = "| ";
67                  for(int i = 0; i < entries.length; i++){
68              out += entries[i] + "\t| ";
69                  }
70                  return out;
71          }
72
73          public String toString(int[] size){
74                  String out = "| ";
75                  for(int i = 0; i < entries.length; i++){
76                          String str = entries[i].toString();
77                          int frontPad, backPad;
78                          if(str.length() < size[i]){
79                                  frontPad = (int)Math.floor((size[i]-str.length())/2.0);
80                                  backPad = size[i]-str.length() - frontPad;
81                          }else{
82                                  frontPad = 0;
83                                  backPad = 0;
84                          }
85                          for(int j = 0; j < frontPad; j++)
86                                  out += " ";
87                          out += str;
88                          for(int j = 0; j < backPad; j++)
89                                  out += " ";
90                          out += " | ";
91                  }
```

```
92                  return out;
93          }
94
95          public int mySize(){return 1;}
96
97          public int size(){return entries.length; }
98
99          public String toFile(){
100                 String out = "";
101                 for(int i = 0; i < entries.length; i++){
102                   if( i != entries.length - 1 )
103               out += entries[i] + ",";
104             else
105               out += entries[i];
106                 }
107                 return out;
108         }
109
110         private class RecordIterator implements Iterator<Record>{
111                 private boolean next = true;
112                 private Record data;
113                 public RecordIterator(Record aData){
114                         data = aData;
115                 }
116                 public boolean hasNext(){
117                         return next;
118                 }
119                 public Record next() {
120                         next = false;
121                         return data;
122                 }
123                 public void remove() {                }
124         }
125 }
```

8.2.7 JoinPredicate.java

```
1   // Michael Vitrano and Jared Pochtar
2
3   package rtlib;
4
5   public interface JoinPredicate {
6       boolean test(Record record, Record a, Record b);
7   }
```

8.2.8 JoinMap.java

```
1   // Michael Vitrano and Jared Pochtar
2
3   package rtlib;
4
5   public interface JoinMap {
6       public Record map (Record record, Record a, Record b);
7       public UserDefinedType getType();
8   }
```

8.2.9 FilterPredicate.java

```
1   // Michael Vitrano and Jared Pochtar
2
3   package rtlib;
4
5   public interface FilterPredicate {
6       boolean test(Record elem);
7   }
```

8.2.10 FilterMap.java

```
1  // Michael Vitrano and Jared Pochtar
2
3  package rtlib;
4
5  public interface FilterMap {
6      public Record map(Record elem);
7                  public UserDefinedType getType();
8  }
```

8.2.11 Data.java

```
1  // Michael Vitrano and Jared Pochtar
2
3  package rtlib;
4
5  import java.util.Date;
6  import java.text.SimpleDateFormat;
7
8  public class Data {
9
10         public static final int INT_TYPE = 1;
11         public static final int FLOAT_TYPE = 2;
12         public static final int STRING_TYPE = 3;
13         public static final int BOOL_TYPE = 4;
14         public static final int DATE_TYPE = 5;
15     public int _Integer;
16         public double _Float;
17         public String _String;
18         public boolean _Bool;
19         public Date _Date;
20         private int type;
21
22
23         /* Encapsulates the basic data types. Constructor knows which type
24          * is correct.
25          *
26          */
27         public Data(String aString){
28                 try{
29                         SimpleDateFormat format = new SimpleDateFormat();
30                         _Date = format.parse(aString);
31                         type = DATE_TYPE;
32                 }catch(Exception e){
33                         _String = aString;
34                         type = STRING_TYPE;
35                 }
36         }
37
38         public Data(int anInt){
39                 _Integer = anInt;
40                 type = INT_TYPE;
41         }
42
43         public Data(double aFloat){
44                 _Float = aFloat;
45                 type = FLOAT_TYPE;
46         }
47
48         public Data(boolean aBool){
49                 _Bool = aBool;
50                 type = BOOL_TYPE;
51         }
52
53         public int getType(){
54                 return type;
```

```
55          }
56
57          public String getString(){
58                      return _String;
59          }
60
61          public int getInt(){
62                      return _Integer;
63          }
64
65          public double getFloat(){
66                      return _Float;
67          }
68
69          public boolean getBool(){
70                      return _Bool;
71          }
72
73          public boolean equalTo(Data other){
74                  if(other.type == INT_TYPE)
75                      return this._Integer == other._Integer;
76                  else if(other.type == STRING_TYPE)
77                      return this._String.equals(other._String);
78                  else if(other.type == BOOL_TYPE)
79                      return this._Bool == other._Bool;
80                  else if (other.type == FLOAT_TYPE)
81                      return this._Float == other._Float;
82                  else
83                      return this._Date.equals(other._Date);
84          }
85
86          public String toString(){
87                  if(type == INT_TYPE)
88                      return ""+_Integer;
89                  if(type == STRING_TYPE)
90                      return _String;
91                  if(type == FLOAT_TYPE)
92                      return ""+_Float;
93                  if(type == FLOAT_TYPE)
94                      return ""+_Bool;
95                  return _Date.toString();
96          }
97
98          public Data copyOf(){
99                  if(type == INT_TYPE)
100                     return new Data(_Integer);
101                 else if(type == STRING_TYPE)
102                     return new Data(new String(_String));
103                 else if(type == BOOL_TYPE)
104                     return new Data(_Bool);
105                 else if(type == FLOAT_TYPE)
106                     return new Data(_Float);
107                 else
108                     return new Data(_Date.toString());
109         }
110
111 }
```

8.2.12 CsvInterpreter.java

```
1   // Michael Vitrano and Jared Pochtar
2
3   package rtlib;
4
5   import java.io.BufferedReader;
6   import java.io.File;
7   import java.io.FileReader;
```

```
8   import java.io.FileWriter;
9   import java.util.StringTokenizer;
10
11
12  public class CsvInterpreter {
13
14          public static Table toTable(String filename, UserDefinedType type) {
15                  File file = new File(filename);
16                  BufferedReader reader;
17
18                  try{
19                           reader = new BufferedReader(new FileReader(file));
20
21                  Table table = new Table(type);
22                  String line = null;
23
24                  int dataTypes[] = type.typeList();
25
26                  while( (line = reader.readLine())!= null){
27
28                          StringTokenizer st = new StringTokenizer(line, "," );
29                          int i = 0;
30                          Record newRow = new Record(type);
31                          while(st.hasMoreTokens() && i < dataTypes.length){
32
33                                  try{
34                                  if(dataTypes[i] == Data.BOOL_TYPE){
35                                          newRow.addData(new Data(Boolean.getBoolean(st.nextToken())));
36                                  }
37                                  else if(dataTypes[i] == Data.FLOAT_TYPE){
38                                          newRow.addData(new Data(Double.parseDouble(st.nextToken())));
39                                  }
40                                  else if(dataTypes[i] == Data.INT_TYPE){
41                                          newRow.addData(new Data(Integer.parseInt(st.nextToken())));
42                                  }
43                                  else
44                                          newRow.addData( new Data((st.nextToken())));
45                                  }catch(Exception e){
46                                       e.printStackTrace();
47                                                  return new Table(type);
48                                  }
49
50                                  i++;
51                          }
52                          table.addRow(newRow);
53                  }
54
55                  return table;
56                  }catch(Exception e){
57                          return new Table(type);
58                  }
59          }
60
61          public static Table toFile(String filename, Table table) {
62                  File file = new File(filename);
63                  FileWriter writer;
64                  try {
65                          writer = new FileWriter(file);
66                          writer.write(table.toFile());
67                          writer.close();
68                  }catch(Exception e){
69                          System.out.println("Bad file: " + filename);
70                          return null;
71                  }
72                  return table;
73          }
74
75
```

```
76
77
78  }
```