# TaML

Table Manipulation Language
Final Report

Adam Dossa (aid2112)
Qiuzi Shangguan (qs2130)
Maria Taku (mat2185)
Le Chang (lc2879)

*Columbia University*
*19th December 2012*

# Table of Contents

# 1 Introduction

The purpose of TaML is to provide users with an efficient and simple language for building, editing, and manipulating tables (commonly known as spreadsheets). For example, through only a few simple lines of code, a user can create a table, populate the table with data, edit the data, and perform mathematical operations on this data (summing values in various cells etc.). As one could imagine, performing operations such as this in Java or C++ could take hundreds of lines of code!

At a higher level, the functionality of this language could be used to allow programmers to quickly and efficiently manage budgets, calculate yearly taxes, or otherwise keep track of various types of numerical data and the relationships between this data.

# 2 Language Tutorial

TaML is a simple C-like language which you can use for efficient table/spreadsheet manipulation. It has the built-in types Table, Line and Cell which used to represent the main taml paradigm, namely a table that contain either integers or floats. It is worth noting that although TaML is statically typed at a high level, since Tables, Lines and Cells hold underlying values of either integers or floats, these types can behave like generic types.

## 2.1 Lexical Conventions

### 2.1.1 Basic types

Let's begin by introducing three built-in types of TaML: Cell, Line, and Table.

Variable declaration:

```
table intTable = ([10,10],int);                  # declare a int Table of size 10*10
table c_intTable= intTable[0~5,0~5];# initialize another Table of  size 6*6 that equals the upper left part of
                                    # intTable
line  sec =t[2,@];                  # initialize a Line that references row 2 of intTable
cell  cel = sec[1];                 # initialize a Cell that references cell 1 of sec
```

There are also other commonly used types such as char, string, and bool which have a similar declaration style:

```
char c='9';
bool b= true;
string hello="hello world";
```

The sample output of a Table with the underlying type "int" is shown below:

```
      A     B     C
      ==============
1     | 1   | 2   | 3  |
2     | 11  | 12  | 13 |
3     | 21  | 22  | 23 |
4     | 31  | 32  | 33 |
5     | 41  | 42  | 43 |
6     | 51  | 52  | 53 |
      ==============
```

### 2.1.2 The main function

The main function is the entry point of a TaML program

```
func void main(){
        # variable declarations
        # function calls
}
```

Other functions have the same declaration style:

```
func table reverse_T(Table origin_T, int a){
        # variable declarations
        # function calls
}
```

### 2.1.3 Your first program

Now let's begin writing a TaML program which initializes a float table, assigns 1 to all the cells in the table and then prints it:

```
func void main(){
    table float_T = ([10,10],float);
    ^float_T=1.0;
    print(float_T);
}
```

The print function takes a variable of any type (int, float, string, char, bool, cell, line, or table) and prints it. In this case it prints a Table.

### 2.1.4 Range operators

The range operators give an efficient way to copy from or assign values to a Table/Line/Cell:

**~ :** *line newLine = myLine[1~3];* #assign the cells 1 to 3 in *myLine* to *newLine*
**@ :** *table copy= t[@,@];* #copy table *t* to table *copy*

### 2.1.5 Caret operator

The caret operator is used to extract the underlying value of a Table/Line/Cell expression:

*^cel = 6;* #assign 6 to the value of Cell *cel*
*^int_Table[1,5] = 7;* #assign 7 to the value of the Cell at *int_Table[1,5]*

### 2.1.6 Installation and Compilation

To compile TaML source code, make sure you have a 1.6+ Javac version and download the built-in Java_lib (pre-compiled TaML library code) to your computer.

To compile and run a file named “myEample.taml” : ./run.sh  myExample.taml

### 2.1.7 Implementation and dynamic view of the game of life:

```
int r = 30;
int c = 30;
int T = 50;

func table getInitialBoard(int a, int b){
        table board = ([a,b], int);
        ^board = 0;
        ^board[12,11] = 1;
        ^board[12,12] = 1;
        ^board[11,12] = 1;
        ^board[12,13] = 1;
        ^board[13,13] = 1;
        return board;
}

func int getNeighbour(table t, int a, int b){
        int res;
        if (a == -1) {
                a = 0;
        }
        if (b == -1) {
                b = 0;
        }
        if (a > (r - 1)) {
                a = r - 1;
        }
        if (b > (c - 1)) {
                b = c - 1;
        }
        res = ^t[a,b];
        return res;
}

func int getNumberNeighbours(table t, int a, int b){
```

```
        int res = 0;

        res = res + getNeighbour(t, a-1, b+1);
        res = res + getNeighbour(t, a-1, b);
        res = res + getNeighbour(t, a-1, b-1);
        res = res + getNeighbour(t, a, b+1);
        res = res + getNeighbour(t, a, b-1);
        res = res + getNeighbour(t, a+1, b+1);
        res = res + getNeighbour(t, a+1, b);
        res = res + getNeighbour(t, a+1, b-1);

        return res;
}

func void updateCell(cell c, int n){
        if (^c == 1) {
                if ((n < 2) || (n > 3)) {
                        ^c = 0;
                }
        } else {
                if (n==3) {
                        ^c = 1;
                }
        }
}

func void main(){
        table board = getInitialBoard(r,c);
        print(board);
        int a;
        int b;
        int i;
        for (i = 0; i < T; i = i + 1){
                table nBoard = ([r,c],int);
                ^nBoard = 0;
                for (a = 0; a < 30; a = a + 1) {
                        for (b = 0; b < 30; b = b + 1) {
                                int neighbours;
                                neighbours = getNumberNeighbours(board,a,b);
                                ^nBoard[a,b] = neighbours;
                        }
```

```
            }
            for (a = 0; a < 30; a = a + 1) {
                for (b = 0; b < 30; b = b + 1) {
                    cell c = board[a,b];
                    int n = ^nBoard[a,b];
                    updateCell(c,n);
                }
            }
            print(board);
        }
}
```

# 3 Language Reference Manual

## 3.1 Lexical Conventions

### 3.1.1 Character Set

TaML uses the ASCII character set, and assumes all input streams are in this character set.

### 3.1.2 White Space

As in the C language, whitespace (defined as any of blanks, tabs, newlines and comments), is ignored except as it serves to separate tokens. White space is required to separate any tokens of the above types which would otherwise be adjacent.

If the input stream has been parsed into tokens up to a given character, we adopt a greedy parsing approach and the next token it taken to include the longest possible string of characters that could possibly constitute a token.

### 3.1.3 Comments

A comment is introduced by a # character, and the comment is taken to be the text to the right hand side of the # character (including the # character) until the closest following new line character in the input stream, or the end of the input stream, whichever occurs first. Specifically an occurrence of an additional # character within a comment as defined above is ignored (the # is taken as part of the comment), and as such comments do not nest, or span multiple lines.

Note that a # cannot form part of a string literal or be a character literal, so its sense is not ambiguous in these contexts.

### 3.1.4 Token Types

Token types in TaML can be one of the following:

I. Identifiers
II. Keywords
III. Constants (literals)
IV. Operators
V. Functions

### 3.1.5 Identifiers

An identifier is a sequence of letters, digits, and underscores (_). The first character must be a letter. Uppercase and lowercase letters are distinct. Identifier length is unlimited.

### 3.1.6 Keywords

Table 3.1.6 lists the set of identifiers that are reserved for keywords in TaML. These identifiers cannot be used for any other purpose.

| | | | | | |
|---|---|---|---|---|---|
| bool | else | func | NULL | table | while |
| break | false | if | print | true | |
| cell | for | int | return | then | |
| char | float | line | string | void | |

*Table 3.1.6: Keywords in TaML*

### 3.1.7 Constants

There are several types of constants allowed in TaML. Every constant has a specific type associated with it of either int, float, char, string or bool. Int, float, string, and char can also be associated with the special constant NULL, which indicates an unset or undefined value. NULL must still have a type (of either int, float, char or string) associated with it.

#### *3.1.7.1 Integer Constant*

An integer constant is a sequence of digits which is optionally signed by a preceding negation character "-". All integer constants are considered to be decimal and of type int.

#### *3.1.7.2 Float Constant*

A floating constant consists of an integer part, a decimal point, a fraction part, an e, and an optionally signed integer exponent. The integer and fraction parts both consist of a sequence of digits. Either the integer part or the fraction part (but not both) can be missing. Either the decimal point or the e and the exponent (not both) can be missing. All float constants are considered to be decimal and of type float.

#### *3.1.7.3 Character Constant*

A character constant is a single character enclosed in single quotation marks, such as 'x'. Note the character can be any member of the 52 upper and lower case standard English alphabet, one of the digits from 0 to 9, any generally accepted English keyboard symbols such as '!', '@', etc., or '\r', '\n'. The only character a char is not allowed to be is the single quote (‘). Empty characters are not allowed. Every character constant is considered to be of type char.

#### *3.1.7.4 String Constant*

A string constant is a sequence of characters surrounded by double quotation marks, as in "cat". As in character constants, the sequence of characters can include any of the 52 upper and lower case standard English alphabet, any digits, any generally accepted English keyboard symbols, and any of '\r', '\n'. The only character not

accepted by a string is a quotation mark (") or escaped quotation mark (\"). Unlike char constants, an empty string is allowed. Every string constant is considered to be of type string.

*3.1.7.5 Boolean Constant*

A boolean constant can be either the token “true” or the token “false.” Every boolean constant is considered to be of type bool.

# 3.2 Types

TaML has two main types -- general types and table types. Explicit conversions between types (e.g., casting) is not allowed in TaML.

## 3.2.1 General Types

The general types consist of integers, floating point numbers, characters, strings, and booleans. The size, range, and default value for each is as specified in Table 3.2.1.

| **Primitive Type** | **Size** | **Range** | **Default Value** |
|---|---|---|---|
| Integer | 4 bytes | -2,147,483,648 to 2,147,483,647 | NULL |
| Floating Point | 4 bytes | 1.40129846432481707e-45 to 3.40282346638528860e+38 | NULL |
| Character | 1 byte | ASCII | NULL |
| String | ≥ 1 byte | One or more ASCII | NULL |
| Boolean | 1 byte | true, false | false |

*Table 3.2.1: General Data Types Information*

*3.2.1.1 Integers*

Integers in TaML are always signed and are represented by 4 bytes. This results in a range of roughly -2 million to 2 million for integers. The default value of an integer is set as NULL.

*3.2.1.2 Floating Point Numbers*

Floating point numbers in TaML are represented by 4 bytes, resulting in a range of roughly 1.40129846432481707e-45 to 3.40282346638528860e38. The default value for a floating point is NULL.

*3.2.1.3 Characters*

Each character is represented by 1 byte, and has a default value of NULL.

Characters consist of single characters as defined above in section 3.1.7.3. Empty characters are not allowed. For example, *char c = '';* is not allowed syntax.

In TaML, characters are not allowed to be used in Tables (tables only accept integers and floats), and thus the main use of chars and strings in TaML is for output to a terminal or other output destination.

*3.2.1.4 Strings*

Strings consist of zero or more characters as defined above in section 3.1.7.4. Unlike char, an empty string is allowed. The size of the string is dependent upon the number of characters in the string. The default value of a string is NULL.

In TaML, strings are not allowed to be used in Tables (tables only accept integers and floats), and thus the main use of chars and strings in TaML is for output to a terminal or other output destination.

*3.2.1.5 Booleans*

Booleans values are 1 byte and consist of either a "true" or "false" value. The default value for a boolean is false.

In TaML, bools are not allowed to be used in Tables (tables only accept integers and floats), and thus the main use of bools in TaML is for comparison logic.

### 3.2.2 Table Types

In addition to the general types, TaML also provides for several "table types" that are specific to the TaML language. These table types are the cell, the line, and the table. The default values, memory allocations, and example syntax for initializing these types are shown in Table 3.2.

In general, the table types can provide more functionality than the general types mentioned in section 3.2.1. For example, a cell type may not only represent a certain value (such as an integer value held by that cell) but also a reference or other means of associating that cell with a particular coordinate in the table.

| **Table Type** | **Example Initialization Syntax** | **Default Value** | **Memory Allocation** |
|---|---|---|---|
| Table | *table t = ([3,4], int);*<br><br>creates a table type called "t" with 3 rows and 4 columns<br><br>**Note:** Tables are limited to a max of 50 columns and 500 rows. | NULL | Upon creation, the table is allocated a portion of memory that is large enough to accommodate all cells in the tables as well as any supplementary information for managing the table. |
| Line | *line r = t[3,@];*<br><br>Creates a line called "r" that holds all cells in row 3 of table "t" (e.g., r | NULL | Lines are composed of a 1d array of cells. These cells are references to one or more cells in a table (e.g., and are not |

| | | | |
|---|---|---|---|
| | represents a row)<br><br>*line c = t[@,2];*<br><br>Creates a line called “c” that holds all cells in column 2 of table “t” (e.g., c represents a column)<br><br>*line sc = t[2~5, 2];*<br><br>Creates a line called "sc" that holds all cells in column 2 from rows 2 to 5 inclusive.<br><br>Note: The @ and ~ operators are discussed in more detail in section 3.2.2 | | personally allocated memory space.) |
| Cell | *cell c = t[1,2];  # where "t" is a table*<br><br>Creates a cell called “c” that corresponds to row 1, column 2 of table “t”<br><br>*cell c = l[2]  # where "l" is a line*<br><br>Creates a cell called "c" that corresponds to cell 2 in line "l". Line l itself can refer to one or more cells of a table. | NULL | Cells are references to a cell in a table (or are a reference to a cell in a line, which is itself a reference to a cell in a table), and in this case are not personally allocated memory space. |

*Table 3.2.2: Table Types - default values and example initialization Syntax*

*3.2.2.1 Table*

A table in TaML consists of one or more cells arranged in rows and columns. The default value of a table that has not yet been initialized is NULL. In order to create and initialize a table, the number of rows, number of columns, and numeric type of the table should be specified. The syntax is:

*table tableName = ([number_of_rows, number_of_columns], data_type);*

Where the *data_type* is either “int” or “float.” For example, to create a table named “myTable” with 3 rows and 4 columns and which holds integers, you would define:

*table myTable = ([3,4], int);*

The values stored in the cells of the table must all be consistent. For example, if *myTable* is initialized to hold integers, then all cells in *myTable* must hold integers.

Lastly, TaML only allows for creating 2-dimensional tables. However, a 1-dimensional table can be mimicked by setting either the number of rows or columns to 1. For example, a table with 1 row of floats can be mimicked by using:

```
table myTable = ([1,10], float);
```

TaML does not allow for creating or "mimicking" tables with 3-dimensions or greater. Furthermore, the size for both the rows and columns must be strictly greater than 0.

Moreover, tables in TaML are limited to a maximum number of 50 columns and 500 rows.

In order to manipulate the values of cells in a table, the caret operator, ^, can be used. This operator is discussed in more detail in section 3.3.6.1 of this Language Reference Manual.

*3.2.2.2 Line*

A line in TaML consists of a 1d array of cells. These cells are references to a contiguous line of one or more cells in a table. In other words, a line can be equal to either a column or a row in the Table, or to a contiguous subset of a column or row in a table.

The default value of a line that has not yet been initialized is NULL. To create and initialize a line that represents a row or column, respectively, the following syntax can be used:

```
# Create a "row" line
line lineName = tableName[row_number, startIndex~endIndex];

# Create a "column" line
line lineName = tableName[startIndex~endIndex, column_number];
```

The tilde ("~") is an operator in TaML which indicates a range of contiguous cells. For example, to create a "row" line called "myLine" that represents cells 2 through 4 (inclusive) in row 5 of the table called "myTable" you could define:

```
line myLine = myTable[5,2~4];
```

The ampersand operator ("@") can be used in place of the tilde syntax to indicate an entire row or an entire column in a table. In other words, the syntax [@,3] indicates all cells from cell 0 to the last cell in column 3. Or, to put this in layman's terms, [@,3] merely indicates column 3.

Thus, as an example, the syntax for creating a line called "myLine" that represents row 4 in the table called "myTable" you could define:

*line myLine = myTable[4,@];*

Note that when an initializing a line, exactly one of the column or row values must include a tilde or ampersand. For example, the following line initializations are not allowed:

*#NOT allowed: Either the row or column must have a tilde or ampersand*
*line myLine = myTable[4,5];*

*#NOT allowed: Both the row and column cannot have a tilde/ampersand*
*line myLine = myTable[4~5,1~5];*
*line myLine = myTable[@,1~5];*
*line myLine = myTable[4~5,@];*

In order to manipulate the values of cells in a line, the caret operator, ^, can be used. This operator is discussed in more detail in section 3.3.6.1 of this Language Reference Manual.

*3.2.2.3 Cell*

A cell type represents the basic unit of a table. The default value of a cell that has not been initialized is NULL.

Each cell can represent one coordinate in the table (e.g., is a reference to a cell in a table) or one coordinate in a line (e.g., is a reference to a cell in a line).

As mentioned above, all cells in a table or line should represent the same type of data. For example, if one cell in a table represents an integer, then all cells should hold an integer. However, note that a cell with value NULL will not clash with the other cells in the table. For example, if a table has cells representing integers, then a cell with value NULL is also allowed to be in the table.

A cell can be created and initialized with the syntax:

*cell cellName = tableName[row_number, column_number];*

For example, to create a cell named “c” that represents the third row and fourth column of the table named “myTable” you could define:

*cell c = myTable[3,4];*

A cell may also be created and initialized based on a line:

*cell cellName = lineName[cell_number];*

For example, to create a cell named “c” that represents the third cell of the line named “myLine” you could define:

*cell c = myLine[3];*

Note that the third column of *myLine* is itself a reference to a cell in a table. And thus the value of the third column of *myLine* (and therefore also the value of "c") is determined by this cell of the table.

In order to manipulate the values of cells, the caret operator, ^, can be used. This operator is discussed in more detail in section 3.3.6.1 of this Language Reference Manual.

# 3.3 Expressions

## 3.3.1 Primary Expressions

There are three types of primary expressions: identifiers, constants, and parenthesized expressions.

### *3.3.1.1 Identifiers*

Identifiers represent variables or functions.

### *3.3.1.2 Constants*

Constants can be divided into five categories: integer, float, character, string and boolean. The specific definitions for each can be seen in Section 3.1.7.

### *3.3.1.3 Parenthesized Expressions*

Parenthesized expressions have the same value as the expression without parentheses. The function of the parentheses is to change the inside expression's precedence in the process of evaluation.

## 3.3.2 Assignment Operator

In our language, "=" is the assignment operator. The function of an assignment is to pass a value to the identifier on the left side of the expression. Assignment operators are binary and right-associative. The left operand must be a "left value." We define legal left values in our language as variables, where the type of the variable is either line, cell, int, float, bool, char or string. The right operand must be an expression. The types of the two operands needs to be consistent. Once the assignment statement is taken, the left operand will have the same value as the right expression, and the expression returns the value of the left operand. We note that consistent here usually means "of the same type", with the exception of where a cell is the left hand side of an assignment operation, in which case it is defined as in section 3.2.2.3.

## 3.3.3 Arithmetic Operators

Arithmetic operators include "+", "-", "*", "/" and "%".

+ → addition

| | | |
|---|---|---|
| - | → | subtraction |
| * | → | multiplication |
| / | → | division |
| % | → | mod |

All of the operators above are binary and left-associative.

For all operators, both operands must be of the same type, since no implicit casting is allowed. Furthermore, all operands associated with arithmetic operators must be of type int or float.

### 3.3.4 Comparative operators

Comparative operators are “>”,”<”, “>=”, “<=”, ‘“==” and “!=”. All of the comparative operators are binary and left-associative. The two operands on either side of a comparative operator(“>”, “<”, “>=”, “<=”) must be of the same type and must be of type int or float. The two operands for operator “==” and “!=” can be of type int, float, or bool and must be type consistent.

A comparative expression returns a boolean value which indicates the predicate.

### 3.3.5 Logical operators

Logical operators in our language includes “&&”, ‘||’ and “!”.

| | | |
|---|---|---|
| && | → | logical and |
| \|\| | → | logical or |
| ! | → | logical not |

“&&” and “||’ are binary and left-associative. The operands on both sides of the operator must be of boolean type. “!” is unary operator and the operand after “!” must be a boolean type as well. An expression involving logical operators returns a boolean value.

### 3.3.6 Table operators

*3.3.6.1 Value access operator '^'*

In TaML, the cell, line, and table types are in essence a "dual meaning." These types can either refer to the entities themselves, or can refer to the *value held by the cell, line, or table.* In order to differentiate between whether the entity itself or the value is desired, the caret operator '^' is used.

For example, assuming we have two cells named *c1* and *c2*, then we can have the following cases:

| | |
|---|---|
| *c1 = c2*; | These cells themselves are set to be equal. They point to the same place in memory. |

*^c1 = c^2;* The *value held* by c1 is set equal to the value held by c2. In other words, c1 and c2 both have their own distinctive memory allocations. But both cells hold the same (integer or float) value.

As you can see from these examples, the caret operator thus servers to access the value held by a cell, line, or table. Some more examples are shown in the table below.

| **Sample Program** | **Notes on the syntax meaning** |
|---|---|
| *Table t1 = ([10,10], int);* | Declaring a table. |
| *cell c1 = t1[0,0];*<br>*line l1 = t1[@,1];* | Cell c1 and Line l1 are set equal to cell(s) in table t1. In other words, they are now references to cells in t1. |
| *^c1 = 2000;* | The value of c1 is changed to 2000. This will also change the value of t1[0,0] since c1 is a reference to this cell. |
| *^l1 = 4000;* | All values in l1 are set to 4000. This will also change the values in column 1 of t1 to 4000, since l1 is a reference to these values. |
| *^l1[0] = 5000;* | The value of the first cell in l1 is set equal to 5000. This will also change the value in t1[0,1], since l1[0] is a reference to this cell of table 1. |
| *^c1 = ^t1[1,1];* | The value of c1 is now changed to the value of t1[1,1]. Note, however that c1's still references t1[0,0]. In other words, this serves to set the value of t[0,0] equal to t[1,1] |
| *c1= t1[2,2];* | c1 now instead references t1[2,2]. The value held by c1 will thus be the value at t1[2,2]. Any changes in the value of c1 (e.g., by ^c1 = 2000;) will now change the value at t1[2,2] |
| *if( ^c1 > ^l1[1]) {*<br>*… do something*<br>*}* | To use the values of cells/lines/tables in comparisons or arithmetic, the ^ operator should be used to access the underlying value. |
| *print(c1);* | The TaML print function does not require the caret operator. This will print the value held by c1. |

*Table 4.6: Example usage of the ^ operator*

*3.3.6.2 Table access operator '[]'*

**<u>Case 1:</u> table1[m,n]**

m and n are expressions that evaluate to positive integers (greater than or equal to 0). In this case, *^table1[m,n]* returns the value of the coordinate at row m, column n. Note that *table[m,n]* without the ^ operator would return the cell at row m, column n, as discussed in section 3.3.6.1.

```
e.g,    cell c = table1[2,3];        # c references the cell at table1[2,3]
        int a = ^table1[m,n] ;       # Assigns the value of table coordinate [m,n]
                                     # to the variable "a"
        ^table1[m,n] = 2000;         # Sets the value of table coordinate [m,n] to 2000
```

**Case 2: table1[m1~m2, n] -- table1[m, n1~n2] – table1[@, n] – table1[m, @]**

If one of m or n is in the format of "integer~integer" or is an '@', and the other is a single integer, then a line is returned.

```
e.g.,   line line1=  table1[2~3,5];  # line 1 references a line of cells running from rows 2 to 3,
                                     # in column 5 of table 1
        line line2 = table1[3, @];   # line 2 references the cells in row 3 of table1
```

**Case 3: table1[m1~m2, n1~n2] -- table1[@,@]**

If m and n both are in the format of "integer~integer" or are both @, then a table is returned.

```
e.g.,   table t2=  table1[2~3,5~6];  # t2 is a copy of a subset of table1
        table t2 = table1[@,@];      # t2 is a copy of the entire table1
```

**Case 4: table1 (without a table accessor identifier)**

This syntax can be used to set the value of an entire table at once. For example:

```
e.g.,   t2 = 4000;                   # ALL cells in table t2 are set to 4000
```

*3.3.6.3 Line access operator '[]'*

**Case 1: line1[m]**

m is an expression that evaluates to a positive integers (greater than or equal to 0). In this case, *^line1[m]* returns the value of the coordinate at index m. Note that line*[m]* without the ^ operator would return the cell at index m,as discussed in section 4.6.1.

```
e.g.    int a= ^line1[2];            # Assigns the value of line1[2] to int "a"
        ^line1[2]=1000;              # The value of line[2] is set equal to 1000
        cell c1 = line1[0];          # c1 now references the cell at line1[0]
```

**Case 2: line[m1~m1] – line[@]**

If m is in the format of "integer~integer" or is an '@', line1[m] returns a line.

e.g., *line line2 = line1[2~3];* *# line2 references the cells from index 2 to 3 of line1*
*line line2 = ^line1[@];* *# line2 is a copy of the entire line1*

**Case 3: line (no line access operator)**

This syntax can be used to set the value of an entire line at once. For example:

*e.g., line2 = 4000;* *# ALL cells in line2 are set to 4000*

*3.3.6.4 The tilde (~) operator*

1) If one of m and n is in the format of "integer~integer", table1[m,n] returns a line.

*line r = table1[1,2~3]*

2) If both m and n are in the format of "integer~integer", table [m,n] returns a table. In this case we can initialize a new table using this approach, with the new table inheriting the type from the table from which it is being initialized.

*table table2 = table[1~2,3~4]*

*3.3.6.5 The at (@) symbol*

1) The symbol @ represents all the cells in a table's row or columns. For example, table1[@,3] returns all cells in column 3. As another example, table1[4,@] returns all cells in row 4.

In other words, all of the expression above mean we want to access the value of specific cells in table. The operator '[]' thus takes three operands: object-name, row-num and column-num. Object-name must be either of type table or of type line. The row-num and column-num values must be integers, an @, or in an "integer~integer" format.

## 3.3.7 Precedence Rules

To avoid ambiguity, we have defined the precedence of operators as below. The operators have on top have higher priority than those below it.

| | |
|---|---|
| 1 | () |
| 2 | ! |
| 3 | * / % |
| 4 | + - |
| 5 | > < <= >= |
| 6 | != == |
| 7 | && |
| 8 | \|\| |
| 9 | = |

# 3.4 Statements

## 3.4.1 Variable declarations and initialization statements

*3.4.1.1 Type Specifiers*

In TaML, Type specifiers are as follows:

*int :* *integer numbers*
*float:* *floating point numbers*
*char:* *characters*
*string:* *strings*
*bool:* *boolean values*
*cell:* *a cell unit*
*line:* *a horizontal or vertical 1d array of cells*
*table:* *a 2d table of cells*

*3.4.1.2 Variable Declarations*

Variable declarations are treated as statements in TaML. Declarations have the form of :

*Type_Specifier identifier_list;*

Where an *identifier_list* can be either a list of identifiers, or a single identifier:

*identifier_list* → *identifier_list* | *identifier*

For example, to declare two tables at once:

*Table a , b;*

*3.4.1.3 Variable Initialization*

Variable initializations are treated as statements in TaML.

To declare and initializations a variable at the same time, the following syntax can be used:

*Type_Specifier identifier_list = intialization_value;*

Where, once again, the *identifier_list* can be either a list of identifiers, or a single identifier. For example, to declare two tables at once:

*Table a , b = ([m,n],int);*

#### *3.4.1.4 Initialization and Declaration of Table Types*

The specific syntax for declaring and initializing the table types (i.e., cells, lines, and tables) is summarized in Table 3.2.2 and is discussed in detail in Section 3.2.2.

### 3.4.2 Expression statements

In TaML, most statements are expression statements which have the form of

*expression;*

Every expression can be used here. Note there is a semicolon at the end of the expression.

### 3.4.3 Selection statements

Selection statements have the following two forms in TaML:

*if (controlling_expression) block_statement*
*if (controlling_expression) block_statement else block_statement*

Selection statements choose one of a set of statements to execute, based on the evaluation of the *controlling_expression*.

The *controlling_expression* of an *if* statement must have scalar type.

For both forms of the *if* statement, the first statement is executed if the *controlling_expression* evaluates to true. For the second form, the second statement is executed if the *controlling_expression* evaluates to false. To avoid ambiguity, an *else* clause that follows multiple sequential else-less *if* statements is associated with the most recent *if* statement in the same block (that is, not in an enclosed block).

### 3.4.4 Block statements

Block statements are a list of zero or more *statements* and zero or more *declarations* surrounded by braces:

*{*
*declaration(s) and/or statement(s)*

*}*

Note that each statement and declaration must end in a semicolon.

Also note that a *break* statement can be used to transfer control outside of a block statement. The *break* statement is discussed in more detail in section 3.4.7.

### 3.4.5 Iterative statements

TaML supports both the *for* and *while* iterative statements. The *for* loop takes the form:

*for (first_expression; second_expression; third_expression) block_statement*

the *first_expression* in the iterative statement is evaluated once at the beginning of the first iteration, while the *second_expression* is evaluated every time before the execution of the *block_statement*. If the *second_expression* is equal to false, the iteration terminates and the *block_statement* is not executed. Otherwise, if the *second_expression* evaluates to true, the *block_statement* will be executed. The *third_expression* is evaluated at the end of each execution of the *block_statement* and re-establishes the conditions for the next loop (e.g., for determining whether *first_expression* is still satisfied).

The *while* loops takes the form:

*while ( control_expression ) block_statement*

The *control_expression* must be of type boolean. The *while* statement is executed by first evaluating the *control_expression*. Execution then continues by making a choice based on the resulting value of the *control_expression:*

1. If the value is true, then the contained *block_statement* is executed.
2. If the value of the *control_expression* is false, no further action is taken and the *while* statement completes normally.

### 3.4.6 Return Statements

You can use the return statement to end the execution of a function and return program control to the function that called it. Here is the general form of the return statement:

*return return_value;*

The *return_value* can be any of the allowed return types:

*int*
*float*
*string*
*char*
*bool*
*cell*
*line*
*table*
*void*

Note that when the return type is *void,* the value of *return_value* is empty. In other worse, in this case the proper syntax is simply:

*return;*

Alternatively, when the return type is *void*, then the function is not required to have a return statement. For example, in TaML the following function would be allowed:

```
func void noReturnExample(){
        print(39 + 3);
}
```

### 3.4.7 Break Statements

A break statement transfers control out of an enclosing block statement. The syntax for a break statement is simply:

*break;*

Note that in TaML, labels are not allowed after a break statement; instead, as mentioned above, control of the program is simply transferred outside of the enclosing block statement.

Thus, a break statement attempts to transfer control to the innermost enclosing while or for statement. This innermost enclosing statement then immediately completes normally.

## 3.5 Functions

### 3.5.1 Function Declarations

In TaML, it is possible to specify a named function, which can then subsequently be referenced via its identifier.

A function is declared using the reserved keyword func as follows:

*func return_type function_identifier (parameter_list) statement_block*

*return_type* can be one of either int, float, string, char, bool, cell, line, table, or void and represents the type of the value that is returned by a call to a given function

*function_identifier* must conform to the same format as identifiers (as in section 2.5). Two functions cannot be declared with the same identifier, nor can an identifier be used which matches an identifier used to represent a variable elsewhere in the program.

*parameter_list* is a (possibly empty) list of parameters that the function takes as arguments. The parameter list is a comma separated list of type and identifier pairs, for example:

```
int a, int b, float c
```

*statement_block* is a block statement as defined in 3.4.4. Within such a block statement, if the return type of the function is not void, there must be a return statement, which returns a value of the appropriate type.

An example function declaration:

```
func int myFunction(int a, int b) {
        int c;
        c = a + b;
        return c;
}
```

### 3.5.2 Nested Functions

In TaML, nested functions are not allowed.

### 3.5.3 The Main Function

Execution begins in a method named "main" in TaML, so there must exist one and only one main function. The main function should have the following format:

```
func void main( ){}
```

As illustrated above, the *return_type* of the main function must be of type void.

### 3.5.4 Built-In Functions

There is one built-in function, "print" in TaML. Print does not return any value. The print function takes exactly one argument, which can be of type: int, float, bool, string, char, table, line or cell.

When the type of parameter is int, float, bool, char, string or cell, the print function will print the value associated with its argument on the screen and move to a newline. If the argument is of type table or line, the output will be as followings:

```
>>print(line_1)
          A       B       C       D       E       F   …..
          ========================================
      1 | 1000  |  2000  | 3000  |  4000  | 5000  |  6000

>>print(table_1)
            A     B       C       D       E       F   …..
          ========================================
      1 | 1000  |  2000  | 3000  |  4000  | 5000  |  6000
      2 | 2000  |  3000  | 4000  |  5000  | 6000  |  7000
      3 | ……
      4 |
      5 | ……
      6 | 6000  |  7000  | 8000  |  9000 |  1000  |  2000
      7 | ……
```

As mentioned previously, tables can have up to 500 rows and up to 50 columns. In other words, the row indices in the print(table_1) function can go from 1 to 500, and the column indices can go from A to XX. Similarly, when a line is printed it has a single row index of 1, and the column indices may go from A to XX.

## 3.6 Scope

In TaML, there are two kinds of lexical scopes for variables: local and global.

Any variable declared outside of a function or statement block has global scope; such a variable will be visible starting from its declaration until the end of the program file.

Any variable declared within a function or statement block has local scope and is only visible within that function or statement block respectively. In other words, a local variable of the same name declared elsewhere is a different variable.

# 4 Project Plan

## 4.1 Project Process

Our technical approach to team management for our language design and implementation was to employ a collaborative process, which maximized our ability to work independently within a cohesive framework that guided overall progress.

The key components in this strategy were:

1. An online group schedule around which we could organize any required meetings.
2. The use of git / github as a repository and version control infrastructure.
3. Regular meetings as a group, and sufficient scheduled meetings with our assigned TA.
4. Focusing our design on an approach, which allowed us to work independently on at least some of the steps.
5. Generating a set of milestones early on the project and having target dates associated with each milestone against which we could measure our progress as a whole.
6. Having a flexible approach to work allowing people to work at their own pace, whilst maintaining an overall momentum through the agreed milestone framework.

Identified Milestones were:

a. Language Reference Manual
b. AST design
c. Test framework design
d. AST implementation
e. Test framework implementation
f. Static semantic checking design
g. Static semantic checking implementation
h. Java printer design
i. Java printer implementation
j. Test cases for all stages of the compilation process
k. Final Project / Presentation

After some creative brainstorming around possible languages, we chose TaML as a pragmatic option that combined some interesting language features within a well-understood paradigm (that of a spreadsheet) with being conceptually simple enough that we felt it was a reasonable objective to implement within the time constraints. We also decided to use JAVA as the compilation language for TaML reasoning that this allowed us to focus on the language design and implementation.

Having decided on our language, generating the Language Reference Manual was the first focus of the group. Having held several meetings during which we agreed on all of the main points of the language, we all

participated in writing different sections of the manual. Following an iterative approach we then met to discuss the combined document, with each team member leading the discussion on their section, and then individually worked on making the relevant updates to ensure consistency across the whole document where issues were identified. This process was repeated until we were happy with our final LRM. We also took the opportunity to create the first iteration of our scanner / parser for our language.

The next group stage was to agree on a high-level approach for our AST representation and our testing framework through group discussion. We decided to split into two sub-teams, one focusing on the implementation of the AST and the other on the implementation of our testing framework. By identifying two sub-components for this stage of the development we were able to effectively parallelize our efforts whilst maintaining some of the advantages of working within a team that were critical given the unfamiliar development environment.

Once this had been completed we took a similar approach for the next stage. We determined that it would be relatively straightforward to work on the semantic checking of our AST and creation of an SAST in parallel to creating the process which prints the SAST to JAVA. This was possible as the SAST was very similar to the AST, except for the additional type information attached to each expression.

We decided early on to try and test each component of our translator independently. This meant having the option to output each stage of our translation process from our translator in a format that lent itself to comparison to a golden source. We decided a directory structure, naming convention and language (shell script) for our testing environment early on, and ensured that we had this working for each piece of our translator as it was completed.

Throughout the process we used the weekly team meetings to allow each individual team member or sub team to present their work to the wider group. This ensured that we all remained on the same page regarding overall progression and gave every team member an opportunity to critique each others work.

## 4.2 Programming Guide

We start by noting that this process was more difficult than might otherwise be the case in a more cohesive workplace environment as each team member preferred to use their own editors and development environments ranging from emacs to eclipse.

We used as a reference the ocaml style guide below, although we were less restrictive:
http://caml.inria.fr/resources/doc/guides/guidelines.en.html

The golden rule in all development as always was to ensure a reasonable level of commenting across all code, preferring commenting at the top of functions rather than interspersed throughout.

In addition, we tried to use meaningful variable and function names whilst not being overly verbose.

In terms of formatting we were less proscriptive provided the format was readable and followed sensible ocaml idioms, such as having “in” ending a line rather than beginning it.

## 4.3 Project Timeline

Proposal: 26th September
Language Reference Manual: October 29th
AST design: November 7th
Test framework design: November 7th
AST implementation: November 20th
Test framework implementation: November 20th
Static semantic checking design: December 1st
Static semantic checking implementation: December 7th
Java printer design: December 6th
Java printer implementation: December 16th
Test cases for all stages of the compilation process: December 16th
Final Project / Presentation: December 19th

## 4.4 Roles and Responsibilities

As discussed we took a collaborative approach where every team member participated to some degree in all aspects of the process.

At different times during our execution of the project different team members took on different roles. Every team member had an opportunity to take a lead on at least one development aspect and we all collaborated in terms of design.

Whilst having a more top-down approach is often preferable, within our specific group dynamic we felt that a more collaborative approach would enable us to leverage the different cultural features of our group, and our mixed backgrounds and abilities.

We all took part in debugging and fixing across the system, and some specific responsibilities/leadership roles were:

Adam Dossa: Developed utility functions to generate strings for scanner / ast / sast and hooks into taml.ml to call these utilities. Developed semantic checking of AST and generation of SAST.

Le Chang: Developed generation of AST, parser, and interpreter(printing to JAVA).

Maria Taku: Developed testing framework and test cases. Helped implement additional semantic checking. Implemented Java libraries used by our compiled code.

Qiuzi Shangguan: Developed generation of AST,parser and interpreter(printing to JAVA).

## 4.5 Software Development Environment

Different team members used different development environments. The linux environment was used to compile and test our translator (using cygwin on windows, terminal on mac), whilst some members used emacs + tuareg, some members used vi and some members used eclipse. Git and GitHub which were used for version control and as an online repository respectively were used with a mixture of the command line approach and various lightweight front-ends (e.g. GitHub).

## 4.6 Project Log

a5f4133 - Maria A. Taku, 3 hours ago : Fixing formatting
b13be51 - Maria A. Taku, 5 hours ago : Updating taml.tar.gz
5073c90 - Maria A. Taku, 5 hours ago : updating test suite
3cf7aaa - Maria A. Taku, 5 hours ago : Merge branch 'master' of https://github.com/joyayako/TaML
98764ff - Maria A. Taku, 5 hours ago : Update test cases
81e1480 - Maria A Taku, 5 hours ago : updating test cases
0dedb1b - Maria A Taku, 6 hours ago : updating slides
58bb502 - Le Chang, 23 hours ago : commented jpp.ml
79347a4 - Adam Dossa, 27 hours ago : Slightly fixed version of demo
4ea4884 - Maria A. Taku, 28 hours ago : removing unused files
996d1d5 - Adam Dossa, 29 hours ago : unify
23563d4 - Adam Dossa, 29 hours ago : Merge slides
7908bf8 - Adam Dossa, 29 hours ago : Add code coverage html
49c8f97 - Le Chang, 29 hours ago : le.ppt
eb42e0c - Maria A. Taku, 29 hours ago : Merge branch 'master' of https://github.com/joyayako/TaML
6bf56e2 - Maria A. Taku, 29 hours ago : cleaning up dir
680f08c - Adam Dossa, 29 hours ago : ppt 4
a892876 - Maria A. Taku, 29 hours ago : Merge branch 'master' of https://github.com/joyayako/TaML
038961f - Maria A. Taku, 29 hours ago : updating gitignore
d20e317 - Adam Dossa, 29 hours ago : fix some cases
e7189f2 - Adam Dossa, 29 hours ago : Add run.sh
2fc11c5 - Maria A. Taku, 30 hours ago : Cleaning up unused files
a9099d4 - Maria Ayako Taku, 30 hours ago : Merge branch 'master' of https://github.com/joyayako/TaML
fa978b4 - Maria Ayako Taku, 30 hours ago : Update test cases
f32ce59 - Adam Dossa, 30 hours ago : Fixes for SAST & JPP
864ef46 - Maria Ayako Taku, 30 hours ago : Merge branch 'master' of https://github.com/joyayako/TaML
60a068a - Maria Ayako Taku, 30 hours ago : Fixing bug in test case
5a98aa1 - Quizi Shangguan, 30 hours ago : test_assignment_2
fbf2662 - Quizi Shangguan, 30 hours ago : char->character
e29d9ca - Maria Ayako Taku, 31 hours ago : Merge branch 'master' of https://github.com/joyayako/TaML
c6c5d68 - Maria Ayako Taku, 31 hours ago : Fixing bug in printers

39c28ef - Le Chang, 31 hours ago : jpp.ml
d9567af - Maria Ayako Taku, 31 hours ago : Updating Printers
bb24482 - Le Chang, 33 hours ago : updated test cases
b9d7010 - Adam Dossa, 34 hours ago : Game of Life Demo + translate fix
2ba52cd - Le Chang, 2 days ago : seval update
374fe55 - Le Chang, 2 days ago : jpp.ml
44eab15 - Quizi Shangguan, 2 days ago : Merge branch 'master' of https://github.com/joyayako/TaML
29bcb5b - Quizi Shangguan, 2 days ago : add .ppt I can't use pptx, sorry
ef6e8a0 - Maria A Taku, 2 days ago : Removing unused Noexpr from language
18b57ad - Maria A Taku, 2 days ago : Correct missing IF_NO_ELSE error
b436c78 - Maria A Taku, 2 days ago : minor updates
a17f2ff - Maria A Taku, 2 days ago : Update Java Lib
d96dce7 - Maria A Taku, 2 days ago : adding slide deck
b733ee0 - Maria A Taku, 2 days ago : adding new case to Tester.Java
f5610f7 - Quizi Shangguan, 2 days ago : add file Java_out_result to store .out and .Java files for debugging
ab962ba - Quizi Shangguan, 2 days ago : update testall.sh
2ff8fe0 - Quizi Shangguan, 2 days ago : update cleantest and testall
5f1f506 - Le Chang, 2 days ago : Merge branch 'master' of https://github.com/joyayako/TaML
e736f2e - Le Chang, 2 days ago : jpp.ml
27ce307 - Adam Dossa, 2 days ago : Add tests for SAST
06159ac - Adam Dossa, 2 days ago : Fix some tests and translate
0ffab6c - Le Chang, 2 days ago : new printer
04e1ad2 - Maria A Taku, 2 days ago : Updating Java Lib with new line assignments
cd92334 - Maria A Taku, 2 days ago : Housekeeping: Updating READMEs, gitignore, etc.
4a162e0 - Adam Dossa, 2 days ago : Reverse statements in sast block
99b8a20 - Maria A Taku, 3 days ago : Merge branch 'master' of https://github.com/joyayako/TaML
f860d79 - Maria A Taku, 3 days ago : Updating test cases to increase code coverage
61abe7e - Le Chang, 3 days ago : jpp.ml
4f247ca - Le Chang, 3 days ago : jpp.ml
100c1e0 - Quizi Shangguan, 3 days ago : NewTotalCell
350c4ef - Quizi Shangguan, 3 days ago : add NewTotalCell as part in AST
5070d64 - Adam Dossa, 3 days ago : Build Sast Tree
fad4b8d - Maria A Taku, 3 days ago : updated testing suite to check .Java and .out files)
f588644 - Maria A Taku, 3 days ago : Updating LRM with final language design
b6002f8 - Maria A Taku, 3 days ago : Updating Java lib: setVal() for all cases, etc.
0bb4d62 - Maria Ayako Taku, 3 days ago : Merge branch 'master' of https://github.com/joyayako/TaML
f2f31aa - Quizi Shangguan, 4 days ago : update taml
e9092ae - Le Chang, 4 days ago : jpp.ml
41bb5b7 - Quizi Shangguan, 4 days ago : update testall.sh
60e5192 - Quizi Shangguan, 4 days ago : delte testjpp and update testall.sh
b9fd9a3 - Adam Dossa, 4 days ago : Allow named classes
4d183bb - Quizi Shangguan, 4 days ago : update testjpp.sh
0c587fc - Quizi Shangguan, 4 days ago : update jpp
89fec70 - Quizi Shangguan, 4 days ago : update parser jpp
79290f6 - Maria Ayako Taku, 4 days ago : Updating Java Libs
6a95a75 - Adam Dossa, 4 days ago : First draft of project plan
7071bd7 - Le Chang, 4 days ago : jpp.ml
116cb5b - Le Chang, 4 days ago : jpp.ml
0812d02 - Le Chang, 4 days ago : Merge branch 'master' of https://github.com/joyayako/TaML

c34698d - Le Chang, 4 days ago : jpp.ml
50e9047 - Maria Ayako Taku, 4 days ago : Merge branch 'master' of https://github.com/joyayako/TaML
91ac721 - Maria Ayako Taku, 4 days ago : Updating test cases with new syntax
1dd32ee - Le Chang, 4 days ago : update
36096bd - Quizi Shangguan, 4 days ago : taml.ml
a4cc2db - Maria Ayako Taku, 4 days ago : Adding Printers to Java_lib
bc979c4 - Le Chang, 4 days ago : Merge branch 'master' of https://github.com/joyayako/TaML
312af8a - Le Chang, 4 days ago : jpp.ml
5fe4a6b - Quizi Shangguan, 4 days ago : test jpp script
4023951 - Quizi Shangguan, 4 days ago : test jpp scrit
1e853fb - Maria Ayako Taku, 4 days ago : Merge branch 'master' of https://github.com/joyayako/TaML
e3c0616 - Adam Dossa, 4 days ago : Put compile into taml as an option
35fb8e1 - Maria Ayako Taku, 4 days ago : updating LRM
30fac7d - Quizi Shangguan, 4 days ago : makefile
38fddee - Quizi Shangguan, 4 days ago : update one testcase
e5f1f95 - Le Chang, 5 days ago : jpp.ml
20d39bf - Le Chang, 5 days ago : jpp update
1f3e282 - Quizi Shangguan, 5 days ago : change lvalue in parser
d30d7a0 - Quizi Shangguan, 5 days ago : add TableAsCellValue LineAsCellValue
cd3117b - Maria A Taku, 5 days ago : fixing repo
c0bdaf6 - Maria A Taku, 5 days ago : Fixing repo
d7d54b3 - Maria A Taku, 5 days ago : Updating .gitignore
4b1f3e0 - Le Chang, 5 days ago : jpp.ml
51c8825 - Le Chang, 5 days ago : new lib
5c94f96 - Le Chang, 5 days ago : jpp.ml
dd098fe - Le Chang, 5 days ago : jpp.ml
4c24888 - Le Chang, 6 days ago : jpp update
0a6865a - Le Chang, 6 days ago : jpp update
e94b70b - Le Chang, 6 days ago : jpp update
589badf - Quizi Shangguan, 6 days ago : add different tableasline_r and tableasline_l
49707b7 - Le Chang, 6 days ago : jpp
3686642 - Le Chang, 6 days ago : cellascell
9cbd45e - Le Chang, 6 days ago : cellascell
6795bf3 - Maria Ayako Taku, 6 days ago : Fixing some bugs in semantic_check.ml
a758fd2 - Le Chang, 6 days ago : Java code printer version 1
6726427 - Maria A Taku, 7 days ago : Cosmetic changes to naming conventions
a628023 - Adam Dossa, 11 days ago : Fix FOR loops and return statements
7d44434 - Adam Dossa, 11 days ago : Semantically Checked Trees
818a177 - Maria Ayako Taku, 11 days ago : Updating Java library
cfef908 - Maria Ayako Taku, 11 days ago : Updating LRM
b9e560e - Maria Ayako Taku, 2 weeks ago : Updated Tables to autoresize columns to fit contents
53f3e53 - Maria Ayako Taku, 2 weeks ago : Merge branch 'master' of https://github.com/joyayako/TaML
c9a935f - Maria Ayako Taku, 2 weeks ago : Update LRM with col/row limits
f0d6c96 - Quizi Shangguan, 2 weeks ago : Change testcase output
17bfea1 - Quizi Shangguan, 2 weeks ago : Add void into parse, parse tree for fdecl have two cases
e13b363 - Maria Ayako Taku, 2 weeks ago : Added Void to Sanner Parser. Upated test inputs
afd9f22 - Maria Ayako Taku, 2 weeks ago : Updated Java Lib with function to copy a table from a table
d296ad2 - Maria Ayako Taku, 2 weeks ago : Updates to LRM as discussed in today's meeting
b2b7a37 - Maria A Taku, 2 weeks ago : Updating Java Lib

e7818bd - Maria A Taku, 2 weeks ago : Updating Java Lib
350badb - Adam Dossa, 2 weeks ago : Update test cases
6cd5f27 - Maria A Taku, 2 weeks ago : first round of JAST etc. code
32de8c2 - Le Chang, 2 weeks ago : bugs
3dbd7a0 - Maria A Taku, 2 weeks ago : Fixed small bugs in parser.mly
247566c - Maria A Taku, 2 weeks ago : commenting
f7894c0 - Maria A Taku, 2 weeks ago : Adding blank jast.ml etc. files. Updating Makefile
a5fabc2 - Adam Dossa, 3 weeks ago : Clean ast.mli
503b512 - Adam Dossa, 3 weeks ago : Code to allow testing of ASTs
cf22e40 - Maria A Taku, 3 weeks ago : updating tests to check MOD and BREAK
d262642 - Maria A Taku, 3 weeks ago : Updating Scanner/Parser with BREAK and MOD
466cdc7 - Maria A Taku, 3 weeks ago : Updating LRM with mod and break
ef79776 - Maria A Taku, 3 weeks ago : Adding comments so it's pretty for the TA's to grade
6aecfd6 - Maria A Taku, 3 weeks ago : Adding comments so it's pretty for the TA's to grade
8875e7a - Maria A Taku, 3 weeks ago : new dir
04c7d60 - Maria Ayako Taku, 3 weeks ago : Reorganizing folders to make room for Compiler/Interpreter
2762269 - Maria Ayako Taku, 3 weeks ago : Updated README for test files
e09c8c4 - Maria Ayako Taku, 3 weeks ago : Updated test to show number of failed tests
55f136f - Maria Ayako Taku, 3 weeks ago : Updated test cases
9e0050e - Maria A Taku, 3 weeks ago : Adding final test
290eeef - Maria A Taku, 3 weeks ago : Updating the test suite
47ebcef - Maria A Taku, 3 weeks ago : Updated String/char in scanner to make them more robust
90e72bb - Maria A Taku, 3 weeks ago : Updating gitignore
5b30f05 - Maria A Taku, 3 weeks ago : LRM was inconsistent about table initialization.  Edited so table initialization is always consistent with parser.mly
7cb00fb - Quizi Shangguan, 3 weeks ago : change parser.mly
f58bc8a - Quizi Shangguan, 3 weeks ago : Handle Unary Minus
9a28ea6 - Quizi Shangguan, 3 weeks ago : Merge branch 'master' of https://github.com/joyayako/TaML
392d49f - Maria A Taku, 3 weeks ago : Removed negative sign ability from scanner (should be handled by parser) Added first batch of test files
effbc2f - Maria A Taku, 4 weeks ago : added test_results.out to .gitignore
a22100c - Maria A Taku, 4 weeks ago : small edit to remove annoying, unimportant error message
66397cf - Maria A Taku, 4 weeks ago : Completed testall.sh. Commented line 16 in taml.ml so code works for now (temp fix).  Added PRINT to scanner_utils.ml
7f0dabb - Maria A Taku, 4 weeks ago : Merge branch 'master' of https://github.com/joyayako/TaML
4e39cad - Maria A Taku, 4 weeks ago : daily commit
57ef5db - Quizi Shangguan, 4 weeks ago : parser.mly
15f248c - Quizi Shangguan, 4 weeks ago : modify minor error in parser.mly
4bf2f0e - Quizi Shangguan, 4 weeks ago : Merge branch 'master' of https://github.com/joyayako/TaML
8b47b49 - Quizi Shangguan, 4 weeks ago : Make reverse of program in parser.mly
d511b15 - Le Chang, 4 weeks ago : debugs
26381cb - Adam Dossa, 4 weeks ago : Make a better good test for taml
bfd3eef - Adam Dossa, 4 weeks ago : ignore cmo/cmi files
ed546eb - Adam Dossa, 4 weeks ago : Add ast to Makefile & taml.ml
080df6f - Maria A. Taku, 4 weeks ago : No changes -- just merging
1063a9e - Quizi Shangguan, 4 weeks ago : Merge branch 'master' of https://github.com/joyayako/TaML
21cadb2 - Quizi Shangguan, 4 weeks ago : parser.mly
0808d41 - Quizi Shangguan, 4 weeks ago : parser.mly
b08345c - Quizi Shangguan, 4 weeks ago : change ast

44620b9 - Adam Dossa, 4 weeks ago : add ast etc to makefile
13fbfb6 - Maria A Taku, 4 weeks ago : First half of testall.sh completed
f145ebb - Maria A Taku, 4 weeks ago : Merge branch 'master' of https://github.com/joyayako/TaML
cdc478c - Maria A Taku, 4 weeks ago : First half of testall.sh completed
58c14ef - Quizi Shangguan, 4 weeks ago : change parser
25dd193 - Quizi Shangguan, 4 weeks ago : change ast
e09f6af - Quizi Shangguan, 4 weeks ago : change parser
670ad5a - Quizi Shangguan, 4 weeks ago : change parser
8fc84db - Quizi Shangguan, 4 weeks ago : change ast
914657a - Quizi Shangguan, 4 weeks ago : change type
fdf3553 - Le Chang, 4 weeks ago : parser
ffdf30c - Le Chang, 4 weeks ago : parser
a34f11e - Maria A Taku, 4 weeks ago : Completed scanner_utils.ml
bad1c78 - Maria A Taku, 4 weeks ago : Updating LRM/Scanner to include WHILE loops
3914059 - Le Chang, 4 weeks ago : Merge branch 'master' of https://github.com/joyayako/TaML
6850d48 - Le Chang, 4 weeks ago : ast
1e9a632 - Maria A Taku, 4 weeks ago : Adding .gitignore file
6f2f0aa - Maria A Taku, 4 weeks ago : adding .gitignore
6628ef8 - Le Chang, 4 weeks ago : file for scope checking and type checking
a51be76 - Le Chang, 4 weeks ago : ast
f5cdedc - Le Chang, 4 weeks ago : adding ast
500ee80 - Le Chang, 4 weeks ago : adding ast
9e2a39b - Le Chang, 4 weeks ago : adding ast
bf2929c - Maria A Taku, 4 weeks ago : Updated LRM to include memory allocation for table/line/cell and to allow cell-from-line assignments such as 'cell c = line[2]'
752ff1d - Maria A Taku, 4 weeks ago : Updating LRM
ce60da5 - Maria A Taku, 5 weeks ago : Updated the LRM based on the TA's feedback from 11/9/12
511b36b - Adam Dossa, 5 weeks ago : Initial checkin for testing infrastructure
3fac81c - Maria A Taku, 7 weeks ago : Updated the LRM
637e964 - Maria A Taku, 7 weeks ago : Edited Parser.mly to allow variables to be assigned NULL
03e63d4 - Adam Dossa, 7 weeks ago : Added various new grammer rules
a7a25f0 - Adam Dossa, 7 weeks ago : Added various needed grammer
84d8cbb - Maria A Taku, 7 weeks ago : Added ability to declare int via a cell selection. Removed the final 'rule not reduced' warning
3aab0ed - Le Chang, 7 weeks ago : Merge branch 'master' of https://github.com/joyayako/TaML
c33c1fd - Le Chang, 7 weeks ago : Adding second version of Parser -- corrected reduce/reduce conflict
9af7374 - Maria A Taku, 7 weeks ago : Moving Documentation to its own folder to clean up the directory
7ca0723 - Maria A Taku, 7 weeks ago : Merge branch 'master' of https://github.com/joyayako/TaML
6c17cbe - Maria A Taku, 7 weeks ago : Moving Documentation to its own folder to clean up the directory
a73640e - Maria A Taku, 7 weeks ago : Moving Documentation to its own folder to clean up the directory
931ffb9 - Adam Dossa, 8 weeks ago : Additional check in of scanner / parser first draft
f90633a - Adam Dossa, 8 weeks ago : First draft of scanner / parser for TaML
b1b52a9 - Adam Dossa, 8 weeks ago : test for Adam
bace98d - Adam Dossa, 8 weeks ago : Add 123 to test.txt
41eab25 - Adam Dossa, 8 weeks ago : Revert "Revert "This is a test""
5231ae5 - Maria A Taku, 8 weeks ago : Removing test file
92a31cf - Maria A Taku, 8 weeks ago : This is a test
efcd22e - Maria A Taku, 8 weeks ago : Adding finished versions of papers we submitted for this class: Proposal, etc.
13eba44 - Maria Taku, 8 weeks ago : Initial commit

# 5 Architectural Design

## 5.1 Major components

The TaML translator takes in TaML source code files with the extension name .taml as input. It is designed to be a compiler which converts TaML source code into Java source code. And the Java code is compiled and run using Java environment. The architecture of TaML consists of five major components: scanner, parser, translator, Java printer, and TaML Java library.

### 5.1.1 Scanner

The scanner is the first stage of the compiler which takes in TaML source code and then outputs tokens according to the lexical rules our language defines. The scanner is implemented in scanner.mll.

### 5.1.2 Parser

The tarser receives the output of sequence tokens from scanner, and constructs an abstract syntax tree (AST) according to our syntax production rules. Parser.mly contains the related code.

### 5.1.3 Translator

The translator runs static semantic checking on the AST and output a semantically-checked AST (SAST).

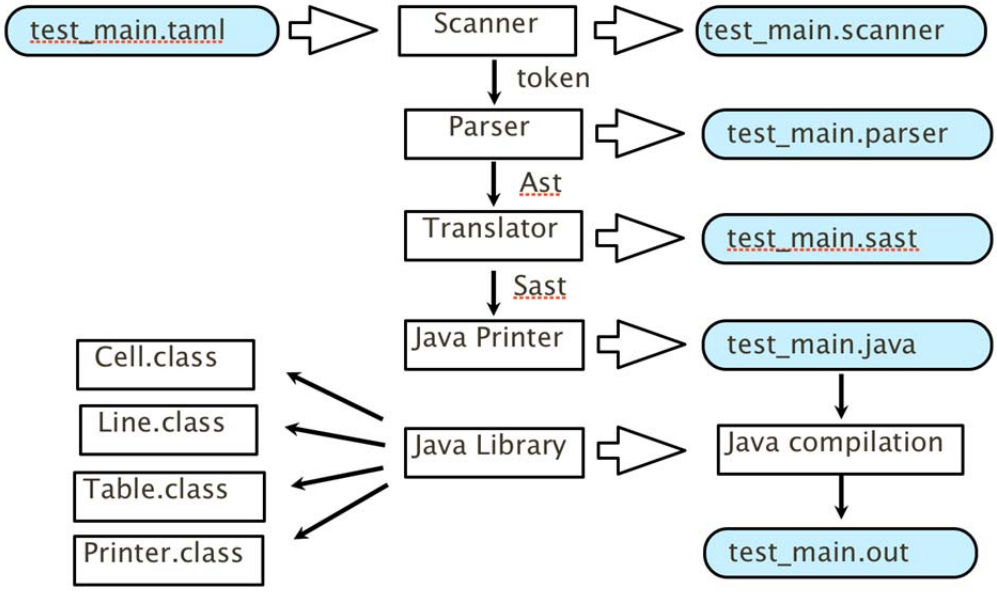



*Figure 5.1 Overview of TaML compiler architecture*

It is worth noting that our language was made more complex by our choice to make it somewhat dynamically typed. Whilst every variable has a specific type determined at compile time, since a Cell can have either integers or floats as its underlying type, and this underlying type could be dynamically determined, this aspect of our typing was dynamic.

This had the immediate consequence that whilst we did perform static type checking, it is not possible to check the consistency of all types at compile time, and some checking must be deferred to runtime. The approach taken was a best efforts based type checking, where if we have explicit type information we perform static type checking, and otherwise we perform a weaker form of type checking where we make no assumptions over whether a cell contained either integers or floats.

We performed the following static semantic checks:

1. Validity of break statements.
2. Validity of return statements.
3. Type checks across assignments, binary operators and initialization, using strong type checking where we have explicit type information, otherwise weak.
4. All variables are declared before use.
5. Indexes into our Table / Line types were correctly types, and that we only indexed into Table / Line types.
6. Boolean operators used only with Booleans.
7. If / While /For predicates were Boolean, and For loop counters were integers.
8. All functions are defined.
9. Function calls match the type signature of the function being called in both its arguments and return types.
10. Every function not returning type Void had an always reachable return statement.
11. Every program contains a main function.

### 5.1.4 Java Printer

After semantic checking, the Java printer converts the SAST into Java source code by walking through the tree. There are several changes from .taml to .Java. In TaML, the main function is the entry point and the Java printer transforms this function into “public static void main(String[] args).” For table manipulation expressions and statements, the printer will output Java code which calls corresponding functions in the Java library. The Java printer also creates a new class that has the same name as the .taml file (e.g., if the .taml file is called “demo.taml” the Java class will be named “demo”), which contains all printed Java code, including the main function. The Java printer is implemented in jpp.ml.

### 5.1.5 Java Library

The Java library provides Table, Line, and Cell classes. All convenient operations on the tables, lines, and cells are implemented by functions in each of these classes. Cells are the fundamental objects for Table: every table

is a two-dimension array of cells, and every line is a one-dimensional array of cells. The following figure gives an overview of the Java library.

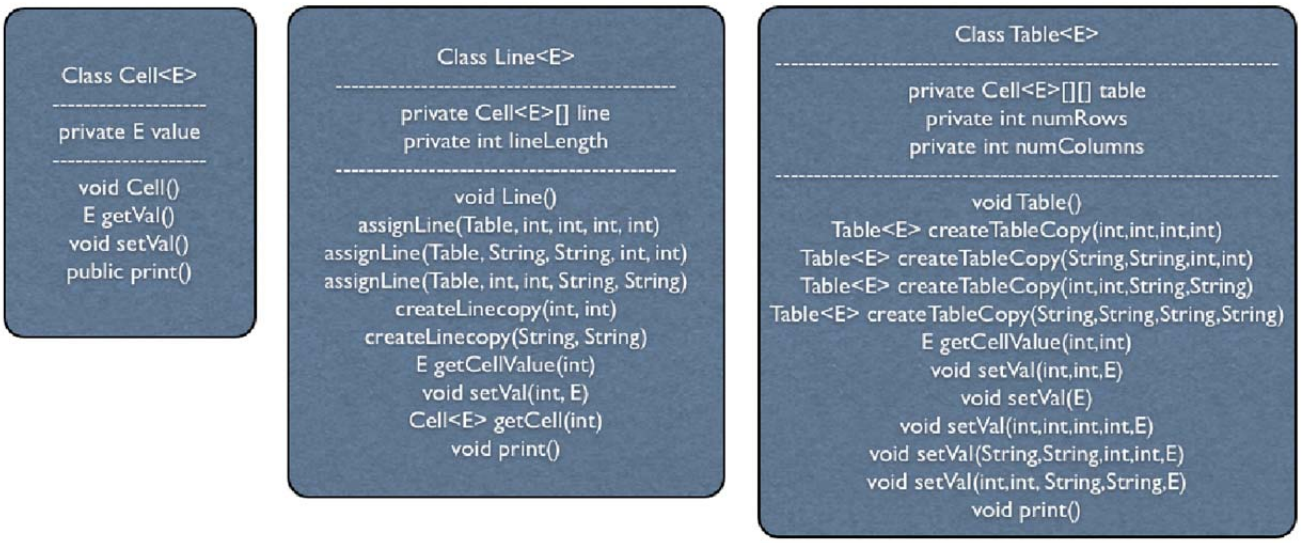



*Figure 5.2 Classes in Java Library*

## 5.2 Interfaces between components

The interface files use in TaML are ast.mli and sast.mli.

The ast.mli interface is provided by the Parser; the Parser passes an AST to the Translator.

The sast.mli interface is provided by the Translator; the Translator outputs a semantically-checked SAST to the Java printer.

## 5.3 Responsibility

Every group member is familiar with every component of TaML and worked at least in part on every component. However, some members had more responsibility and/or leadership roles for one or more components. The following table shows who had the main responsibility for each part of the TaML compiler.

| | Adam Dossa | Le Chang | Maria Taku | Qiuzi Shangguan |
|---|---|---|---|---|
| Scanner | ♦ | ♦ | ♦ | ♦ |
| Parser | ♦ | ♦ | ♦ | ♦ |
| Ast.mli | | ♦ | | ♦ |

| | | | | |
|---|---|---|---|---|
| Translator | ♦ | | ♦ | |
| Sast.mli | ♦ | | | |
| Java Printer | | ♦ | | ♦ |
| Java Library | | | ♦ | |

# 6 Test Plan

In order verify and maintain the integrity of the TaML, a full regression suite was built. The basic test cases were chosen and coded near the beginning of our project. Output "golden" files of expected results were then created for all steps of the language development. For example, there were output cases that verified scanner results, parser results, semantically checked ast results, Java results, and output results. As each step in the development of the language was passed, and as new features were added, the regression test suite was always run to verify that the language operated correctly and that new code was not breaking old features.

## 6.1 The TaML Test Suite

The file extensions of the TaML Test Suite included:

| | |
|---|---|
| .taml | → TaML input files. These contain the actual, TaML code |
| .scanner | → Testing output after the Scanner phase. Prints out tokens in a string-friendly format |
| .parser | → Testing output after the Parser phase. Outputs the AST structure in a string-friendly format. The structure of the tree is indicated by recursively putting children within brackets ( [ ] ). |
| .sast | → Testing output after semantic checking. Outputs a semantically correct AST in a string-friendly format |
| .Java | → taml code which has been translated into the target language, Java |
| .out | → the final output resulting from running a .taml file. In other words, the output produced by running a .Java file |

The files themselves of the test suite were stored within two folders:

| | |
|---|---|
| test-input | → This folder stores all the test case input files (e.g., contains all .taml files) |
| test-outpt | → This folder contains all expected output files in response to the test-input folder's cases. As such, this folder contains .scanner, .parser, .sast, .Java, and .out files. |

## 6.2 Testing Automation

In order to run the tests, an automated script called "testall.sh" was created. Running this script automatically executes all test cases in the test-input folder, generates appropriate outputs for each test case, compares these outputs to the "golden" expected output files contained in the test-output folder, and then saves the results of the comparison in a file called test_results.out. The test_results.out file includes a detailed description of any difference between the generated and expected outputs. A brief summary of the errors/differences is also printed to the terminal for the user's reference.

## 6.3 Test Cases

Table 6.2 shows a listing of all test-cases and the basic features they were intended to verify. These test cases were chosen in order to cover all aspects of our language, such as the all keywords that could appear, all control situations that could appear (e.g., an IF statement without an ELSE, etc.), all function behaviors (e.g., recursion, return types, parameter passing, etc.), all variable behavior (e.g., types of variable, local vs. global scope, initialization situations, assignment situations, etc.), all uses of TaML specific types (e.g., table, line, and cell behaviors), and such.

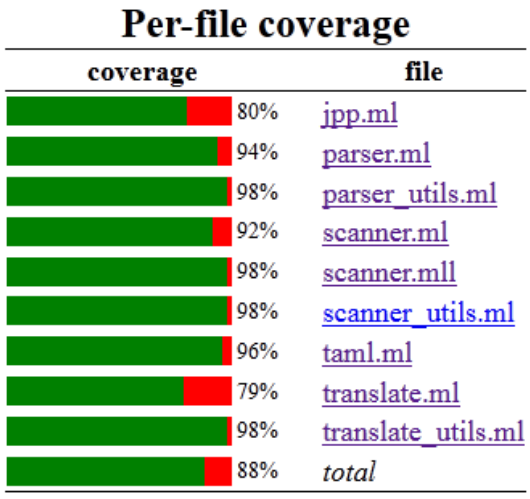
| **File** | **Testing Target** |
|---|---|
| test_arithmetic_1.taml | Tests basic arithmetic functions |
| test_arithmetic_2.taml | Test precedence and unary minus |
| test_assignment_1.taml | Test assignment of general types (int, float, string, char, bool) |
| test_assignment_2.taml | Test assignment of TaML types (table, line, cell) |
| test_assignment_3.taml | Test assignment of NULL to variables |
| test_boolean_logic_1.taml | Test boolean logic operators |
| test_break_1.taml | Test basic breaking from a FOR loop |
| test_break_2.taml | Test basic breaking from a WHILE loop |
| test_break_3.taml | Test breaking from more advanced statements |
| test_break_4.taml | Test breaking from more advanced statements |
| test_comments_1.taml | Test comments are parsed correctly |
| test_comparison_logic_1.taml | Test comparison logic operators |
| test_declaration_1.taml | Test default behavior or declared variables |
| test_for_1.taml | Test for loops |
| test_func_1.taml | Test recursion |
| test_func_2.taml | Test using expressions as parameters |
| test_func_3.taml | Test passing arguments to functions |
| test_func_4.taml | Test return values of functions |
| test_global_1.taml | Test global variable behavior |
| test_if_1.taml | Test successful execution of an IF-no-ELSE statement |

| | |
|---|---|
| test_if_2.taml | Test successful execution of IF in an IF/ELSE statement |
| test_if_3.taml | Tests non-execution of an IF-no-ELSE statement |
| test_if_4.taml | Test successful execution of ELSE in an IF/ELSE statement |
| test_if_5.taml | Test boolean expressions in controlling IF/ELSE statements |
| test_main.taml | General Purpose Test: Tests all major table manipulation functions which can be performed by TaML |
| test_punctuation_1.taml | Test punctuation: more for scanner/parser testing |
| test_return_1.taml | Test all return types |
| test_scope_1.taml | Test local versus global scope |
| test_while_1.taml | Test WHILE loops |

*Table 6.2 Testing Suite Test Cases*

## 6.4 Code Coverage

In order to gauge the coverage of the test suite, a bisect report was generated. The summary results of the bisect report are shown below:




Although the report does not show exactly 100% coverage, checking the individual file reports showed that the uncovered code is, for the most part, extraneous code such as unused helper functions, exception handling, unimplemented add-ons (e.g., verbose vs quiet warning modes), etc..

During creation of the test suite, previous versions of the bisect report were used to identify any important cases that had been missed, and appropriate test cases were created to cover these situations.

## 6.5 Responsibilities

The individual responsibilities for the Testing Suite were as follows:

Adam Dossa: Wrote the utility files for generating output files after each intermediate testing step (e.g., scanner_utils.ml, parser_util.ml, translate_utils.ml). Generated .sast output files. Routinely ran the testing suite upon updating code/adding new features.

Le Chang: Generated .Java and .out output files. Amended testall.sh to generate, compile, and run .Java files. Routinely ran the testing suite upon updating code/adding new features.

Maria Taku: Wrote testall.sh that was responsible for running test-files, generating outputs, comparing outputs to expected results, and collecting the results into a report (test_results.out). Wrote all test-input test cases. Generated .scanner and .parser output files. Routinely ran the testing suite upon updating code/adding new features.

Qiuzi Shangguan: Generated .Java and .out output files. Amended testall.sh to generate, compile, and run .Java files. Routinely ran the testing suite upon updating code/adding new features.

## 6.6 Sample Programs

As mentioned above, the target language for TaML is Java. A few representative TaML-source language programs along with their Java-target language are shown below.

### 6.6.1 Basic Sample Program (No TaML Types)

The first sample program illustrates how basic TaML is interpreted into Java. As you can see from this example, when TaML-specific features (e.g., cell, line, table manipulations) are not involved, the languages do appear fairly similar. The below code is for the testing case, test_global_1.taml.

TaML Code

```
int a;
int b;

func void print_a(){
        print(a);
}

func void print_b(){
```

```
        print(b);
}

func void inc_ab(){
        a = a + 1;
        b = b + 1;
}

func void main(){
        a = 42;
        b = 21;
        print_a();
        print_b();
        inc_ab();
        print_a();
        print_b();
}
```

Java Target Language Code

```
package Java_lib;

public class test_global_1 {
        public static Integer a=null;
        public static Integer b=null;

        public static void print_a()
        {
                Printers.print(a);
        }

        public static void print_b()
        {
                Printers.print(b);
        }

        public static void inc_ab()
        {
                a = a + 1;
                b = b + 1;
        }
```

```
        public static void main(String[] args)
        {
                a = 42;
                b = 21;
                print_a();
                print_b();
                inc_ab();
                print_a();
                print_b();
                }
}
```

### 6.6.2 More Complex Sample Program (TaML Types are used)

In order to support the translation of TaML into the target code, various Java libraries were created and stored in the Java_lib package. These libraries supplemented the creation and manipulation of TaML types (i.e., cells, lines and tables) when being translated into Java. As such, the below Java target code includes various Java_lib calls in order to properly deal with the TaML types.

This sample code provides a very basic example program that can create a budget, fill the budget with expenses, and then check whether or not you are within your budget. In a more real-world scenario, the *fillBudget()* function could, for example, be edited to provide a user's actual expense information.

TaML Code

```
string good = "your budget is good";
string bad = "spending too much money";
table t = ([10,10],float);
cell expenses = t[0,1];
cell maxBudget = t[1,1];

func void main(){
        setBudget(999.99);
        fillBudget();
        checkBudget();
}

func void setBudget(float maxBud){
        ^maxBudget = maxBud;
}

func void fillBudget(){
```

```
        ^expenses = 0.0;
        int i;
        for(i=0; i<10; i=i+1){
                ^t[i,0] = 100.0;
                ^expenses = ^expenses + ^t[i,0];
        }
}

func void checkBudget(){
        if(^expenses > ^maxBudget){
                print(bad);
        } else {
                print(good);
        }
}
```

Java Target Language Code

```
package Java_lib;

public class Ex
{
public static String good="your budget is good";
public static String bad="spending too much money";
public static Table<Float> t = new Table<Float>(10,10);
public static Cell<Float> expenses=t.getCell(0,1);
public static Cell<Float> maxBudget=t.getCell(1,1);

public static void main(String[] args)
{
        setBudget(999.99f);
        fillBudget();
        checkBudget();
}

public static void setBudget(Float maxBud)
{
        maxBudget.setVal(maxBud);
}

public static void fillBudget()
{
```

```
            expenses.setVal(0.f);
            Integer i=null;
            for (i = 0;i<10;i = i + 1)
            {
                  t.setVal(i,0,100.f);
                  expenses.setVal(expenses.getVal() + t.getCellValue(i,0));
            }
      }

      public static void checkBudget()
      {
            if ((Float)expenses.getVal()>(Float)maxBudget.getVal())
            {
                  Printers.print(bad);
            }
            else
            {
                  Printers.print(good);
            }
      }
}
```

# 7 Lessons Learned

Maria Taku

From the very beginning, take into account your target output language and how its syntax will be affected by your language design choices. Due to our TaML language design, near the end of our project we encountered several challenging (yet very interesting) obstacles to overcome. For example, our TaML types (cell, line, table) are all capable of representing integers or float. When translating into Java, this turned into a very interesting run-time type checking situation, where the Java code basically doesn't know until runtime whether the Cell, Line, and Table objects should hold integers or floats. In order to handle this issue, we had to develop several creative solutions, such as casting in Java to float/int as necessary, coding the Cell, Lines, and Tables as generics capable of holding float or integers, and lots of function overloading to ensure the proper translation of TaML code.

Another lesson I learned was to come up with a strong Language Reference Manual right from the start which clearly describes your language expectations. Throughout the development of TaML, or LRM was a valuable source in making certain everyone had the same "TaML" language in mind. It kept us all on track with what the expected behavior, functionality, and structure of TaML should be and--even though we often worked separately on various components of the language--allowed us to seamlessly integrate our components as necessary.

My advice to future teams would be to start early, and set a good foundation for your language--decide exactly what your language should do and what it should look like so everyone is on the same page. I would also suggest meeting up routinely to work together. By this I don't necessarily working on the same code or part of the project, but just to work together at, for example, the same table. This allows you to easily bring up bugs/concerns/integration questions as they appear, solve them, and move forward with the language. Your productivity is a lot higher than it would be if you were emailing questions and potentially waiting hours for responses before you can continue with your work.

Adam Dossa

The key lesson is that productivity as a team is wholly dependent on ones ability to parallelize across tasks within a project. To achieve this it is necessary to break up the project up into small modular tasks which are either highly decoupled or have well defined interfaces which can act as contracts between the developers working independently on these tasks.

Given that every team member was not familiar with Ocaml, there was a steep learning code for both the language and the idioms and paradigms underpinning the building of a translator. Often problems only became apparent once we had started coding a specific piece, so conforming to an iterative development process with regular group checkpoints enabled us to refine our ideas and plan over the course of the project.

Having dynamic typing complicated the semantic checking which was not fully appreciated at the time we wrote our LRM, but we decided to continue with this approach rather than retrospectively amending our LRM.

Having modular test cases as we went along meant it was immediately clear when a breaking change was made which was helpful, and meant that we were able to spot bugs earlier on in the development process without needing to wait for a finished project to do end to end testing.

To future teams I would offer the usual advice to start early. In addition I think it is sensible to give some thought early on to how your compiled programs will actually appear in their target language. This may help to guide some of the decisions in your LRM as well as give an indication of areas of difficulty that may be encountered later on.

Le Chang

The most important thing I learned from this project is that a clear design of a language, including its syntax, type system and usage is the key to a productive and meaningful compiler building process. Teamwork and version control make things easier,regular communication both with team members and TAs also facilitate our project a lot.

For those who had little idea before taking this class, about how programs are compiled and how functional programming works, I suggest you start early on reading some code fragments from past projects, actively participate in class and take advantage of TA hour and Pof. office hour. Start trying to code early.

When we were building the Ast and Parser, we were certainly confused about what would the Ast of our Table, Line ,Cell expression looks like. It took some time before we figure it out, even that, we still modified the Ast multiple times after the initial version. It was quite helpful that we started our testing process early so that a bug can be fixed pretty quickly before we move to next stage.

Translating to the target code revealed some weakness about our less explicit type system, type checking becomes harder, it is easy to define some fancy gammer and type and at first it also may seem really not hard to implement, but later you'll find the weakness of the design. The problem we met in translating taml code into Java code is type casting which we hadn't thought would happen earlier.

One final advice: Working with your team member at the same place for a afternoon every week would greatly enhance the productivity!

Qiuzi Shangguan

Firstly, thanks Adam, Maria and Le. I have learned a lot from you during our development of TaML. How to cooperate, communicate and coordinate in teamwork is the valuable experience I have got. I am appreciated of working together with you.

For the project, the most impressive lesson is design of language is important. After deciding to do table manipulation language, we design our language quickly. No much consideration of whether the grammar is ambiguous or unfeasible for later stages' implementation. So, we encountered many questions about implicit type of table, line and cell when type checking and printing to Java source code.

Another lesson is about LRM. The suggestion about LRM is that every detail should be defined clearly and every group member need to be familiar and have same understanding with LRM. LRM defines the language, so every time we're not sure about how our language should be in specific situation, LRM is the guideline. We still add new rules in LRM in later stages when we found the current grammars isn't feasible or no specific rules existing about certain problems.

It is necessary to follow the traditional procedures of developing Compiler. After type checking, we have tried to just output the original AST and print it to Java source code, skip the procedure of constructing SAST. However, we realized later that the SAST information is really necessary for following stages.

The use of Git and Github is really necessary for version control and group coding. And the early build-up of testing suite is really useful for debugging and finding the deficiency in language design.

# 8 Appendix (Code Listing)

## 8.1 Scanner.mll

```
(* Scanner.mll - Performs lexical analysis to generate tokens *)
(* Authors: Adam Dossa, Le Chang, Quizi Shangguan, Maria Taku *)

{ open Parser }

let digit = ['0' - '9']
let ip = digit+                          (* Float numbers - Pre-decimal point *)
let fp = digit+                          (* Float numbers - post-decimal point*)
let ie = digit+                          (* Integer numbers *)
let dp = '.'
let exp = 'e' ('+'|'-')? ip
(* Clearly ip, fp, ie are all identical, but for clarity we separate *)

rule token = parse
  [' ' '\t' '\r' '\n'] { token lexbuf } (* Whitespace *)
| "#"      { comment lexbuf }           (* Comments *)
| ';'      { SEMI }                     (* Punctuation *)
| '('      { LPAREN }
| ')'      { RPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| ','      { COMMA }
| '+'      { PLUS }                     (* Arithmetic *)
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '%'      { MOD }
| '='      { ASSIGN }                   (* Assignment *)
| "=="     { EQ }                       (* Comparison Logic *)
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }                      (* Boolean Logic *)
| "!"      { NOT }
```

```
| "||"     { OR }
| "return" { RETURN }                  (* Control Flow*)
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "while"  { WHILE }
| "break"  { BREAK }
| "int"    { INT }                    (* Data Types *)
| "float"  { FLOAT }
| "bool"   { BOOL }
| "char"   { CHAR }
| "string" { STRING }
| "NULL"   { NULL }                   (* Null *)
| "void"   { VOID }                    (* Void -- only used as return type *)
| '['      { LSQPAREN }               (*TaML exclusive*)
| ']'      { RSQPAREN }
| '@'      { WILDCARD }
| '~'      { RANGE }
| "cell"   { CELL }
| "line"   { LINE }
| "table"  { TABLE }
| "func"   { FUNC }
| "print"  { PRINT }
| "^"      { REF }                    (* Indicates an ID is for a Cell type. Example: ^myCell = 500; *)
(* Literals/Constants *)
| ie as lit { INTLITERAL(int_of_string lit) }
| (((ip dp fp)|(ip dp)|(dp fp))(exp)?)|(ip exp) as lit { FLOATLITERAL(float_of_string lit) }
| ['\"'] ([^ '\"']* as lit) ['\"'] { STRINGLITERAL(lit) }(* String can accept anything but " or /".  Can be empty *)
| ['\''] ( _ as lit) ['\''] { CHARLITERAL(lit) }          (* Char can accept anything but single quote. Cannot be
empty *)
| "true" | "false" as lit { BOOLLITERAL(bool_of_string lit) }

(* Variables *)
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as ident { ID(ident) }

(* End of File *)
| eof { EOF }

(* Anything else*)
| _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }

and comment = parse
  ['\r' '\n'] { token lexbuf }    (* Return to normal scanning *)
| _    { comment lexbuf }         (* Ignore other characters *)
```

## 8.2 Scanner_utils.ml

```
(* scanner_utils.ml - This utility is used for testing purposes.     *)
(* It outputs the results of the scanner (tokens) into string form   *)
(* Author: Adam Dossa and Maria Taku *)


open Scanner
open Parser

(* generate a string for each token. *)
let token_to_string token =
   (match token with
      | SEMI ->                        (* Punctuation *)
         "SEMICOLON\n"
      | LPAREN ->
         "LEFT_PAREN\n"
      | RPAREN ->
         "RIGHT_PAREN\n"
      | LBRACE ->
         "LEFT_BRACE\n"
      | RBRACE ->
         "RIGHT_BRACE\n"
      | COMMA ->
         "COMMA\n"
      | PLUS ->                        (* Arithmetic *)
         "PLUS\n"
      | MINUS ->
         "MINUS\n"
      | TIMES ->
         "TIMES\n"
      | DIVIDE  ->
         "DIVIDE\n"
      | MOD  ->
         "MOD\n"
      | ASSIGN ->                      (* Assignment *)
         "ASSIGN\n"
      | EQ ->                          (* Comparison Logic *)
         "EQUAL\n"
      | NEQ ->
         "NOT_EQUAL\n"
```

```
| REF ->
    "REFERENCE\n"
| LT  ->
   "LESS_THAN\n"
| LEQ ->
   "LESS_THAN_OR_EQUAL\n"
| GT  ->
   "GREATER_THAN\n"
| GEQ ->
   "GREATER_THAN_OR_EQUAL\n"
| AND ->                              (* Boolean Logic *)
   "AND\n"
| NOT ->
   "NOT\n"
| OR ->
   "OR\n"
| RETURN ->                           (* Control Flow*)
   "RETURN\n"
| IF ->
   "IF\n"
| ELSE ->
   "ELSE\n"
| FOR ->
   "FOR\n"
| WHILE ->
   "WHILE\n"
| BREAK ->
   "BREAK\n"
| INT ->                              (* Data Types *)
   "INT\n"
| FLOAT ->
   "FLOAT\n"
| BOOL ->
   "BOOL\n"
| CHAR ->
   "CHAR\n"
| STRING ->
   "STRING\n"
| NULL ->                         (* Null *)
   "NULL\n"
| VOID ->                         (* Void *)
  "VOID\n"
| PRINT ->                        (* TaML Exclusive *)
   "PRINT\n"
```

```
      | LSQPAREN  ->
          "LEFT_SQUARE_PAREN\n"
      | RSQPAREN  ->
          "RIGHT_SQUARE_PAREN\n"
      | WILDCARD ->
          "WILDCARD\n"
      | RANGE ->
          "RANGE\n"
      | CELL ->
          "CELL\n"
      | LINE ->
          "LINE\n"
      | TABLE ->
          "TABLE\n"
      | FUNC ->
          "FUNC\n"
      | INTLITERAL i ->                       (* Literals/Constants *)
          "INTLITERAL: " ^ string_of_int(i) ^ "\n"
      | FLOATLITERAL f ->
          "FLOATLITERAL: " ^ string_of_float(f) ^ "\n"
      | CHARLITERAL c ->
          "CHARLITERAL: " ^ (Char.escaped c) ^ "\n"
      | STRINGLITERAL s ->
          "STRINGLITERAL: " ^ s ^ "\n"
      | BOOLLITERAL b ->
          "BOOLLITERAL: " ^ string_of_bool(b) ^ "\n"
      | ID id ->                              (* Variables *)
          "ID: " ^ id ^ "\n"
      | EOF ->                                (* End of File *)
          "END_OF_FILE\n")

(* Generate a string for a set of tokens *)
let string_of_tokens tokens =
   let token_list = List.map token_to_string tokens in
   String.concat "" token_list
```

---

## 8.3 Type.mli

```
(* type.mli - Type definitions used by the Parser *)
(* Authors: Le Chang and Quizi Shangguan*)

type taty_ret =  (*TaML type definitions*)
```

```
    RInt
  | RFloat
  | RChar
  | RString
  | RBoolean
   (*TaML exclusive*)
  | RTable
  | RLine
  | RCell
  | RVoid

type taty_var =  (*TaML type definitions*)
    Int
  | Float
  | Char
  | String
  | Boolean
   (*TaML exclusive*
  | Table
  | Line
  | Cell

type literal =
    IntL of int
  | FloatL of float
  | CharL of char
  | BoolL of bool
  | StringL of string

type binop =
    Add
  | Mult
  | Div
  | Equal
  | Neq
  | Less_than
  | Leq
  | Greater_than
  | Geq
  | Or
  | And
  | Mod

type uniop =
```

```
    Not
```

## 8.4 Parser.mly

```
/* parser.mly */
(* Authors: Adam Dossa, Le Chang, Quizi Shangguan, Maria Taku *)

%{ open Ast %}
%{ open Type %}

%token <string> ID
%token RETURN IF ELSE FOR WHILE BREAK VOID
%token INT FLOAT BOOL CHAR STRING CELL LINE TABLE
%token <int> INTLITERAL
%token <float> FLOATLITERAL
%token <char> CHARLITERAL
%token <string> STRINGLITERAL
%token <bool> BOOLLITERAL
%token NULL VOID
%token SEMI LBRACE RBRACE COMMA
%token ASSIGN
%token RANGE
%token AND NOT OR REF
%token EQ NEQ LT LEQ GT GEQ
%token PLUS MINUS TIMES DIVIDE MOD
%token LSQPAREN RSQPAREN WILDCARD FUNC
%token PRINT
%token LPAREN RPAREN
%token EOF

%nonassoc NOELSE
%nonassoc ELSE

%right ASSIGN
%left OR
%left AND
%left EQ NEQ
%left LT GT LEQ GEQ
%left PLUS MINUS
%left TIMES DIVIDE MOD
%right NOT REF
%left LPAREN
```

```
%start program
%type <Ast.program> program

%%
program:
   program_r        { List.rev $1 }

program_r:
   /* nothing */    { [] }
 | program_r fdecl  { FuncDef($2)::$1 }
 | program_r vdecl  { GlobalVar($2)::$1 }

/* List of all return types */
ptype_ret:
    INT             { RInt }
  | FLOAT           { RFloat }
  | BOOL            { RBoolean }
  | CHAR            { RChar }
  | STRING          { RString }
  | TABLE           { RTable }
  | CELL            { RCell }
  | LINE            { RLine }
  | VOID            {RVoid}

/* List of all types */
ptype_var:
    INT             { Int }
  | FLOAT           { Float }
  | BOOL            { Boolean }
  | CHAR            { Char }
  | STRING          { String }
  | TABLE           { Table }
  | CELL            { Cell }
  | LINE            { Line }

/* table[range_op,range_op] range_op can be '@' or '1~2' */
range_op:
    WILDCARD                    { (All,All) }
  | expr RANGE expr             { ($1,$3) }

/* e.g. func int some_func (int a, int b, float c) { Table foo; do_something; } */
fdecl:
   FUNC ptype_ret ID LPAREN formals_opt RPAREN block_stmt
```

```
            {{ fname =    FuncSig($2,$3);
                          formals = $5;
                          body= $7;
            }}

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

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


/* we allow variable initialization to default (null or false) or specific values */
/* Default initiliazations: Keep in mind that our language will default set all of */
/* these to a value of "NULL" except for bool, which has a default value of "false" */
vdecl:
    ptype_var ID SEMI                   { NullInit($1,$2) }                 /* default initialization*/
  | ptype_var ID ASSIGN expr SEMI       { VarInit($1,$2,$4) }               /* Explicit initialization value*/
  /* table default initialization*/
  | ptype_var ID ASSIGN LPAREN LSQPAREN expr COMMA expr RSQPAREN COMMA ptype_var RPAREN SEMI
                                        { TableInit($1,$2,$6,$8,$11) }

block_stmt:
    LBRACE stmt_list RBRACE { Block (List.rev $2) }

/* List of Statements declared -- used in fdecl*/
stmt_list:
  /* nothing */           { [] }
  | stmt_list stmt        { $2::$1 }

stmt:
    expr SEMI                   { Expr($1) }
  | PRINT expr SEMI             { Print($2) }      /*print a table,a line a cell or any other literals*/
  | RETURN expr SEMI            { Return($2) }
  | BREAK SEMI                  { Break }
  | IF LPAREN expr RPAREN block_stmt %prec NOELSE                       { If($3,$5,NoElseMark) }
  | IF LPAREN expr RPAREN block_stmt ELSE block_stmt                    { If($3,$5,$7) }
  | FOR LPAREN expr SEMI expr SEMI expr RPAREN block_stmt               { For($3,$5,$7,$9) }
  | WHILE LPAREN expr RPAREN block_stmt                                 { While($3,$5) }
  | vdecl                       {VarDecl($1)}
```

```
literal:
    INTLITERAL          { IntL($1) }
  | FLOATLITERAL        { FloatL($1) }
  | CHARLITERAL         { CharL($1) }
  | STRINGLITERAL       { StringL($1) }
  | BOOLLITERAL         { BoolL($1) }

expr:
    literal                     { Literal($1) }
  | ID                          { Id($1) }
  | ID LSQPAREN range_op COMMA range_op RSQPAREN  { TableAsTable($1,fst $3,snd $3,fst $5,snd $5) } /* table[1~2,3~4]
*/
  | ID LSQPAREN expr COMMA range_op RSQPAREN    { TableAsLine_R($1,$3,fst $5,snd $5) } /* table[3,1~2] */
  | ID LSQPAREN range_op COMMA expr RSQPAREN    { TableAsLine_L($1,fst $3,snd $3,$5) } /* table[2~3,5] */
  | ID LSQPAREN expr COMMA expr RSQPAREN        { TableAsCell($1,$3,$5) }  /* table[3,5] */
  | REF ID LSQPAREN expr COMMA expr RSQPAREN    { TableAsCellValue($2,$4,$6) }  /* ^table[3,5]*/
  | ID LSQPAREN range_op RSQPAREN               { LineAsLine($1,fst $3,snd $3) } /* line[1~2] */
  | ID LSQPAREN expr RSQPAREN                   { LineAsCell($1,$3) } /* line[1] */
  | REF ID LSQPAREN expr RSQPAREN               { LineAsCellValue($2,$4) } /* ^line[1] */
  | REF ID                                      { CellAsCell($2) } /* ^cell */
  | MINUS expr %prec TIMES                      { UnaryMinus($2) }

  | expr PLUS   expr            { Binop($1, Add, $3) }
  | expr MINUS  expr            { Binop($1, Sub, $3) }
  | expr TIMES  expr            { Binop($1, Mult, $3) }
  | expr DIVIDE expr            { Binop($1, Div, $3) }
  | expr MOD expr               { Binop($1, Mod, $3) }
  | expr EQ     expr            { Binop($1, Equal, $3) }
  | expr NEQ    expr            { Binop($1, Neq, $3) }
  | expr LT     expr            { Binop($1, Less_than, $3) }
  | expr LEQ    expr            { Binop($1, Leq, $3) }
  | expr GT     expr            { Binop($1, Greater_than, $3) }
  | expr GEQ    expr            { Binop($1, Geq, $3) }

  | expr AND    expr            { Binop($1, And, $3) }
  | expr OR     expr            { Binop($1, Or, $3) }
  | NOT expr                    { Uniop (Not, $2)}
  | lvalue ASSIGN expr          { Assign($1,$3) }
  | ID LPAREN actuals_opt RPAREN { Call($1,$3) }            /* Function Call */
  | LPAREN expr RPAREN          { $2 }
  | NULL                        { Null }

actuals_opt:
  /* nothing */     { [] }
```

```
  | actuals_list    { List.rev $1 }

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

lvalue:
    ID                { Id($1) }
  | REF ID            { NewTotalCell($2) }

  | REF ID LSQPAREN range_op COMMA range_op RSQPAREN       { TableAsTable($2,fst $4,snd $4,fst $6,snd $6) } /*
  | REF ID LSQPAREN range_op COMMA expr RSQPAREN           { TableAsLine_L($2,fst $4,snd $4,$6) } /* table[2~3,5] */
  | REF ID LSQPAREN expr COMMA expr RSQPAREN               { TableAsCellValue($2,$4,$6) }  /* ^table[3,5]*/

  | REF ID LSQPAREN expr RSQPAREN                          { LineAsCellValue($2,$4) } /* ^line[1] */
  | REF ID LSQPAREN range_op RSQPAREN                      { LineAsLine($2,fst $4,snd $4) } /* line[1~2] */
```

---

## 8.5 Parser_utils.ml

```
(* parser_utils.ml - This utility is used for testing purposes.     *)
(* It outputs the results of an AST in string-readable form         *)
(* Author: Adam Dossa  *)


open Type
open Ast

let string_of_taty_var = function
    Int -> "TYPE_INT"
  | Float -> "TYPE_FLOAT"
  | Char -> "TYPE_CHAR"
  | String -> "TYPE_STRING"
  | Boolean -> "TYPE_BOOL"
  | Table -> "TYPE_TABLE"
  | Line -> "TYPE_LINE"
  | Cell -> "TYPE_CELL"

let string_of_taty_ret = function
    RInt -> "TYPE_INT"
  | RFloat -> "TYPE_FLOAT"
  | RChar -> "TYPE_CHAR"
  | RString -> "TYPE_STRING"
```

```
  | RBoolean -> "TYPE_BOOL"
  | RTable -> "TYPE_TABLE"
  | RLine -> "TYPE_LINE"
  | RCell -> "TYPE_CELL"
  | RVoid -> "TYPE_VOID"

let string_of_literal = function
    IntL(l) -> "LITERAL_INT:[" ^ (string_of_int l) ^ "]"
  | FloatL(l) -> "LITERAL_FLOAT:[" ^ (string_of_float l) ^ "]"
  | CharL(l) -> "LITERAL_CHAR:[" ^ (Char.escaped l) ^ "]"
  | BoolL(l) -> "LITERAL_BOOL:[" ^ (if l then "#TRUE#" else "#FALSE#") ^ "]"
  | StringL(l) -> "LITERAL_STRING:[" ^ l ^ "]"

let string_of_binop = function
   Add -> "ADD"
 | Sub -> "SUB"
 | Mult -> "MULT"
 | Div -> "DIV"
 | Equal -> "EQUAL"
 | Neq -> "NEQ"
 | Less_than -> "LT"
 | Leq -> "LTE"
 | Greater_than -> "GT"
 | Geq -> "GTE"
 | Or -> "OR"
 | And -> "AND"
 | Mod ->"MOD"

let string_of_uniop = function
   Not -> "NOT"

let rec string_of_expr = function
    Literal(l) -> "EXPR_LITERAL:[" ^ string_of_literal l ^ "]"
  | TableAsTable(s,e1,e2,e3,e4) -> "EXPR_TABLE_AS_TABLE:[" ^ s ^ "," ^ (string_of_expr e1) ^ "," ^ (string_of_expr
e2) ^ "," ^
            (string_of_expr e3) ^ "," ^ (string_of_expr e4) ^ "]"
  | TableAsLine_L(s,e1,e2,e3) -> "EXPR_TABLE_AS_LINE_L:[" ^ s ^ "," ^ (string_of_expr e1) ^ "," ^ (string_of_expr e2)
^ "," ^
            (string_of_expr e3) ^ "]"
  | TableAsLine_R(s,e1,e2,e3) -> "EXPR_TABLE_AS_LINE_R:[" ^ s ^ "," ^ (string_of_expr e1) ^ "," ^ (string_of_expr e2)
^ "," ^
            (string_of_expr e3) ^ "]"
  | TableAsCell(s,e1,e2) -> "EXPR_TABLE_AS_CELL:[" ^ s ^ "," ^ (string_of_expr e1) ^ "," ^ (string_of_expr e2) ^ "]"
```

```
  | TableAsCellValue(s,e1,e2) -> "EXPR_TABLE_AS_CELL_Value:[" ^ s ^ "," ^ (string_of_expr e1) ^ "," ^ (string_of_expr
e2) ^ "]"
  | LineAsLine(s,e1,e2) -> "EXPR_LINE_AS_LINE:[" ^ s ^ "," ^ (string_of_expr e1) ^ "," ^ (string_of_expr e2) ^ "]"
  | LineAsCell(s,e1) -> "EXPR_LINE_AS_CELL:[" ^ s ^ "," ^ (string_of_expr e1) ^ "]"
  | LineAsCellValue(s,e1) -> "EXPR_LINE_AS_CELL_Value:[" ^ s ^ "," ^ (string_of_expr e1) ^ "]"
  | CellAsCell(s) -> "EXPR_CELL_AS_CELL:[" ^s^"]"
  | NewTotalCell(s) -> "EXPR_New_Total_CELL:[" ^s^"]"
  | Id(s) -> "EXPR_ID:[" ^ s ^ "]"
  | Binop(e1, o, e2) -> "EXPR_BINOP:[" ^ string_of_expr e1 ^ "," ^ string_of_binop o ^ "," ^ string_of_expr e2 ^ "]"
  | Assign(e1, e2) -> "EXPR_ASSIGN:[" ^ string_of_expr e1 ^ "," ^ string_of_expr e2 ^ "]"
  | Call(f, el) ->
    "EXPR_CALL:[" ^ f ^ ",LIST:[" ^ String.concat ", " (List.map string_of_expr el) ^ "]]"
  | Uniop(o, expr) -> "EXPR_UNIOP:[" ^ string_of_uniop o ^ "," ^ string_of_expr expr ^ "]"
  | All -> "ALL"
  | Null -> "NULL"
  | UnaryMinus(e) -> "EXPR_UNARYMINUS:[" ^ string_of_expr e ^ "]"

let string_of_init = function
    NullInit(t, s) -> "INIT_NULL:[" ^ string_of_taty_var t ^ "," ^ s ^ "]"
  | VarInit(t,s,e) -> "INIT_VAR:[" ^ string_of_taty_var t ^ "," ^ s ^ "," ^ (string_of_expr e) ^ "]"
  | TableInit(t1,s,e1,e2,t2) -> "INIT_TABLE:[" ^ string_of_taty_var t1 ^ "," ^ s ^ "," ^ string_of_expr e1 ^ ", " ^
string_of_expr
            e2 ^ ", " ^ string_of_taty_var t2 ^ "]"

let rec string_of_stmt = function
    Block(stmts) ->
    "STMT_BLOCK:[LIST:[" ^ String.concat "," (List.map string_of_stmt stmts) ^ "]]"
  | Expr(expr) -> "STMT_EXPR:[" ^ string_of_expr expr ^ "]";
  | Return(expr) -> "STMT_RETURN:[" ^ string_of_expr expr ^ "]";
  | Print(expr) -> "STMT_PRINT:[" ^ string_of_expr expr ^ "]";

  | If(e, s, NoElseMark) -> "STMT_IF_NOELSE:[" ^ string_of_expr e ^ "," ^ string_of_stmt s ^ "]"
  | If(e, s1, s2) ->  "STMT_IF:[" ^ string_of_expr e ^ "," ^
    string_of_stmt s1 ^ "," ^ string_of_stmt s2 ^ "]"
  | For(e1, e2, e3, s) ->
    "STMT_FOR:[" ^ string_of_expr e1  ^ "," ^ string_of_expr e2 ^ "," ^
    string_of_expr e3  ^ "," ^ string_of_stmt s ^ "]"
  | While(e, s) -> "STMT_WHILE:[" ^ string_of_expr e ^ "," ^ string_of_stmt s ^ "]"
  | NoElseMark -> "STMT_NOELSEMARK"
  | VarDecl(s)->string_of_init s
  | Break -> "STMT_BREAK"

let string_of_funcSig = function
  FuncSig(t, s) -> "FUNCSIG:[" ^ string_of_taty_ret t ^ "," ^ s ^ "]"
```

```
let string_of_varSig = function
  VarSig(t, s) -> "VARSIG:[" ^ string_of_taty_var t ^ "," ^ s ^ "]"

let string_of_func_decl fdecl =
  "FUNCDECL:[" ^ string_of_funcSig fdecl.fname ^ ",LIST:[" ^
  String.concat "," (List.map string_of_varSig fdecl.formals) ^ "],LIST:[" ^
  string_of_stmt fdecl.body ^ "]]"

let string_of_construct = function
    GlobalVar(i) -> "CONSTRUCT_GLOBALVAR:[" ^ (string_of_init i) ^ "]"
  | FuncDef(f) -> "CONSTRUCT_FUNCDEF:[" ^ (string_of_func_decl f) ^ "]"

let string_of_program construct_list =
  "PROGRAM:[LIST:[" ^ (String.concat "," (List.map string_of_construct construct_list)) ^ "]]\n"
```

---

# 8.6 Ast.mli

```
(* ast.mli -- Abstract Syntax Tree for Parser *)
(* Authors: Le Chang and Quizi Shangguan *)


open Type

(* Expressions *)
type expr =

    Literal of literal                                  (* 3.14159 etc.  *)
  | TableAsTable of string * expr * expr * expr * expr     (* represent table expression that looks like
t[1~2,2~5],t[1~2,@] *)
  | TableAsLine_R of string * expr * expr * expr          (* represent table expression that looks like t[4,3~5] *)
  | TableAsLine_L of string * expr * expr * expr          (* represent table expression that looks like t[4,3~5] *)
  | TableAsCell of string * expr * expr                   (* represents table expression that looks like t[5,3] *)
  | TableAsCellValue of string * expr * expr              (* represents table expression that looks like ^t[5,3] *)
  | LineAsLine of string * expr * expr                    (* represent line expression that looks like l[1~2] *)
  | LineAsCell of string * expr                           (* represent line expression that looks like l[4] *)
  | LineAsCellValue of string * expr                      (* represent line expression that looks like ^l[4] *)
  | CellAsCell of string                       (* An ID for a cell.  Distinguised from other ID's in that it's prefixed
by ^ *)
  | Id of string                                          (* foo *)
  | Binop of expr * binop * expr                          (* a + b *)
  | Assign of expr * expr                                 (* a = t[1,2~4] *)
```

```
  | Call of string * expr list                              (* foo(1, 25) *)
  | Uniop of uniop * expr                                   (* not a *)
  | All                                                     (* string here can only be "@" - wildcard*)
  | Null                                                    (* for null assignment*)
  | UnaryMinus of expr                                      (* -1 *)
  | NewTotalCell of string  (* An ID for a cell,table and line.  Distinguised from other ID's in that it's prefixed
by ^ *)

(* Return type and Name of a function *)
type funcSig = FuncSig of taty_ret * string

type varSig = VarSig of taty_var * string

(*initialization of variables *)
(* taty*string --> data type * var name *)
type  init =
    NullInit of taty_var * string                                (*assign null/default value*)
  | VarInit of taty_var * string * expr                          (*assign a specific value to the new variable*)
  | TableInit of taty_var * string * expr * expr * taty_var      (* e.g., table t = ([3,4], int) -- taty is only int
or float*)

(* Statements  *)
type stmt =
    Block of stmt list
  | Expr of expr                      (*foo = bar + 3; *)
  | Return of expr                    (* return 42 also includes return function_name *)
  | Print of expr
  | If of expr * stmt * stmt          (* if (foo == 42) {} else {} *)
  | For of expr * expr * expr * stmt  (* for loop *)
  | While of expr * stmt              (* while (i<10) { i = i + 1 } *)
  | NoElseMark                        (* For loops without else*)
  | VarDecl of init
  | Break

(* Function Declarations *)
type func_decl = {
             fname : funcSig;         (* Name and return type of the function *)
             formals : varSig list;   (* Formal arguments, type & names *)
             body : stmt;
}

(* Bascially, the parts of a program *)
type construct =
    GlobalVar of init
```

```
  | FuncDef of func_decl

type program = construct list (* global vars, funcs *)
```

---

## 8.7 Translate.ml

```
(* translate.ml - performs semantic checking on AST                      *)
(* Note on var_scope vs. und_scope: Due to the fact that in TaML         *)
(* tables/lines/cells can hold int OR float, we made a differentiation   *)
(* between general "variable type" and "underlying type." For            *)
(* table/line/cell the variable type is table/line/cell respectively     *)
(* while the underlying type will be int or float -- depending on what   *)
(* the object holds.  For all other types -- int/float/string/char/bool  *)
(* variable type = underlying type                                       *)
(* Author: Adam Dossa, modified by Maria Taku*)

open Type
open Ast
open Sast

type mode = Quiet | Verbose (* compiler directives *)

let print_warning mode w = match mode with
  | Verbose -> print_string("/* WARNING: " ^ w ^ "*/\n")
  | Quiet   -> print_string ""

(* Exception types *)                          (* TODO - use these instead of generic error everywhere *)
exception ReturnException of string           (* Missing return statement in function *)
exception VariableNotFoundException of string (* Using an undeclared variable *)
exception FunctionNotFoundException of string (* Using an undeclared function *)
exception BinopException of string            (* Mismatching types in binop *)
exception OtherException of string            (* Misc exception *)
exception AssignmentException of string       (* Assigning incompatable things *)
exception FunctionException of string         (* Incorrect call to built in functions *)
exception WhileExprException of string        (* Incorrect specification of while *)
exception IfExprException of string           (* Incorrect specification of if *)
exception ReturnTypeException of string       (* Mismatching return statement *)
exception MainNotFoundException of string     (* Missing main function *)
exception Error of string                     (* Placeholder - TODO remove *)

(* HELPER FUNCTION: convert normal types to return types*)
(* For simplicity we "used return type" everywhere in semantic checking *)
```

```
let var_taty_to_ret_taty t = match t with
    Int     -> RInt
  | Float   -> RFloat
  | Char    -> RChar
  | String  -> RString
  | Boolean -> RBoolean
   (*TaML exclusive*)
  | Table   -> RTable
  | Line    -> RLine
  | Cell    -> RCell

(* HELPER FUNCTION: return types to string representation*)
let taty_ret_to_string tp = match tp with
    RInt                  -> "RInt"
  | RFloat          -> "RFloat"
  | RChar           -> "RChar"
  | RString  -> "RString"
  | RBoolean -> "RBoolean"
   (*TaML exclusive*)
  | RTable          -> "RTable"
  | RLine           -> "RLine"
  | RCell           -> "RCell"
  | RVoid           -> "RVoid"

type symbol_table = {
  parent : symbol_table option;
  variables : (string * taty_ret) list
}

(* all function names are global -- no nested funcs allowed, so  *)
(* just build a table of all functions upon program initialization*)
type function_table = {
  functions: (string * taty_ret * taty_var list * stmt) list
}

type translation_environment = {
  return_type: taty_ret;               (* Function' ™s return type *)
  return_seen: bool;                   (* Have we seen a return statement *)
  var_scope: symbol_table;             (* symbol table for vars *)
  und_scope: symbol_table;             (* symbol table for underlying types *)
  fun_scope: function_table;           (* list of functions *)
  break_possible: bool;                (* can we break out of this *)
}
```

```
(* HEPLER FUNCTION: Add variable to an envirionment's var_scope*)
let add_to_var_table env s t =
  let new_vars = (s,t) :: env.var_scope.variables in
  let new_sym_table = {parent = env.var_scope.parent; variables = new_vars} in
  let new_env = {env with var_scope = new_sym_table} in
  new_env

(* HEPLER FUNCTION: Add variable to an envirionment's und_scope*)
let add_to_und_table env s t =
  let new_vars = (s,t) :: env.und_scope.variables in
  let new_sym_table = {parent = env.und_scope.parent; variables = new_vars} in
  let new_env = {env with und_scope = new_sym_table} in
  new_env

(* HEPLER FUNCTION: Search for variable in the symbol tables *)
let rec find_var_table (var_scope : symbol_table) name =
  try
    List.find (fun (s,_) -> s = name) var_scope.variables
  with Not_found ->
    match var_scope.parent with
        Some(parent) -> find_var_table parent name
      | _ -> raise Not_found

(* HEPLER FUNCTION: Search for variable in the symbol tables in "underlying type" *)
let rec find_und_table (und_scope : symbol_table) name =
  try
    (* let () = List.iter (fun (s,_) -> print_string ("Checked: "^s)) und_scope.variables in *)
    List.find (fun (s,_) -> s = name) und_scope.variables
  with Not_found ->
    match und_scope.parent with
        Some(parent) -> find_und_table parent name
      | _ -> raise Not_found

(* HEPLER FUNCTION: Search for function in our environment *)
let rec find_function (fun_scope : function_table) name =
    List.find (fun (s,_,_,_) -> s = name) fun_scope.functions

let get_better_und_type t1 t2 = match (t1,t2) with
    (RCell, _) as a -> snd a
  | (_, RCell) as a -> fst a
  | (_,_) as a -> fst a

(* HEPLER FUNCTION: Represent all table/line/cell return types as       *)
(* "cell return type" for simplicity.  Whether these hold int/float *)
```

```
(* can only be dynamically determined at run time so this can't         *)
(* be semantically checked at the moment *)
let default_und_type tp = match tp with
    RTable   -> RCell
  | RLine    -> RCell
  | RCell    -> RCell
  | _ as tp -> tp

(* HEPLER FUNCTION *)
let get_type_for_literal l = match l with
    IntL(l)         -> Int
  | FloatL(l)       -> Float
  | CharL(l)        -> Char
  | BoolL(l)        -> Boolean
  | StringL(l)      -> String

(* HEPLER FUNCTION: check variable v has type t *)
let check_type_of_expr env v t =
  let (id, id_t) = find_var_table env.var_scope v in
  if (id_t = t) then true else false

(* HEPLER FUNCTION: check both sides of binop are semantically compatible*)
let check_compatable_types_binop t1 t2 = match (t1,t2) with
    (RCell, RInt)                    -> true
  | (RInt, RCell)                    -> true
  | (RFloat, RCell)                  -> true
  | (RCell, RFloat)                  -> true
  | (RInt, RInt)                     -> true
  | (RFloat, RFloat)                 -> true
  | (RBoolean, RBoolean)             -> true
  | (RCell, RCell)                   -> true
  | _                                -> false

(* HEPLER FUNCTION: check both sides of assign are semantically compatible*)
let check_compatable_types_assign t1 t2 = match (t1,t2) with
    (RCell, RInt)   -> true
  | (RInt, RCell)   -> true
  | (RFloat, RCell) -> true
  | (RCell, RFloat) -> true
  | (RLine, RInt)   -> true
  | (RLine, RFloat) -> true
  | (tp, fp) -> if (tp = fp) then true else false

(* Get type of an expression so we can semantically check it *)
```

```
let rec get_type_for_expr env e = match e with
      | Literal(i) ->
            let t = var_taty_to_ret_taty (get_type_for_literal i) in
            (t,t)
      | TableAsTable(s,e1,e2,e3,e4) ->
            (* Check table is indexed with integers*)
            let t1 = get_type_for_expr env e1 and
            t2 = get_type_for_expr env e2 and
            t3 = get_type_for_expr env e3 and
            t4 = get_type_for_expr env e4    in
            if not ((fst t1) = RInt) ||
                  not ((fst t2) = RInt) ||
                  not ((fst t3) = RInt) ||
                  not ((fst t4) = RInt)     then
            raise (Error("Table '" ^ s ^ "' was indexed without Integers" )));
            let res = try
                  check_type_of_expr env s RTable
            with Not_found ->
                  raise (Error("Undeclared Table " ^ s)) in
            (if not res then
                  raise (Error("Non-Table indexed as table " ^ s)));
            let (_, und_type) = find_und_table env.und_scope s in
                  (RTable, und_type)
      | TableAsLine_R(s,e1,e2,e3) ->
            (* Check table is indexed with integers*)
            let t1 = get_type_for_expr env e1 and
            t2 = get_type_for_expr env e2 and
            t3 = get_type_for_expr env e3 in
            (if not ((fst t1) = RInt) ||
                  not ((fst t2) = RInt) ||
                  not ((fst t3) = RInt) then
            raise (Error("Table '" ^ s ^ "' was indexed without Integers" )));
            let res = try
                  check_type_of_expr env s RTable
            with Not_found ->
                  raise (Error("Undeclared Table " ^ s)) in
            (if not res then
                  raise (Error("Non-Table indexed as table " ^ s)));
            let (_, und_type) = find_und_table env.und_scope s in
                  (RLine, und_type)
      | TableAsLine_L(s,e1,e2,e3) ->
            (* Check table is indexed with integers*)
            let t1 = get_type_for_expr env e1 and
            t2 = get_type_for_expr env e2 and
```

```
        t3 = get_type_for_expr env e3 in
        (if not ((fst t1) = RInt) ||
              not ((fst t2) = RInt) ||
              not ((fst t3) = RInt) then
        raise (Error("Table '" ^ s ^ "' was indexed without Integers" )));
        let res = try
              check_type_of_expr env s RTable
        with Not_found ->
              raise (Error("Undeclared Table " ^ s)) in
        (if not res then
              raise (Error("Non-Table indexed as table " ^ s)));
        let (_, und_type) = find_und_table env.und_scope s in
              (RLine, und_type)
  | TableAsCell(s,e1,e2) ->
        (* Check table is indexed with integers*)
        let t1 = get_type_for_expr env e1 and
        t2 = get_type_for_expr env e2 in
        (if not ((fst t1) = RInt) ||
              not ((fst t2) = RInt) then
        raise (Error("Table '" ^ s ^ "' was indexed without Integers" )));
        let res = try
              check_type_of_expr env s RTable
        with Not_found ->
              raise (Error("Undeclared Table " ^ s)) in
        (if not res then
              raise (Error("Non-Table indexed as table " ^ s)));
        let (_, c_type) = find_und_table env.und_scope s in
              (RCell, c_type)
| TableAsCellValue(s,e1,e2) ->
        (* Check table is indexed with integers*)
        let t1 = get_type_for_expr env e1 and
        t2 = get_type_for_expr env e2 in
        (if not ((fst t1) = RInt) ||
              not ((fst t2) = RInt) then
        raise (Error("Table '" ^ s ^ "' was indexed without Integers" )));
        let res = try
              check_type_of_expr env s RTable
        with Not_found ->
              raise (Error("Undeclared Table " ^ s)) in
        (if not res then
              raise (Error("Non-Table indexed as table " ^ s)));
        let (_, c_type) = find_und_table env.und_scope s in
              (RCell, c_type)
  | LineAsLine(s,e1,e2) ->
```

```
        (* Check line is indexed with integers*)
        let t1 = get_type_for_expr env e1 and
        t2 = get_type_for_expr env e2 in
        (if not ((fst t1) = RInt) ||
              not ((fst t2) = RInt) then
        raise (Error("Line '" ^ s ^ "' was indexed without Integers" )));
        let res = try
              check_type_of_expr env s RLine
        with Not_found ->
              raise (Error("Undeclared Line " ^ s)) in
        (if not res then
        raise (Error("Non-Line indexed as line " ^ s)));
        let (_, c_type) = find_und_table env.und_scope s in
        (RLine, c_type)
  | LineAsCell(s,e1) ->
        (* Check line is indexed with integers*)
        let t1 = get_type_for_expr env e1 in
        (if not ((fst t1) = RInt) then
              raise (Error("Line '" ^ s ^ "' was indexed without Integers" )));
        let res = try
              check_type_of_expr env s RLine
        with Not_found ->
              raise (Error("Undeclared Line " ^ s)) in
        (if not res then
              raise (Error("Non-Line indexed as line " ^ s)));
        let (_, c_type) = find_und_table env.und_scope s in
              (RCell, c_type)
| LineAsCellValue(s,e1) ->
        (* Check line is indexed with integers*)
        let t1 = get_type_for_expr env e1 in
        (if not ((fst t1) = RInt) then
              raise (Error("Line '" ^ s ^ "' was indexed without Integers" )));
        let res = try
              check_type_of_expr env s RLine
        with Not_found ->
              raise (Error("Undeclared Line " ^ s)) in
        (if not res then
              raise (Error("Non-Line indexed as line " ^ s)));
        let (_, c_type) = find_und_table env.und_scope s in
              (RCell, c_type)
  | CellAsCell(s) ->
        let res = try
              check_type_of_expr env s RCell
        with Not_found ->
```

```
                raise (Error("Undeclared Line " ^ s)) in
            (if not res then
                raise (Error("Non-Line indexed as line " ^ s)));
            let (_, c_type) = find_und_table env.und_scope s in
                (RCell, c_type)
      | NewTotalCell(s) ->
            let res = try
                (check_type_of_expr env s RCell) || (check_type_of_expr env s RLine) || (check_type_of_expr env s
RTable)
            with Not_found ->
                raise (Error("Undeclared Table/Line/Cell " ^ s)) in
            (if not res then
                raise (Error("^id need id to be table/line/cell " ^ s)));
            let (_, c_type) = find_und_table env.und_scope s in
                (RCell, c_type)
      | Id(s) ->
            let (_, s_type) = try
                find_var_table env.var_scope s (* locate a variable by name *)
            with Not_found ->
                raise (Error("Undeclared 1 Identifier " ^ s))
            in let (_, u_type) = try
                find_und_table env.und_scope s (* locate a variable by name *)
            with Not_found ->
                raise (Error("Undeclared 2 Identifier " ^ s))
            in (s_type, u_type)
      | Binop(e1,op,e2) ->
        let t1 = get_type_for_expr env e1
        and t2 = get_type_for_expr env e2 in
        (if not (check_compatable_types_binop (fst t1) (fst t2)) then
            raise (Error("Mismatch in underlying types for binary operator: " ^ taty_ret_to_string (fst t1) ^ ":" ^
           taty_ret_to_string (fst t2))));
        (if ((op = And || op = Or) && not ((fst t1) = RBoolean)) then
            raise (Error("Non Boolean argument passed to And / Or")));
        if op = Equal || op = Neq || op = Less_than || op = Leq || op = Greater_than || op = Geq then
            (RBoolean, get_better_und_type (snd t1) (snd t2))
        else
            t1
      | Assign(e1,e2) ->
        let t1 = get_type_for_expr env e1
        and t2 = get_type_for_expr env e2 in
        (if not (check_compatable_types_assign (fst t1) (fst t2)) then
            raise (Error("Mismatch in types for assignment")));
        (fst t1, get_better_und_type (snd t1) (snd t2))
      | Call(s,el) ->
```

```
            let (fname, fret, fargs, fbody) = try
                find_function env.fun_scope s
            with Not_found ->
                raise (Error("Undeclared Function " ^ s)) in
            let el_ty = List.map (fun s -> fst (get_type_for_expr env s)) el in
            let fn_ty = List.map (fun s -> var_taty_to_ret_taty s) fargs in
            (* TODO - allow Cells to be passed as int / float *)
            (if not (el_ty = fn_ty) then
                raise (Error("Mismatching types in function call for: " ^ s)));
            (fret, default_und_type fret)
        | Uniop(op,e1) ->
            let t1 = get_type_for_expr env e1 in
            (if not ((fst t1) = RBoolean) then
                raise (Error("Unary op applied on non-boolean")));
             t1
        | All -> (RInt, RInt)
        | Null -> (RInt, RInt)
        | UnaryMinus(e1) ->
            let ut = get_type_for_expr env e1 in
            (if not ((fst ut) = RInt || (fst ut) = RFloat || (fst ut) = RBoolean || (fst ut) = RCell) then
                raise (Error("Unary Minus used on non Int / Float / Boolean")));
            ut

let rec get_sexpr_for_expr env e =
  let (t1,t2) = get_type_for_expr env e in
  ExprType(e,t1,t2)


(* THREE HELPER FUNCTIONS: Gets type/name/underlying type *)
(* from a variable's signature *)
let get_types_from_varSig vs =
  let VarSig(ty,nm) = vs in
  ty

let get_name_undtype_from_varSig vs =
  let VarSig(ty,nm) = vs in
  (nm, (default_und_type (var_taty_to_ret_taty ty)))

let get_name_type_from_varSig vs =
  let VarSig(ty,nm) = vs in
  (nm, (var_taty_to_ret_taty ty))


(* HELPER FUNCTION: Add function to env and return updated env *)
```

```
let add_func_to_env env func_inst =
  let f_table = env.fun_scope in
  let old_functions = f_table.functions in
  let FuncSig(rt, nm) = func_inst.fname in
  let al = List.map (fun vs -> get_types_from_varSig vs) func_inst.formals in
  let fun_stmt = func_inst.body in
  let new_functions = (nm, rt, al, fun_stmt)::old_functions in
  let new_fun_scope = {functions = new_functions} in
  let new_env = {env with fun_scope = new_fun_scope} in
  new_env


(* HELPER FUNCTION: Add a newly initialized var to env and return updated env *)
let add_init_to_env env init_inst = (* TODO - check that the variable being assigned to matches the type of thing
being assigned to it *)
  (* Note that if we are initializing a Cell, then it can be initialized from either a float or int or cell *)
  let env = match init_inst with
    | NullInit(t,s) -> add_to_var_table env s (var_taty_to_ret_taty t)
    | VarInit(t,s,e) -> add_to_var_table env s (var_taty_to_ret_taty t)
    | TableInit(t1,s,e1,e2,t2) -> add_to_var_table env s (var_taty_to_ret_taty t1) in
  let (tp,env) = match init_inst with
    | VarInit(t,s,e) ->
      let env = add_to_und_table env s (snd (get_type_for_expr env e)) in
      (snd (get_type_for_expr env e), env)
    | TableInit(t1,s,e1,e2,t2) ->
      let env = add_to_und_table env s (var_taty_to_ret_taty t2) in
      ((var_taty_to_ret_taty t2), env)
    | NullInit(t,s) ->
      let env = add_to_und_table env s (default_und_type (var_taty_to_ret_taty t)) in
      ((var_taty_to_ret_taty t), env) in
  (tp, env)

(* HELPER FUNCTION: Combine two environments*)
let rec combine_env env lenv =
  let ret_seen = List.fold_left (fun a e -> a && e.return_seen) true lenv in
  let new_env = {env with return_seen = ret_seen} in
  new_env

(* Get type of a statement so we can semantically check it *)
let rec get_env_for_stmt env st = match st with
      | Block(sl) ->
            (* If we enter a block, set break to 1, and iterate variables *)
            let new_var_scope = {parent = Some(env.var_scope); variables = [] } in
            let new_env = {env with var_scope = new_var_scope} in
```

```
            let getter (env, acc) s =
                  let (st, ne) = get_env_for_stmt env s in
                  (ne, st::acc) in
                  let (ls, st) = List.fold_left (fun e s -> getter e s) (new_env,[]) sl in
                  let revst = List.rev st in
            (SBlock(revst), ls)
      | Expr(e) ->
            let _ = get_type_for_expr env e in
            (SExpr(get_sexpr_for_expr env e), env)
      | Return(e) ->
            let t = get_type_for_expr env e in
            (if not ((fst t) = env.return_type) then
                  raise (Error("Incompatable Return type")));
            let new_env = {env with return_seen = true} in
            (SReturn(get_sexpr_for_expr env e), new_env)
      | Print(e) ->
            let _ = get_type_for_expr env e in
            (SPrint(get_sexpr_for_expr env e), env)
      | If(e,s1,s2) ->
            let t = get_type_for_expr env e in
            (if not ((fst t) = RBoolean) then
                  raise (Error("Non-boolean IF predicate")));
            let new_env = {env with break_possible = true} in
            let (st1, new_env1) = get_env_for_stmt new_env s1
            and (st2, new_env2) = get_env_for_stmt new_env s2 in
            let ret_seen = (new_env1.return_seen && new_env2.return_seen) in
            let new_env = {env with return_seen = ret_seen} in
            (SIf((get_sexpr_for_expr env e), st1, st2), new_env)
      | For(e1,e2,e3,s) ->
            let t1 = fst (get_type_for_expr env e1)
            and t2 = fst (get_type_for_expr env e2)
            and t3 = fst (get_type_for_expr env e3) in
            (if not (t1 = RInt && t3 = RInt && t2 = RBoolean) then
                  raise (Error("Bad FOR loop, not integer bounds")));
            (if not (t1 = t3) then
                  raise (Error("Bad FOR loop, can not assign to iterator")));
            let new_env = {env with break_possible = true} in
            let (st, new_env) = get_env_for_stmt new_env s in
                  (SFor((get_sexpr_for_expr env e1), (get_sexpr_for_expr env e2), (get_sexpr_for_expr env e3), st),
new_env)
      | While(e,s) ->
            let t = fst (get_type_for_expr env e) in
            (if not (t = RBoolean) then
                  raise (Error("Bad WHILE loop, not integer bounds")));
```

```
            let new_env = {env with break_possible = true} in
            let (st, new_env) = get_env_for_stmt new_env s in
                  (SWhile((get_sexpr_for_expr env e), st), new_env)
      | NoElseMark -> (SNoElseMark, env)
      | VarDecl(i) ->
            let (si, new_env) = add_init_to_env env i in
            (SVarDecl(SInit(i,si)), new_env)
      | Break ->
            (if not (env.break_possible) then
                  raise (Error("Bad BREAK statement")));
            (SBreak, env)


(* Add all global vars and functions to our environment *)
let initialize_globals env constr = match constr with
    GlobalVar(i) ->
      let (_, new_env) = add_init_to_env env i in
      new_env
  | FuncDef(f) ->
      let new_env = add_func_to_env env f in
      new_env

(* Check that return statements are in order, etc. *)
let check_final_env env = (* TODO - do we need to do anything else here? *)
  (if (false = env.return_seen && env.return_type <> RVoid) then
    raise (Error("Missing Return Statement")));
  true

(* Semantic checking on a function *)
let check_funcs env ct = match ct with
    GlobalVar(v) ->
      let (tp, _) = add_init_to_env env v in
      SGlobalVar(SInit(v, tp))
  | FuncDef(f) ->
      let FuncSig(ret_type,fun_name) = f.fname in
      (* let () = print_string ("Checking function: " ^ fun_name ^ "\n\n") in *)
      let new_variables = List.fold_left (fun a vs -> (get_name_type_from_varSig vs)::a) [] f.formals in
      let new_und_variables = List.fold_left (fun a vs -> (get_name_undtype_from_varSig vs)::a) [] f.formals in
      let new_var_scope = {parent = Some(env.var_scope); variables = new_variables} in
      let new_und_scope = {parent = Some(env.und_scope); variables = new_und_variables} in
      let new_env = {return_type = ret_type; return_seen = false; var_scope = new_var_scope; und_scope =
new_und_scope; fun_scope
            = env.fun_scope; break_possible = false} in
      let (stbody, final_env) = get_env_for_stmt new_env f.body in
```

```
      let _ = check_final_env final_env in
      let sfunc = {sfname = f.fname; sformals = f.formals; sbody = stbody} in
      SFuncDef(sfunc)

(* Empty initializations for tables/environments *)
let empty_symbol_table = {parent = None; variables = []}
let empty_function_table = {functions = []}
let empty_env = {return_type = RVoid; return_seen = false; var_scope = empty_symbol_table; und_scope =
empty_symbol_table; fun_scope = empty_function_table; break_possible = false}


(* MAIN RUNNER FUNCTION *)
let translate prog =
  let env = List.fold_left (fun env ct -> initialize_globals env ct) empty_env prog in
  let res = List.map (fun ct -> check_funcs env ct) prog in
  res
```

---

## 8.8 Translate_utils.ml

```
(* translate_utils.ml - This utility is used for testing purposes.     *)
(* It outputs the results of an SAST in string-readable form          *)
(* Author: Adam Dossa *)

open Type
open Ast
open Sast

let string_of_taty_var = function
    Int -> "TYPE_INT"
  | Float -> "TYPE_FLOAT"
  | Char -> "TYPE_CHAR"
  | String -> "TYPE_STRING"
  | Boolean -> "TYPE_BOOL"
  | Table -> "TYPE_TABLE"
  | Line -> "TYPE_LINE"
  | Cell -> "TYPE_CELL"

let string_of_taty_ret = function
    RInt -> "TYPE_INT"
  | RFloat -> "TYPE_FLOAT"
  | RChar -> "TYPE_CHAR"
  | RString -> "TYPE_STRING"
```

```
  | RBoolean -> "TYPE_BOOL"
  | RTable -> "TYPE_TABLE"
  | RLine -> "TYPE_LINE"
  | RCell -> "TYPE_CELL"
  | RVoid -> "TYPE_VOID"

let string_of_literal = function
    IntL(l) -> "LITERAL_INT:[" ^ (string_of_int l) ^ "]"
  | FloatL(l) -> "LITERAL_FLOAT:[" ^ (string_of_float l) ^ "]"
  | CharL(l) -> "LITERAL_CHAR:[" ^ (Char.escaped l) ^ "]"
  | BoolL(l) -> "LITERAL_BOOL:[" ^ (if l then "#TRUE#" else "#FALSE#") ^ "]"
  | StringL(l) -> "LITERAL_STRING:[" ^ l ^ "]"

let string_of_binop = function
   Add -> "ADD"
 | Sub -> "SUB"
 | Mult -> "MULT"
 | Div -> "DIV"
 | Equal -> "EQUAL"
 | Neq -> "NEQ"
 | Less_than -> "LT"
 | Leq -> "LTE"
 | Greater_than -> "GT"
 | Geq -> "GTE"
 | Or -> "OR"
 | And -> "AND"
 | Mod ->"MOD"

let string_of_uniop = function
   Not -> "NOT"

let rec string_of_expr = function
    Literal(l) -> "EXPR_LITERAL:[" ^ string_of_literal l ^ "]"
  | TableAsTable(s,e1,e2,e3,e4) -> "EXPR_TABLE_AS_TABLE:[" ^ s ^ "," ^ (string_of_expr e1) ^ "," ^ (string_of_expr
e2) ^ "," ^
            (string_of_expr e3) ^ "," ^ (string_of_expr e4) ^ "]"
  | TableAsLine_L(s,e1,e2,e3) -> "EXPR_TABLE_AS_LINE_L:[" ^ s ^ "," ^ (string_of_expr e1) ^ "," ^ (string_of_expr e2)
^ "," ^
            (string_of_expr e3) ^ "]"
  | TableAsLine_R(s,e1,e2,e3) -> "EXPR_TABLE_AS_LINE_R:[" ^ s ^ "," ^ (string_of_expr e1) ^ "," ^ (string_of_expr e2)
^ "," ^
            (string_of_expr e3) ^ "]"
  | TableAsCell(s,e1,e2) -> "EXPR_TABLE_AS_CELL:[" ^ s ^ "," ^ (string_of_expr e1) ^ "," ^ (string_of_expr e2) ^ "]"
```

```
  | TableAsCellValue(s,e1,e2) -> "EXPR_TABLE_AS_CELL_Value:[" ^ s ^ "," ^ (string_of_expr e1) ^ "," ^ (string_of_expr
e2) ^ "]"
  | LineAsLine(s,e1,e2) -> "EXPR_LINE_AS_LINE:[" ^ s ^ "," ^ (string_of_expr e1) ^ "," ^ (string_of_expr e2) ^ "]"
  | LineAsCell(s,e1) -> "EXPR_LINE_AS_CELL:[" ^ s ^ "," ^ (string_of_expr e1) ^ "]"
  | LineAsCellValue(s,e1) -> "EXPR_LINE_AS_CELL_Value:[" ^ s ^ "," ^ (string_of_expr e1) ^ "]"
  | CellAsCell(s) -> "EXPR_CELL_AS_CELL:[" ^s^"]"
  | NewTotalCell(s) -> "EXPR_New_Total_CELL:[" ^s^"]"
  | Id(s) -> "EXPR_ID:[" ^ s ^ "]"
  | Binop(e1, o, e2) -> "EXPR_BINOP:[" ^ string_of_expr e1 ^ "," ^ string_of_binop o ^ "," ^ string_of_expr e2 ^ "]"
  | Assign(e1, e2) -> "EXPR_ASSIGN:[" ^ string_of_expr e1 ^ "," ^ string_of_expr e2 ^ "]"
  | Call(f, el) ->
    "EXPR_CALL:[" ^ f ^ ",LIST:[" ^ String.concat ", " (List.map string_of_expr el) ^ "]]"
  | Uniop(o, expr) -> "EXPR_UNIOP:[" ^ string_of_uniop o ^ "," ^ string_of_expr expr ^ "]"
  | All -> "ALL"
  | Null -> "NULL"
  | UnaryMinus(e) -> "EXPR_UNARYMINUS:[" ^ string_of_expr e ^ "]"

let string_of_sexpr = function
    ExprType(e, t1,t2) -> "(EXPR_TYPE:" ^ string_of_taty_ret t1 ^ ","^ string_of_taty_ret t2 ^ ")" ^ string_of_expr e

let string_of_init = function
    NullInit(t, s) -> "INIT_NULL:[" ^ string_of_taty_var t ^ "," ^ s ^ "]"
  | VarInit(t,s,e) -> "INIT_VAR:[" ^ string_of_taty_var t ^ "," ^ s ^ "," ^ (string_of_expr e) ^ "]"
  | TableInit(t1,s,e1,e2,t2) -> "INIT_TABLE:[" ^ string_of_taty_var t1 ^ "," ^ s ^ "," ^ string_of_expr e1 ^ ", " ^
string_of_expr
            e2 ^ ", " ^ string_of_taty_var t2 ^ "]"

let string_of_sinit = function
    SInit(i,t) -> "(INIT_TYPE:" ^ string_of_taty_ret t ^ ")" ^ string_of_init i

let rec string_of_sstmt = function
    SBlock(stmts) ->
      "STMT_BLOCK:[LIST:[" ^ String.concat "," (List.map string_of_sstmt stmts) ^ "]]"
  | SExpr(expr) -> "STMT_EXPR:[" ^ string_of_sexpr expr ^ "]";
  | SReturn(expr) -> "STMT_RETURN:[" ^ string_of_sexpr expr ^ "]";
  | SPrint(expr) -> "STMT_PRINT:[" ^ string_of_sexpr expr ^ "]";
  | SIf(e, s, SBlock([])) -> "STMT_IF_NOELSE:[" ^ string_of_sexpr e ^ "," ^ string_of_sstmt s ^ "]"
  | SIf(e, s1, s2) ->  "STMT_IF:[" ^ string_of_sexpr e ^ "," ^
    string_of_sstmt s1 ^ "," ^ string_of_sstmt s2 ^ "]"
  | SFor(e1, e2, e3, s) ->
    "STMT_FOR:[" ^ string_of_sexpr e1  ^ "," ^ string_of_sexpr e2 ^ "," ^
    string_of_sexpr e3  ^ "," ^ string_of_sstmt s ^ "]"
  | SWhile(e, s) -> "STMT_WHILE:[" ^ string_of_sexpr e ^ "," ^ string_of_sstmt s ^ "]"
  | SNoElseMark -> "STMT_NOELSEMARK"
```

```
  | SVarDecl(s)->string_of_sinit s
  | SBreak -> "STMT_BREAK"

let string_of_funcSig = function
  FuncSig(t, s) -> "FUNCSIG:[" ^ string_of_taty_ret t ^ "," ^ s ^ "]"

let string_of_varSig = function
  VarSig(t, s) -> "VARSIG:[" ^ string_of_taty_var t ^ "," ^ s ^ "]"

let string_of_sfunc_decl fdecl =
  "FUNCDECL:[" ^ string_of_funcSig fdecl.sfname ^ ",LIST:[" ^
  String.concat "," (List.map string_of_varSig fdecl.sformals) ^ "],LIST:[" ^
  string_of_sstmt fdecl.sbody ^ "]]"

let string_of_sconstruct = function
    SGlobalVar(i) -> "CONSTRUCT_GLOBALVAR:[" ^ (string_of_sinit i) ^ "]"
  | SFuncDef(f) -> "CONSTRUCT_FUNCDEF:[" ^ (string_of_sfunc_decl f) ^ "]"

let string_of_sprogram sconstruct_list =
  "PROGRAM:[LIST:[" ^ (String.concat "," (List.map string_of_sconstruct sconstruct_list)) ^ "]]\n"
```

---

## 8.9 Sast.mli

```
(* sast.mli -- Semanitcally Checked and Typed Abstract Syntax Tree *)
(* Authors: Adam Dossa  *)

open Type
open Ast

(* Expressions *)
type sexpr =
  ExprType of expr * taty_ret * taty_ret

type  sinit =
  SInit of init * taty_ret

type sstmt =
    SBlock of sstmt list
  | SExpr of sexpr                          (*foo = bar + 3; *)
  | SReturn of sexpr                        (* return 42 also includes return function_name *)
  | SPrint of sexpr
  | SIf of sexpr * sstmt * sstmt            (* if (foo == 42) {} else {} *)
```

```
  | SFor of sexpr * sexpr * sexpr * sstmt     (* for loop *)
  | SWhile of sexpr * sstmt                   (* while (i<10) { i = i + 1 } *)
  | SNoElseMark                               (* For loops without else*)
  | SVarDecl of sinit
  | SBreak

type sfunc_decl = {
            sfname : funcSig;                 (* Name and return type of the function *)
            sformals : varSig list;           (* Formal arguments, type & names *)
            sbody : sstmt;
}

(* Bascially, the parts of a program *)
type sconstruct =
    SGlobalVar of sinit
  | SFuncDef of sfunc_decl

type sprogram = sconstruct list (* global vars, funcs *)
```

---

# 8.10 Jpp.ml

```
(* jpp.ml -- Java pretty printer.  Prints TaML code into corresponding Java *)
(* Authors: Le Chang, Quizi Shangguan  *)

open Ast
open Sast
open Type

let default_keyword = "public static"         (*put before  global variables and functions*)
let default_implicit_param = "null"           (*for null declaration*)

(*translate binary operators*)
let string_of_operator = function
        Add         -> "+"       | Sub -> "-"   | Mult       -> "*"       | Div -> "/"
      | Equal       -> "=="      | Neq -> "!=" | Less_than -> "<"       | Leq -> "<="
      | Greater_than -> ">"      | Geq -> ">=" | Or         ->"||"       | And -> "&&"
      | Mod         ->"%"

(*translate unary operators*)
let string_of_operator1 = function
      Not -> "!"
```

```
(*helper function to translte char into string*)
let string_of_char =
      String.make 1

(*translate all Literatls, 1. 3.4f "hello"*)
let string_of_literal = function
    IntL(i) -> string_of_int i
  | FloatL(i) -> string_of_float i^"f"
  | CharL(i) -> "\'" ^ string_of_char i ^ "\'"
  | StringL(i) -> "\"" ^ i ^ "\""
  | BoolL(i) ->string_of_bool i

(*translate return types of taml*)
let string_of_type_return = function
    RInt -> "Integer"
  | RFloat ->"Float"
  | RChar  ->"Character"
  | RString ->"String"
  | RBoolean ->"Boolean"
   (*TaML exclusive*)
  | RTable ->"Table"
  | RLine ->"Line"
  | RCell ->"Cell"
  | RVoid ->"void"

(*translate tyeps*)
let string_of_type =function
    Int ->"Integer"
  | Float ->"Float"
  | Char ->"Character"
  | String ->"String"
  | Boolean ->"Boolean"
   (*TaML exclusive*)
  | Table ->"Table"
  | Line ->"Line"
  | Cell ->"Cell"

(* translate expr into Java code, this function is a helper function *)
(* of string_of_exprs which take Sast.ExprType as the input*)
let rec string_of_expr = function
  | Literal(l) -> string_of_literal l
  | TableAsCellValue(s,e1,e2) -> s^".getCellValue("^string_of_expr e1^","^string_of_expr e2^")"
  | LineAsCellValue(s,e) -> s^".getCellValue("^string_of_expr e^")"
  | CellAsCell(s) -> s^".getVal()"
```

```
  | TableAsCell(s,e1,e2) -> s^".getCell("^string_of_expr e1^","^string_of_expr e2^")"
  | LineAsCell(s,e) -> s^".getCell("^string_of_expr e^")"
  | Id(s) -> s
  | Uniop(op, e) -> string_of_operator1(op) ^ "(" ^ string_of_expr e ^ ")"
  | Binop(e1, o, e2) ->string_of_expr e1 ^ " " ^ string_of_operator(o) ^ " " ^ string_of_expr e2

      (*all table related assignment needs to be translated independently*)
  | Assign(name,TableAsTable(s,e1,e2,e3,e4))  ->    string_of_expr name^"="^s^".createTableCopy("^ string_of_expr
            e1^","^string_of_expr e2^","^string_of_expr e3^","^string_of_expr e4^")"
  | Assign(name,TableAsLine_R(s,e1,e2,e3))    ->    string_of_expr name^"= new Line();\n "^string_of_expr
            name^".assignLine("^s^","^string_of_expr e1^","^string_of_expr e1^","^string_of_expr
e2^","^string_of_expr e3^")"
  | Assign(name,TableAsLine_L(s,e1,e2,e3))    ->    string_of_expr name^"= new Line();\n "^string_of_expr
             name^".assignLine("^s^","^string_of_expr e1^","^string_of_expr e2^","^string_of_expr
      e3^","^string_of_expr e3^")"
  | Assign(name,TableAsCell(s,e1,e2))   ->    string_of_expr name^"="^s^".getCell("^string_of_expr
e1^","^string_of_expr e2^")"
  | Assign(name,LineAsCell(s,e1))       ->    string_of_expr name^"="^s^".getCell("^string_of_expr e1^")"
  | Assign(TableAsCellValue(s,e1,e2),name)    ->    s^".setVal("^string_of_expr e1^","^string_of_expr
e2^","^string_of_expr
            name^")"
  | Assign(TableAsTable(s,e1,e2,e3,e4),name)  ->    s^".setVal("^string_of_expr e1^","^string_of_expr
            e2^","^string_of_expr e3^","^string_of_expr e4^","^string_of_expr name^")"
  | Assign(TableAsLine_R(s,e1,e2,e3),name)    ->    s^".setVal("^string_of_expr e1^","^string_of_expr
e1^","^string_of_expr
            e2^","^string_of_expr e3^","^string_of_expr name^")"
  | Assign(TableAsLine_L(s,e1,e2,e3),name)    ->    s^".setVal("^string_of_expr e1^","^string_of_expr
e2^","^string_of_expr
            e3^","^string_of_expr e3^","^string_of_expr name^")"
  | Assign(LineAsCellValue(s,e),name) -> s^".setVal("^string_of_expr e^","^string_of_expr name^")"
  | Assign(LineAsLine(s,e1,e2),name)   -> s^".setVal("^string_of_expr e1^","^string_of_expr e2^","^string_of_expr
name^")"
  | Assign(NewTotalCell(s),name) ->    s^".setVal("^string_of_expr name^")"
  | Assign(v, e) -> string_of_expr v ^ " = " ^ string_of_expr e
  | Call(f_Name,f_List) -> f_Name ^ "(" ^ String.concat "," (List.map string_of_expr f_List) ^ ")"
  | UnaryMinus (l)  ->    "-"^string_of_expr l
  | All      ->    "\"" ^ "ALL" ^ "\""
  | Null     ->    "null"
  | _        ->    "never get here!"

(*helper function that do explict casting on table related binary operations*)
let string_of_ascell expr tp =
  if (tp = RInt) || (tp = RFloat) then
      match expr with
```

```
          CellAsCell(i) -> "("^string_of_type_return tp^")"^ i^".getVal()"
        | TableAsCellValue(s,e1,e2) -> "(" ^ (string_of_type_return tp) ^ ") " ^s^".getCellValue("^string_of_expr
            e1^","^string_of_expr e2^")"
        | LineAsCellValue(s,e1) ->  "(" ^ (string_of_type_return tp) ^ ") " ^s^".getCellValue("^string_of_expr e1^")"
        | _ as e1 -> string_of_expr e1
    else
      string_of_expr expr

(*translate the expression from Sast, table related assignment needs to be translated independently*)
let rec string_of_exprs = function
    ExprType(  Binop(e1, o, e2),t,tp) -> (string_of_ascell e1 tp) ^ string_of_operator(o) ^ (string_of_ascell e2 tp)
  | ExprType(Assign(TableAsCellValue(s,e1,e2),expr),tp,tp2) -> s^".setVal("^string_of_expr e1^","^string_of_expr
             e2^","^"("^ (string_of_type_return tp2) ^ ") " ^string_of_expr expr^")"
  | ExprType(Assign(LineAsCellValue(s,e1),expr2),tp,tp2) -> s^".setVal("^string_of_expr e1^","^"("^
            (string_of_type_return tp2) ^ ")"^ string_of_expr expr2^")"
  | ExprType(Assign(CellAsCell(s),expr),tp,tp2) -> s^".setVal("^"("^ (string_of_type_return tp2) ^
            ")"^string_of_expr expr^")"
  | ExprType(Assign(name,TableAsCellValue(s,e1,e2)),tp,tp2) -> string_of_expr name^"= (" ^ (string_of_type_return tp)
            ^ ") " ^s^".getCellValue("^string_of_expr e1^","^string_of_expr e2^")"
  | ExprType(Assign(name,LineAsCellValue(s,e1)),tp,tp2) ->
            string_of_expr name^"= (" ^ (string_of_type_return tp) ^ ") " ^s^".getCellValue("^string_of_expr e1^")"
  | ExprType(Assign(name,CellAsCell(s)),tp,tp2) ->
            string_of_expr name^"= (" ^ (string_of_type_return tp) ^ ") " ^s^".getVal()"
  | ExprType(Assign(name,TableAsLine_R(s,e1,e2,e3)),t,tp2) ->
            string_of_expr name^"= new Line<"^string_of_type_return tp2^">();\n "^string_of_expr
            name^".assignLine("^s^","^string_of_expr e1^","^string_of_expr e1^","^string_of_expr
      e2^","^string_of_expr e3^")"
  | ExprType(Assign(name,TableAsLine_L(s,e1,e2,e3)),t,tp2) ->
            string_of_expr name^"= new Line<"^string_of_type_return tp2^">();\n "^string_of_expr
            name^".assignLine("^s^","^string_of_expr e1^","^string_of_expr e2^","^string_of_expr
      e3^","^string_of_expr e3^")"
  | ExprType(Assign(name,TableAsCell(s,e1,e2)),t,tp2) ->
            string_of_expr name^"= new Cell<"^string_of_type_return tp2^">();\n "^string_of_expr
            name^"="^s^".getCell("^string_of_expr e1^","^string_of_expr e2^")"
  | ExprType(Assign(name,LineAsCell(s,e1)),t,tp2) ->
            string_of_expr name^"= new Cell<"^string_of_type_return tp2^">();\n "^string_of_expr
            name^"="^s^".getCell("^string_of_expr e1^")"
  | ExprType(esp,_,_)     -> string_of_expr esp

(*translate initialization from Sast, table related init  needs to be translated independently*)
let string_of_init =function
      (NullInit(Char,s),_)->"char " ^" "^ s
  |   (NullInit(t,s),_)-> string_of_type t^" "^ s ^"=null"
  |   (VarInit(Char,s,Null),_)->"char " ^" "^ s
```

```
  |    (VarInit(Table,name,TableAsTable(s,e1,e2,e3,e4)),_) ->
             "Table "^name^"="^s^".createTableCopy("^ string_of_expr e1^","^string_of_expr e2^","^string_of_expr
             e3^","^string_of_expr e4^")"
  |    (VarInit(Line,name,TableAsLine_R(s,e1,e2,e3)),RInt) ->
             "Line<Integer> "^name^"= new Line<Integer>();\n"^name^".assignLine("^s^","^string_of_expr
e1^","^string_of_expr
             e1^","^string_of_expr e2^","^string_of_expr e3^")"
  |    (VarInit(Line,name,TableAsLine_R(s,e1,e2,e3)),RFloat) ->
             "Line<Float> "^name^"= new Line<Float>();\n"^name^".assignLine("^s^","^string_of_expr
e1^","^string_of_expr
             e1^","^string_of_expr e2^","^string_of_expr e3^")"
  |    (VarInit(Line,name,TableAsLine_L(s,e1,e2,e3)),RInt) ->
             "Line<Integer> "^name^"= new Line<Integer>();\n"^name^".assignLine("^s^","^string_of_expr
e1^","^string_of_expr
             e2^","^string_of_expr e3^","^string_of_expr e3^")"
  |    (VarInit(Line,name,TableAsLine_L(s,e1,e2,e3)),RFloat) ->
             "Line<Float> "^name^"= new Line<Float>();\n"^name^".assignLine("^s^","^string_of_expr
e1^","^string_of_expr
             e2^","^string_of_expr e3^","^string_of_expr e3^")"
  |    (VarInit(Line,name,LineAsLine(s,e1,e2)),RInt) ->
             "Line<Integer> "^name^"="^s^".createLineCopy("^string_of_expr e1^","^string_of_expr e2^")"
  |    (VarInit(Line,name,LineAsLine(s,e1,e2)),RFloat) ->
             "Line<Float> "^name^"="^s^".createLineCopy("^string_of_expr e1^","^string_of_expr e2^")"
  |    (VarInit(Cell,name,TableAsCell(s,e1,e2)),RInt) ->
             "Cell<Integer> "^name^"="^s^".getCell("^string_of_expr e1^","^string_of_expr e2^")"
  |    (VarInit(Cell,name,TableAsCell(s,e1,e2)),RFloat) ->
             "Cell<Float> "^name^"="^s^".getCell("^string_of_expr e1^","^string_of_expr e2^")"
  |    (VarInit(Cell,name,LineAsCell(s,e1)),RInt) ->
             "Cell<Integer> "^name^"="^s^".getCell("^string_of_expr e1^")"
  |    (VarInit(Cell,name,LineAsCell(s,e1)),RFloat) ->
             "Cell<Float> "^name^"="^s^".getCell("^string_of_expr e1^")"
  |    (VarInit(Cell,name,Literal(i)),t) ->
             "Cell<"^string_of_type_return t^"> "^name^"=new Cell<"^string_of_type_return
             t^">();\n"^name^".setVal("^string_of_literal i^")"
  |    (VarInit(Char,s,e),_) ->
             if(string_of_expr e="\'"^"\'")then  "char"^" "^s
                   else "char"^" "^s^"="^string_of_expr e
  |    (VarInit(t,s,e),_) ->
             string_of_type t^" "^s^"="^string_of_expr e

  |    (TableInit(t,s,e1,e2,taty),_)->"Table<"^string_of_type taty^"> "^ s^" = new Table<"^string_of_type
taty^">("^string_of_expr
             e1^","^string_of_expr e2^")"
```

```
(*wrapper function that takes Sast.init as input and pass the parameters to string_of_init*)
let rec string_of_inits =function
      SInit(i,t)-> string_of_init (i,t)

(*translation of statement*)
let rec string_of_stmt = function
    SBlock(stmts) ->"{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | SExpr(expr) -> string_of_exprs expr ^ ";\n"
  | SReturn(e) -> "return " ^ string_of_exprs e ^ ";\n"
  | SPrint (e) -> "Printers.print("^string_of_exprs e^");\n"
  | SWhile(e, s) -> "while (" ^ string_of_exprs e ^ ")\n" ^ string_of_stmt s
  | SNoElseMark -> "{;}\n"
  | SBreak -> "break;\n"
  | SIf(e, s1, SNoElseMark) ->  "if (" ^ string_of_exprs e ^ ")\n" ^ string_of_stmt s1
  | SIf(e, s1, s2) ->  "if (" ^ string_of_exprs e ^ ")\n" ^ string_of_stmt s1 ^ "else\n" ^ string_of_stmt s2
  | SFor(e1, e2, e3, stmt) ->"for (" ^string_of_exprs e1  ^ ";" ^string_of_exprs e2 ^ ";" ^ string_of_exprs e3 ^
")\n" ^
     string_of_stmt stmt ^ "\n"
  | SVarDecl(l)-> string_of_inits l^ ";\n"

(*translation of variable signature*)
let string_of_varsig =function
      VarSig (t, id) ->string_of_type t ^ " " ^ id

(*translation of function signature*)
let string_of_funcsig=function
      FuncSig(t, id) ->string_of_type_return t ^ " " ^ id

(*translation of initialzation*)
let string_of_global vdecl = string_of_init vdecl^ ";\n"

(*function translation, main function translated independently*)
let string_of_func_decl func_decl =
      if(string_of_funcsig func_decl.sfname="void main") then
            default_keyword ^ " " ^string_of_funcsig func_decl.sfname ^ "(" ^
            "String[] args"     ^ ")\n" ^
            string_of_stmt func_decl.sbody
      else
            default_keyword ^ " " ^string_of_funcsig func_decl.sfname ^ "(" ^
            (String.concat "," (List.map string_of_varsig func_decl.sformals))
            ^ ")\n" ^
            string_of_stmt func_decl.sbody

(*import list in Java*)
```

```
let string_of_imports import_list =
  String.concat ";\n" import_list ^ ";\n"

(*translation of constructs*)
let string_of_construct =function
       SGlobalVar(l)->"public static "^string_of_inits l ^ ";\n"
  |    SFuncDef (f)->string_of_func_decl f


(*translation of construct list and the generate whole Java code*)
let string_of_class x import_list programs =
       "\n" ^ string_of_imports import_list ^ "\n" ^
       "\npublic class " ^ x^ "\n{\n" ^String.concat "\n" (List.map string_of_construct programs) ^ "}\n"
```

---

# 8.11 TaML.ml

```
(* taml.ml -- Main runner file *)
(* Author: Adam Dossa *)


open Parser
open Scanner

type action =  Scanner | Ast | Sast | Compile

let _ =
       (* Read argument and assign to "action" *)
  let action = if Array.length Sys.argv > 1 then
    List.assoc Sys.argv.(1) [ ("-a", Ast);
                              ("-s", Scanner);
                              ("-j", Sast);
                              ("-c", Compile);
                            ]
       (* User "Scanner" by default if no argument is provided *)
  else Scanner in

  match action with
    Scanner -> let lexbuf = Lexing.from_channel stdin in
               let rec loop token  =
                  (match token with
                     | EOF -> []
                     | _ as t -> t::loop (Scanner.token lexbuf)) in
```

```
            let tokens = loop (Scanner.token lexbuf) in
            let output = Scanner_utils.string_of_tokens tokens in
            print_string output
    | Ast ->  let lexbuf = Lexing.from_channel stdin in
            let abstract_syntax_tree = Parser.program Scanner.token lexbuf in
            let output = Parser_utils.string_of_program abstract_syntax_tree in
            print_string output
    | Sast ->
            let lexbuf = Lexing.from_channel stdin in
            let abstract_syntax_tree = Parser.program Scanner.token lexbuf in
            let sastree = Translate.translate abstract_syntax_tree in
            let output = Translate_utils.string_of_sprogram sastree in
            print_string output
    | Compile ->
            let lexbuf = Lexing.from_channel stdin in
            let abstract_syntax_tree = Parser.program Scanner.token lexbuf in
            let sastree = Translate.translate abstract_syntax_tree in
            let import_list = ["package Java_lib"] in
            let output = Jpp.string_of_class (Sys.argv.(2)) import_list sastree in
            print_string output
```

---

# 8.12 Makefile (Master, in top directory)

```
# Authors: Adam Dossa and Maria Taku
all:
	cd TaML_src; make


TARFILES = Makefile testall.sh run.sh \
	TaML_src/scanner.mll TaML_src/scanner_utils.ml \
	TaML_src/parser.mly TaML_src/parser_utils.ml \
	TaML_src/ast.mli TaML_src/type.mli TaML_src/sast.mli \
	TaML_src/translate.ml TaML_src/translate_utils.ml \
	TaML_src/jpp.ml TaML_src/taml.ml \
	TaML_src/Makefile \
	test-input/* \
	test-output/* \
	java_lib/java_lib/Cell.java \
	java_lib/java_lib/Line.java \
	java_lib/java_lib/Table.java \
	java_lib/java_lib/Printers.java \
	GameOfLife.taml \
```

```
	BubbleSort.taml


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

.PHONY : test
test : taml testall.sh
	./testall.sh

.PHONY : clean

clean:
	cd TaML_src; make clean
	rm -f test_results.out
	rm -f taml
	rm -f TaML.tar.gz
```

---

# 8.13 Makefile (Inner, in TaML_src directory)

```
# Authors: Adam Dossa and Maria Taku
OBJS = parser.cmo scanner.cmo scanner_utils.cmo parser_utils.cmo translate.cmo translate_utils.cmo jpp.cmo taml.cmo

taml : $(OBJS)
	ocamlc -o ../taml $(OBJS)

scanner.ml : scanner.mll
	ocamllex scanner.mll

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

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

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

.PHONY : clean
clean :
	rm -f scanner.ml scanner.mli scanner_utils.mli \
```

```
      parser.ml parser.mli parser_utils.mli translate.mli \
      translate_utils.mli jpp.mli taml.mli \
      test_results.out \
      *.cmo *.cmi *.out *.diff *.annot

# Generated by ocamldep *.ml *.mli
jpp.cmo: type.cmi ast.cmi
jpp.cmx: type.cmi ast.cmi
parser.cmo: type.cmi ast.cmi parser.cmi
parser.cmx: type.cmi ast.cmi parser.cmi
parser_utils.cmo: type.cmi ast.cmi
parser_utils.cmx: type.cmi ast.cmi
scanner.cmo: parser.cmi
scanner.cmx: parser.cmx
scanner_utils.cmo: scanner.cmo parser.cmi
scanner_utils.cmx: scanner.cmx parser.cmx
semantic_check.cmo: type.cmi ast.cmi
semantic_check.cmx: type.cmi ast.cmi
taml.cmo: translate_utils.cmo translate.cmo scanner_utils.cmo scanner.cmo \
    parser_utils.cmo parser.cmi jpp.cmo
taml.cmx: translate_utils.cmx translate.cmx scanner_utils.cmx scanner.cmx \
    parser_utils.cmx parser.cmx jpp.cmx
translate.cmo: type.cmi sast.cmi ast.cmi
translate.cmx: type.cmi sast.cmi ast.cmi
translate_utils.cmo: type.cmi sast.cmi ast.cmi
translate_utils.cmx: type.cmi sast.cmi ast.cmi
ast.cmi: type.cmi
parser.cmi: ast.cmi
sast.cmi: type.cmi ast.cmi
type.cmi:
```

---

# 8.14 Java_Lib: Cell.Java

```java
// Author: Maria Taku
////////////////////////////////////////////
//                 Basic Cell Type          //
////////////////////////////////////////////

package Java_lib;

// E must be Integer or Float
public class Cell<E> {
```

```
	private E value;

	public Cell(){
		value = null;
	}

	public E getVal(){
		return value;
	}

	////////////////////////////////////////////////////////////////////////////
	// NOTE: if the cell is holding floats, all numbers must end with         //
	// an "f".  For example, myCell.setVal(3.5f)                              //
	// This is because Java automatically assumes decimal point numbers       //
	// are double unless explicitly case to float -- resulting in compile errors //
	////////////////////////////////////////////////////////////////////////////
	public void setVal(E newValue){
		value = newValue;
	}
	  public void setVal(Cell<E> c){
		value = c.getVal();
	  }

	public void print(){
		System.out.println(value);
	}
}
```

---

# 8.15 Java_Lib: Line.Java

```
// Author: Maria Taku
////////////////////////////////////////////////////////////////////////////
//     Line Type - 1d array composed of Cells,                            //
//  where the cells reference cells in a table                            //
// NOTE: if the LINE is holding floats, all numbers must end with         //
// an "f".  For example, mylINE.setCellValue(1, 3.5f)                     //
// This is because Java automatically assumes decimal point numbers       //
// are double unless explicitly case to float -- resulting in compile errors //
////////////////////////////////////////////////////////////////////////////
```

```java
package Java_lib;

//E must be Integer or Float
public class Line<E> {
	private Cell<E>[] line;
	private int lineLength;

	////////////////////////////////////////////////////////////////
	// Initialize to null.                                        //
	// If the line is declared and initialized to a table         //
	// in the same line (e.g., line l = myTable[@,3] ) just       //
	// call assignLine directly after                             //
	////////////////////////////////////////////////////////////////
	public Line(){
		line = null;
		lineLength=0;
	}

	////////////////////////////////////////////////////////////////
	// Create a line that isn't assigned to a table -- it's cells //
	// get their own spot in memory                               //
	////////////////////////////////////////////////////////////////
	public Line(int length){
		lineLength = length;
		line = new Cell[length];
		for (int i=0; i<length; i++){
				Cell<E> c = new Cell<E>();
				line[i] = c;
		}
	}

	//============================================================//
	//                              LINE ASSIGNMENT (TO TABLE)    //
	//    Assign the line to a contiguous row of cells in the table //
	//============================================================//

	////////////////////////////////////////////////////////////////
	// Index Syntax: [1~4, 5] or [3, 4~5]                         //
	////////////////////////////////////////////////////////////////
	public void assignLine(Table<E> t, int startRow, int endRow, int startCol, int endCol){
		//Error Checking
		if(endRow<startRow || endCol<startCol )
			throw new IllegalArgumentException("Error start row/column must be less than end row/column");
```

```
		// when we are creating a "row" line
		if(startRow == endRow ){
			lineLength = endCol - startCol + 1;
			line = new Cell[lineLength];
			for (int i=0; i<lineLength; i++)
				line[i] = t.getCell(startRow,startCol + i);
		}

		// when we are creating a "column" line
		else if (startCol == endCol){
			lineLength = endRow - startRow + 1;
			line = new Cell[lineLength];
			for (int i=0; i<lineLength; i++)
				line[i] = t.getCell(startRow + i, startCol);
		}

		else
			throw new IllegalArgumentException("Error: when assigning lines, either the startRow/endRow or
startCol/endCol MUST be equal");
	}

	////////////////////////////////////////////////////////////////
	// Index Syntax: [@, 5]                                       //
	////////////////////////////////////////////////////////////////
	public void assignLine(Table<E> t, String a1, String a2, int startCol, int endCol){
		//Error Checking
		if(!a1.equals("ALL") || !a2.equals("ALL"))
			throw new IllegalArgumentException("Error -- both rows are not @");
		if(endCol != startCol )
			throw new IllegalArgumentException("Error: end column and start column must be equal when row is
@");

		lineLength = t.numRows();
		line = new Cell[lineLength];
		for (int i=0; i<lineLength; i++)
			line[i] = t.getCell(i, startCol);

	}

	////////////////////////////////////////////////////////////////
	// Index Syntax: [5, @]                                       //
	////////////////////////////////////////////////////////////////
	public void assignLine(Table<E> t, int startRow, int endRow, String a1, String a2){
		//Error Checking
```

```
        if(!a1.equals("ALL") || !a2.equals("ALL"))
                throw new IllegalArgumentException("Error -- both columns are not @");
        if(endRow != startRow )
                throw new IllegalArgumentException("Error: end row and start row must be equal when column is @");

        lineLength = t.numColumns();
        line = new Cell[lineLength];
        for (int i=0; i<lineLength; i++)
                line[i] = t.getCell(startRow, i);

}


//=====================================================================//
//                         LINE COPIERS                                //
// Creates & returns a new line that is a COPY of cells in             //
// the old line. (e.g., table has new memory allocation)               //
// Note that the new Line will be of the same type (int or float)      //
// as the old line.                                                    //
//=====================================================================//

/////////////////////////////////////////////////////////////////////
// Example Usage: Line newL = oldL.createLineCopy(1, 5);          //
/////////////////////////////////////////////////////////////////////
public Line<E> createLineCopy(int startIndex, int endIndex ){
        //Error Checking
        if(endIndex < startIndex )
                throw new IllegalArgumentException("Error start index must be less than end index");

        // Initialize the new line
        int newLength = (endIndex - startIndex) + 1;
        Line<E> newLine = new Line<E>(newLength);

        // Copy cells into the new table
        copyCells(startIndex, endIndex, newLine);
        return newLine;
}

/////////////////////////////////////////////////////////////////////
// Example Usage: Line newL = oldL.createLineCopy("ALL", "ALL");  //
/////////////////////////////////////////////////////////////////////
public Line<E> createLineCopy(String a1, String a2 ){
        //Error Checking
        if(!a1.equals("ALL") || !a2.equals("ALL"))
```

```
            throw new IllegalArgumentException("Error -- both columns are not @");

        // Initialize the new line
        int newLength = lineLength;
        Line<E> newLine = new Line<E>(newLength);

        // Copy cells into the new table
        copyCells(0, newLength-1, newLine);
        return newLine;
}

// HELPER FUNCTION: for copying cells into new table
private void copyCells(int startIndex, int endIndex, Line<E> newLine) {
        int a=0;                        // newTable's index

        for (int i=startIndex; i<=endIndex; i++){
            Cell<E> c = new Cell<E>();
            c.setVal(line[i].getVal());
            newLine.setCell(a,c);
            a++;
        }
}

//=========================================================//
//                        GETTERS & SETTERS                //
//=========================================================//
public E getCellValue(int index){
        return line[index].getVal();
}

// Set single cell to a value
public void setVal(int index, E newValue){
        line[index].setVal(newValue);
}

// Sets entire line to a single Value
public void setVal(E newValue){
        for(int i=0; i<lineLength; i++)
            line[i].setVal(newValue);
}

// Sets subrange of line to a single Value
public void setVal(int startIndex, int endIndex, E newValue){
        if(endIndex < startIndex )
```

```
            throw new IllegalArgumentException("Error start index must be less than end index");

      for(int i=startIndex; i<=endIndex; i++)
            line[i].setVal(newValue);
}

////////////////////////////////////////////////////////////////
// Useful if we want to set a new cell etc. to a cell         //
// in this line (and it's referenced table cell)              //
// e.g., "cell c = myLine[3]" could leverage this function    //
////////////////////////////////////////////////////////////////
public Cell<E> getCell(int index){
      return line[index];
}

private void setCell(int index, Cell<E> c){
      line[index] = c;

}

//=========================================================//
//                      PRINT FUNCTION                     //
//=========================================================//
public void print(){
      StringBuffer[] rowText = new StringBuffer[4];       // text of each Row
      int colLength;                                      // max length of a particular column
      String bars = "=============================================";

      // Setup Row Values & initialize StringBuffers
      rowText[0] = new StringBuffer("" + '\t');
      rowText[1] = new StringBuffer("" + '\t');
      rowText[2] = new StringBuffer("1" + '\t' + "| ");
      rowText[3] = new StringBuffer("" + '\t');

      // Save Data for each line column into rowText[2]
      for (int i=0; i<lineLength; i++){

            // Find Max Length in a certain column
            colLength = String.valueOf(line[i].getVal()).length();

            // Save Data of the line, setting length of column to fit maxLength
            String curNum = String.valueOf(line[i].getVal());
            String paddedNum = String.format("%1$-" + colLength + "s", curNum);
            rowText[2].append(paddedNum + " | ");
```

```
                        // Save Data for Column Labels
                        String curLabel;
                        if(i<26)
                                curLabel = String.valueOf((char)(i + 65));
                        else
                                curLabel = String.valueOf((char)(i + 39))+String.valueOf((char)(i + 39));
                        String paddedLabel = String.format("%1$-" + colLength + "s", curLabel);
                        rowText[0].append("   " + paddedLabel + " ");

                        // Save Data for Bottom & Top Bars
                        String bottomBar = bars.substring(0, colLength+3);
                        rowText[1].append(bottomBar);
                        rowText[3].append(bottomBar);
                }

                // Print rows
                for (int i=0; i<=3; i++){
                        System.out.println(rowText[i]);
                }
        }
}
```

---

# 8.16 Java_Lib: Table.Java

```
// Author: Maria Taku
////////////////////////////////////////////////////////////////////////////
//     Table Type - 2d array composed of Cells                            //
// NOTE: if the table is holding floats, all numbers must end with        //
// an "f".  For example, myTable.setCellValue(1, 2, 3.5f)                 //
// This is because Java automatically assumes decimal point numbers       //
// are double unless explicitly case to float -- resulting in compile errors  //
////////////////////////////////////////////////////////////////////////////


package Java_lib;

// E must be Integer or Float
public class Table<E> {
        private Cell<E>[][] table;
        private int numRows;
```

```
private int numColumns;

////////////////////////////////////////////////////////////////////////////
//     Example TaML to Java conversions:                                  //
//     table t1 = ([3,4], int); --> Table<Integer> t1 = new Table<Integer>(3,4); //
//     table t2 = ([3,4], float); --> Table<Float> t2 = new Table<Float>(3,4);   //
////////////////////////////////////////////////////////////////////////////
public Table(int numR, int numC){
	numRows = numR;
	numColumns = numC;

	// Error Checking
	if(numRows>500 || numColumns>50)
		throw new IllegalArgumentException("Error: There can't be more than 500 rows or 50 columns");
	if(numRows<1 || numColumns<1)
		throw new IllegalArgumentException("Error: You must have at least 1 column and/or row");

	// Fill the table with Cell objects
	table = new Cell[numRows][numColumns];
	for (int i=0; i<numRows; i++){
		for(int j=0; j<numColumns; j++){
			Cell<E> c = new Cell<E>();
			table[i][j] = c;
		}
	}
}


//=========================================================================//
//                         TABLE COPIERS                                   //
// Creates & returns a new table that is a COPY of cells in                //
// the old table. (e.g., table has new memory allocation)                  //
// Note that the new Table will be of the same type (int or float)         //
// as the old table.                                                       //
//=========================================================================//

////////////////////////////////////////////////////////////////////////
// Example Usage: Table newT = oldT.createTableCopy(1, 5, 1, 4);    //
////////////////////////////////////////////////////////////////////////
public Table<E> createTableCopy(int startRow, int endRow, int startCol, int endCol){
	//Error Checking
	if(endRow<startRow || endCol<startCol )
		throw new IllegalArgumentException("Error start row/column must be less than end row/column");
```

```
        // Initialize the new table
        int newNumRows = (endRow - startRow) + 1;
        int newNumCols = (endCol - startCol) + 1;
        Table<E> newTable = new Table<E>(newNumRows,newNumCols);

        // Copy cells into the new table
        copyCells(startRow, endRow, startCol, endCol, newTable);
        return newTable;
    }

    ////////////////////////////////////////////////////////////////////
    // Example Usage: Table newT = oldT.createTableCopy(ALL, ALL, 1, 4);//
    ////////////////////////////////////////////////////////////////////
    public Table<E> createTableCopy(String a1, String a2, int startCol, int endCol){
        //Error Checking
        if(endCol < startCol)
            throw new IllegalArgumentException("Error start column must be less than end column");
        if(!a1.equals("ALL") || !a2.equals("ALL"))
            throw new IllegalArgumentException("Error -- both rows are not @");

        // Initialize the new table
        int newNumRows = numRows;
        int newNumCols = (endCol - startCol) + 1;
        Table<E> newTable = new Table<E>(newNumRows,newNumCols);

        // Copy cells into the new table
        copyCells(0, newNumRows-1, startCol, endCol, newTable);
        return newTable;
    }

    ////////////////////////////////////////////////////////////////////
    // Example Usage: Table newT = oldT.createTableCopy(1, 4, ALL, ALL) //
    ////////////////////////////////////////////////////////////////////
    public Table<E> createTableCopy(int startRow, int endRow, String a1, String a2){
        //Error Checking
        if(endRow < startRow)
            throw new IllegalArgumentException("Error start row must be less than end row");
        if(!a1.equals("ALL") || !a2.equals("ALL"))
            throw new IllegalArgumentException("Error -- both columns are not @");

        // Initialize the new table
        int newNumRows = (endRow - startRow) + 1;
        int newNumCols = numColumns;
        Table<E> newTable = new Table<E>(newNumRows,newNumCols);
```

```
	// Copy cells into the new table
	copyCells(startRow, endRow, 0, newNumCols-1, newTable);
	return newTable;
}

////////////////////////////////////////////////////////////////////
// Example: Table newT = oldT.createTableCopy(ALL, ALL, ALL, ALL)  //
////////////////////////////////////////////////////////////////////
public Table<E> createTableCopy(String a1, String a2, String a3, String a4){
	//Error Checking
	if(!a1.equals("ALL")||!a2.equals("ALL")||!a3.equals("ALL")||!a4.equals("ALL"))
		throw new IllegalArgumentException("Error -- both rows/columns are not @");

	// Initialize the new table
	Table<E> newTable = new Table<E>(numRows,numColumns);

	// Copy cells into the new table
	copyCells(0, numRows-1, 0, numColumns-1, newTable);
	return newTable;
}


// HELPER FUNCTION: for copying cells into new table
private void copyCells(int startRow, int endRow, int startCol, int endCol,Table<E> newTable) {
	int a=0;                        // newTable's row index
	int b=0;                        // newTable's column index

	for (int i=startRow; i<=endRow; i++){
		b = 0;
		for(int j=startCol; j<=endCol; j++){
			Cell<E> c = new Cell<E>();
			c.setVal(table[i][j].getVal());
			newTable.setCell(a, b, c);
			b++;
		}
		a++;
	}
}

//=====================================================================//
//                        GETTERS & SETTERS                            //
//=====================================================================//
public E getCellValue(int row, int column){
```

```
        return table[row][column].getVal();
}

// Set single cell in table to a certain value
public void setVal(int row, int column, E newValue){
        table[row][column].setVal(newValue);
}

// Sets entire table to a single value
public void setVal( E newValue){
        for (int i=0; i<numRows; i++){
                for(int j=0; j<numColumns; j++)
                        table[i][j].setVal(newValue);
        }
}

// Sets subrange of table to a single value. Index ranges ex: [1~2, 3~5]
public void setVal(int startRow, int endRow, int startCol, int endCol, E newValue) {
        if(endRow<startRow || endCol<startCol )
                throw new IllegalArgumentException("Error start row/column must be less than end row/column");

        for (int i=startRow; i<=endRow; i++){
                for(int j=startCol; j<=endCol; j++)
                        table[i][j].setVal(newValue);
        }
}

// Sets subrange of table to a single value. Index ranges ex: [@, 3~5]
public void setVal(String a1, String a2, int startCol, int endCol, E newValue) {
        if(endCol < startCol)
                throw new IllegalArgumentException("Error start column must be less than end column");
        if(!a1.equals("ALL") || !a2.equals("ALL"))
                throw new IllegalArgumentException("Error -- both rows are not @");

        for (int i=0; i<numRows; i++){
                for(int j=startCol; j<=endCol; j++)
                        table[i][j].setVal(newValue);
        }
}

// Sets subrange of table to a single value. Index ranges ex: [3~5, @]
public void setVal(int startRow, int endRow, String a1, String a2, E newValue) {
        if(endRow < startRow)
                throw new IllegalArgumentException("Error start row must be less than end row");
```

```
        if(!a1.equals("ALL") || !a2.equals("ALL"))
                throw new IllegalArgumentException("Error -- both rows are not @");

        for (int i=startRow; i<=endRow; i++){
                for(int j=0; j<numColumns; j++)
                        table[i][j].setVal(newValue);
        }
}


////////////////////////////////////////////////////////////
// Useful if we want to set a cell/line segment to a cell  //
// in the table.  e.g., "cell c = myTable[3,2]" could      //
// leverage this function                                  //
////////////////////////////////////////////////////////////
public Cell<E> getCell(int row, int column){
        return table[row][column];
}

// Private since this is only used internally when copying
// cells into a new table.  Otherwise, users really shouldn't
// be able to copy cells into tables -- they should only be
// allowed to edit the values held by those cells (via
// setCellValue etc. )
private void setCell(int row, int column, Cell<E> c){
        table[row][column] = c;
}

public int numRows(){
        return numRows;
}

public int numColumns(){
        return numColumns;
}

//=========================================================================//
//                        PRINT FUNCTION                                   //
//=========================================================================//
public void print(){
        StringBuffer[] rowText = new StringBuffer[numRows+3];      // text of each Row
        int maxLength;                                             // max length of a particular column
        String bars = "================================================";
```

```
// Setup Row Values & initialize StringBuffers
rowText[0] = new StringBuffer("" + '\t');
rowText[1] = new StringBuffer("" + '\t');
for (int i=2; i<=numRows+1; i++)
	rowText[i] = new StringBuffer(String.valueOf(i-1) + '\t' + "| ");
rowText[numRows+2] = new StringBuffer("" + '\t');

// Save Data for each column into rowText[]
for (int j=0; j<numColumns; j++){

	// Find Max Length in a certain column
	maxLength = 0;
	for (int i=0; i<numRows; i++){
		int curLength = String.valueOf(table[i][j].getVal()).length();
		if(curLength > maxLength)
			maxLength = curLength;
	}

	// Save Data in each row, setting length of column to fit maxLength
	for (int i=0; i<numRows; i++){
		String curNum = String.valueOf(table[i][j].getVal());
		String paddedNum = String.format("%1$-" + maxLength + "s", curNum);
		rowText[i+2].append(paddedNum + " | ");
	}

	// Save Data for Column Labels
	String curLabel;
	if(j<26)
		curLabel = String.valueOf((char)(j + 65));
	else
		curLabel = String.valueOf((char)(j + 39))+String.valueOf((char)(j + 39));
	String paddedLabel = String.format("%1$-" + maxLength + "s", curLabel);
	rowText[0].append("  " + paddedLabel + " ");

	// Save Data for Bottom & Top Bars
	String bottomBar = bars.substring(0, maxLength+3);
	rowText[1].append(bottomBar);
	rowText[numRows+2].append(bottomBar);
}

// Print rows
for (int i=0; i<=numRows+2; i++){
	System.out.println(rowText[i]);
}
```

```
	}
}
```

---

# 8.17 Java_Lib: Printers.Java

```
// Author: Maria Taku
package Java_lib;

public class Printers {

	public static void print(Table t){
		if (t == null)
			System.out.println(t);
		else
			t.print();
	}

	public static void print(Line l){
		if (l == null)
			System.out.println(l);
		else
			l.print();
	}

	public static void print(Cell c){
		if (c == null)
			System.out.println(c);
		else
			c.print();
	}

	public static void print(Integer i){
		System.out.println(i);
	}

	public static void print(Float f){
		System.out.println(f);
	}

	public static void print(Double d){
		System.out.println(d);
	}
```

```
	public static void print(Character c){
		System.out.println(c);
	}


	public static void print(String s){
		System.out.println(s);
	}

	public static void print(Boolean b){
		System.out.println(b);
	}
}
```

---

# 8.18 Run.sh (Main script for running .taml files)

#Author: Adam Dossa

```
# Based on your shell, may need to edit this path to Bourne shell
#!/usr/bin/sh

#dos2unix testall.sh                                    # For windows users

echo "Compiling: "$1
export CLASS_NAME=`echo $1 | sed 's/\.taml//'`
export JAVA_NAME=`echo $1 | sed 's/\.taml/\.Java/'`
./taml -c $CLASS_NAME < $1 > Java_lib/Java_lib/$JAVA_NAME
cd Java_lib/
Javac Java_lib/*.Java
Java Java_lib/$CLASS_NAME
```

---

# 8.19 Testall.sh (Runs the Test Suite)

```
# Author: Maria Taku.  Edited by Le Chang & Quizi Shangguan
# to compile and run .Java code.
#####################################################################
# testall.sh                                                        #
# This script will run all test files in "test_input"               #
# and compare their outputs to the expected outputs in              #
```

```
# "test_output."                                                    #
#                                                                   #
# Any differences between input/expected output will be             #
# saved in test_results.out                                         #
#####################################################################


#####################################################################
#                              Variables and Config                 #
#####################################################################

# Based on your shell, may need to edit this path to Bourne shell
#!/usr/bin/sh

#dos2unix testall.sh                                  # For windows users

# Start with a clean output file
RESULTS=test_results.out
rm -f ${RESULTS}

# Different File Types
TAML=test*.taml
SCANNER=test*.scanner
PARSER=test*.parser
SAST=test*.sast
JAVA=test*.Java
CLASS=test*.class
OUT=test*.out

# Our TaML Main Program
PROGRAM=./taml

NUM_TESTS=$(echo `ls test-output | wc -l`)
NUM_ERROR=0



#####################################################################
#                                    Main Methods                   #
#####################################################################


### Runs all tests (.taml files) and saves outputs to each       ###
### stage with their appropriate extensions   in temp location   ###
```

```
runTestFiles() {
	for FILE in test-input/${TAML}
	do
		SCANNER_TEMP_FILE=$(echo ${FILE} | sed 's/taml/scanner/')
		${PROGRAM} -s < ${FILE} > ${SCANNER_TEMP_FILE} || {
			echo "${FILE} failed to execute during scanning"
		}

		PARSER_TEMP_FILE=$(echo ${FILE} | sed 's/taml/parser/')
		${PROGRAM} -a < ${FILE} > ${PARSER_TEMP_FILE} || {
			echo "${FILE} failed to execute during parsing"
		}

		SAST_TEMP_FILE=$(echo ${FILE} | sed 's/taml/sast/')
		${PROGRAM} -j < ${FILE} > ${SAST_TEMP_FILE} || {
			echo "${FILE} failed to execute during parsing"
		}

	  JPP_TEMP_FILE=$(echo ${FILE} | sed 's/taml/Java/')
	  JPP_CLASS_NAME=$(echo ${FILE} | sed 's/\.taml// ' )
	  JPP_CLASS_NAME=$(echo ${JPP_CLASS_NAME} | sed 's/test-input\/// ' )
	  ${PROGRAM} -c ${JPP_CLASS_NAME} < ${FILE} > ${JPP_TEMP_FILE} || {
		echo "${FILE} failed to execute during generating Java file"
	  }

	done
}

moveJavafile(){
    for FILE in test-input/${JAVA}
    do
        cp $FILE Java_lib/Java_lib/
    done

}

compileJavaFile(){
	for FILE in Java_lib/Java_lib/${JAVA}
	do
		JAVA_FILE=$(echo ${FILE} | sed 's/Java_lib\/Java_lib\//Java_lib\//')
		cd Java_lib
		Javac  ${JAVA_FILE}
		cd ..
	done
```

```
}

runJavaClass() {
    for FILE in Java_lib/Java_lib/${CLASS}
    do
            CLASS_FILE=$(echo ${FILE} | sed 's/Java_lib\/Java_lib\//Java_lib\//')
            OUT_FILE=$(echo ${CLASS_FILE} | sed 's/\.class/\.out/')
            CLASS_FILE=$(echo ${CLASS_FILE} | sed 's/\.class//')
            cd Java_lib
            Java  ${CLASS_FILE} >> ${OUT_FILE}
            cp ${OUT_FILE} ../test-input/
            cd ..
      done
}


######################################################################
#                                  Comparing Outputs                 #
######################################################################

### Given an input file, compares it to the respective            ###
### output "golden" file, and outputs differences to RESULTS      ###
diffFiles() {
      IN_FILE=$1
      OUT_FILE=$(echo $1 | sed 's/input/output/')
      diff -b -q ${IN_FILE} ${OUT_FILE} || {
            echo "FAILED: Output Mismatch in ${IN_FILE}" >> ${RESULTS}
            diff -b ${IN_FILE} ${OUT_FILE} >> ${RESULTS}
            echo "" >> ${RESULTS}
            NUM_ERROR=$(( NUM_ERROR + 1 ))
      }
}


### Test all Scanner output files and outputs results to log  ###
scannerResults(){
      echo "Comparing scanner files..."
      echo "Results of the Scanner test" >> $RESULTS
      for FILE in test-input/${SCANNER}
      do
            diffFiles ${FILE}
      done
      echo "************************************" >> ${RESULTS}
}
```

```
### Test all Parser output files and outputs results to log   ###
parserResults(){
	echo "Comparing parser files..."
	echo "Results of the Parser test" >> ${RESULTS}
	for FILE in test-input/${PARSER}
	do
		diffFiles ${FILE}
	done
	echo "************************************" >> ${RESULTS}
}


### Test all .Java output files and outputs results to log   ###
sastResults(){
	echo "Comparing sast files..."
	echo "Results of the sast test" >> ${RESULTS}
	for FILE in test-input/${SAST}
	do
		diffFiles ${FILE}
	done
	echo "************************************" >> ${RESULTS}
}


JavaResults(){
	echo "Comparing Java files..."
	echo "Results of the Java test" >> ${RESULTS}
	for FILE in test-input/${JAVA}
	do
		diffFiles ${FILE}
	done
	echo "************************************" >> ${RESULTS}
}


### Test all final output files (.out) and outputs results to log   ###
finalResults(){
	echo "Comparing out files..."
	echo "Results of the final output test" >> ${RESULTS}
	for FILE in test-input/${OUT}
	do
		diffFiles ${FILE}
```

```
	done
	echo "************************************" >> ${RESULTS}
}



######################################################################
#                                  Run the Program                   #
######################################################################

echo "See test_results.out for detailed results"

runTestFiles
scannerResults
parserResults
sastResults
JavaResults

moveJavafile
compileJavaFile
runJavaClass

finalResults

rm test-input/${SCANNER}
rm test-input/${PARSER}
rm test-input/${SAST}
rm test-input/${JAVA}
rm test-input/${OUT}

rm Java_lib/Java_lib/${JAVA}
rm Java_lib/Java_lib/${CLASS}
rm Java_lib/Java_lib/${OUT}


echo "There were ${NUM_ERROR} out of ${NUM_TESTS} test cases which failed"
echo "************************************" >> ${RESULTS}
echo "There were ${NUM_ERROR} out of ${NUM_TESTS} test cases which failed" >> ${RESULTS}
```