TaML

Language Reference Manual

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

Columbia University

29h October 2012

Table of Contents

1. Introduction 4
2. Lexical Conventions 4
    2.1 Character Set 4
    2.2 White Space 4
    2.3 Comments 4
    2.4 Token Types 4
    2.5 Identifiers 5
    2.6 Keywords 5
    2.7 Constants 5
        2.7.1 Integer Constant 5
        2.7.2 Float Constant 5
        2.7.3 Character Constant 6
        2.7.4 String Constant 6
        2.7.5 Boolean Constant 6
3. Types 6
    3.1 General Types 6
        3.1.1 Integers 6
        3.1.2 Floating Point Numbers 6
        3.1.3 Characters 7
        3.1.4 Strings 7
        3.1.5 Booleans 7
    3.2 Table Types 7
        3.2.1 Table 8
        3.2.2 Line 9
        3.2.3 Cell 10
4. Expressions 11
    4.1 Primary Expressions 11
        4.1.1 Identifiers 11
        4.1.2 Constants 11
        4.1.3 Parenthesized Expressions 11
    4.2 Assignment Operator 11
    4.3 Arithmetic Operators 11

4.4 Comparative operators 12
4.5 Logical operators 12
4.6 Table operators 12
4.6.1 Table access operator ‘[]’ 12
4.6.2 Line access operator ‘[]’ 13
4.6.3 The tilde (~) operator 13
4.6.4 The at (@) symbol 13
4.7 Precedence Rules 13
5. Statements 14
5.1 Variable declarations and initialization statements 14
5.1.1 Type Specifiers 14
5.1.2 Variable Declarations 14
5.1.3 Variable Initialization 15
5.1.4 Initialization and Declaration of Table Types 15
5.2 Expression statements 15
5.3 Selection statements 15
5.4 Block statements 16
5.5 Iterative statements 16
5.6 Return Statements 16
6. Functions 16
6.1 Function Declarations 17
6.2 The Main Function 17
6.3 Built-In Functions 18
7. Scope 18
8. Language Formalizations / Grammar 18
8.1 Scanner.mll 19
8.2 Parser.mly 20

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. Lexical Conventions

2.1 Character Set

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

2.2 White Space

As in the C language, white space (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.

2.3 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.

2.4 Token Types

Token types in TaML can be one of the following:

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

2.5 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.

2.6 Keywords

Table 2.1 lists the set of identifiers, that are reserved for keywords in TaML. These identifiers cannot be used for any other purpose.

bool	false	if	print	true
cell	for	int	return	then
char	float	line	string	
else	func	NULL	table	

Table 2.1: Keywords in TaML

2.7 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 or float) associated with it.

2.7.1 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.

2.7.2 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.

2.7.3 Character Constant

A character constant is a 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, or one of the digits from 0 to 9, or one of ‘_’, ‘\r’, ‘\n’. Every character constant is considered to be of type char.

2.7.4 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, and any of ‘_’, ‘\r’, ‘\n’. Every string constant is considered to be of type string.

2.7.5 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. Types

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

3.1 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 3.1.

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 3.1: General Data Types Information

3.1.1 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.

3.1.2 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.

3.1.3 Characters

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

Characters consist of single characters as defined above. In 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.

3.1.4 Strings

Strings consist of one or more characters as defined above. 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.

3.1.5 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, characters 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 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 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 3.1. 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**
Table	*table t = ([3,4], int);* creates a table type called "t" with 3 rows and 4 columns	NULL

Line	*line r = t[3,@];* Creates a line called "r" that holds all cells in row 3 of table "t" (e.g., r represents a row) *line c = t[@,2];* Creates a line called "c" that holds all cells in column 2 of table "t" (e.g., c represents a column) *line sc = t[2~5, 2];* Creates a line called sc that holds all cells in column 2 from rows 2 to 5 inclusive. Note: The @ and ~ operators are discussed in more detail in section 3.2.2	NULL
Cell	*cell c = t[1,2];* Creates a cell called "c" that corresponds to row 1, column 2 of table "t"	NULL

Table 3.2: Table Types - default values and example initialization Syntax

3.2.1 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.

3.2.2 Line

A line in TaML consists of either an entire line of cells in a table or a subrange of contiguous 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,@];
```

3.2.3 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. As mentioned above, all cells in a table 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];
```

The type cell has different behaviour depending on the context in which it is referenced.

If it is referenced on the left hand side of an assignment, the behaviour is to assign the right hand side value of the assignment to the coordinate in the table referenced by the cell.

For example the value at row 3, column 4 in myTable can be changed by declaring, for example:

```
c = 2000;                # Changes the value of row 3, column 4 of myTable to 2000
```

If it is referenced outside of an assignment then a cell will resolve to being the value at the coordinate in the table referenced by the table.

For example if we wish to compare whether the value in coordinate row 3, column 4 with row 4, column 3 we could do:

```
cell a = myTable[3,4];
cell b = myTable[4,3];
if (a > b) then {...}
```

4. Expressions

4.1 Primary Expressions

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

4.1.1 Identifiers

Identifiers represent variables or functions.

4.1.2 Constants

Constants can be divided into five categories: integer, float, character, string and boolean. The specific definitions for each can be seen in Section 2.7.

4.1.3 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.

4.2 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 3.2.3.

4.3 Arithmetic Operators

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

```
+  ->  addition
-  ->  subtraction
*  ->  multiplication
/  ->  division
```

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.

4.4 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.

4.5 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.

4.6 Table operators

4.6.1 Table access operator ‘[]’

```
e.g.  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
```

table1[m,n]

1) Note that m and n must be 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.

2) If m or n is in the format of “integer~integer”, and the other is a single integer, then table1[2~3,5] returns a line.

```
line line1=  table1[2~3,5];
```

If m and n both are in the format of “integer~integer”. table1[2~3,5~6] returns a table.

table table2= table1[2~3,5];

4.6.2 Line access operator ‘[]’

e.g. *int a= line1[2];*
line1[2]=1000;

line1[m]

1) Note that m must be an expression that evaluates to a positive integer, greater or equal to. In this case, line1[m] returns the m-th value of the line.

int a = line1[1];

2) If m is in the format of “integer~integer”, line1[m] returns a line.

line line2 = line1[2~3];

4.6.3 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]

4.6.4 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.

4.7 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	()
3	*
4	+ -
5	> < <= >= != ==
6	!
7	&&
8	||
9	=

5. Statements

5.1 Variable declarations and initialization statements

5.1.1 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 in the table
line: a horizontal or vertical array of cells
table: a table with cells in it

5.1.2 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;

5.1.3 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);

5.1.4 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 and is discussed in detail in Section 3.2.

5.2 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.

5.3 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).

5.4 Block statements

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

```
{
        declaration_list
        statement _list
}

declaration_list -> declaration_list   | declaration
statement_list -> statement_list  |  statement
```

Note that each statement and declaration in the *statement_list* and *declaration_list*, respectively, must end in a semicolon.

5.5 Iterative statements

In TaML,we only have one form of iteration statement which takes the form of a *for* loop:

```
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).

5.6 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;
```

6. Functions

6.1 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 void, int, float, bool, line, cell or table 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 5.4. 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;
}
```

6.2 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 return_type main(string[] args){}

The *return_type* of the main function must be of type int.

6.3 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 associate 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  | …...
```

7. 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.

8. Language Formalizations / Grammar

The grammar of TaML is described in the scanner.mll and parser.mly files below

8.1 Scanner.mll

```
{ open Parser }

let digit = ['0' - '9']
let ip = digit+
let fp = digit+
let ie = digit+
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 *)
| '('      { LPAREN }
| ')'      { RPAREN }
| ']'      { LSQPAREN }
| '['      { RSQPAREN }
| '{'      { LBRACE }
| '}'      { RBRACE }
| '@'      { WILDCARD }
| '~'      { RANGE }
| ';'      { SEMI }
| ','      { COMMA }
| '+'      { PLUS }
| '-'      { MINUS }
| '*'      { TIMES }
| '/'      { DIVIDE }
| '='      { ASSIGN }
| "=="     { EQ }
| "!="     { NEQ }
| '<'      { LT }
| "<="     { LEQ }
| ">"      { GT }
| ">="     { GEQ }
| "&&"     { AND }
| "||"     { OR }
| "!"      { NOT }
| "if"     { IF }
| "else"   { ELSE }
| "for"    { FOR }
| "return" { RETURN }
| "NULL"   { NULL }
| "cell"   { CELL }
| "line"   { LINE }
| "table"  { TABLE }
| "func"   { FUNC }
| "int"    { INT }
| "float"  { FLOAT }
| "bool"   { BOOL }
| "string" { STRING }
| "char"   { CHAR }
| ['-']?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) }
| "true" | "false" as lit { BOOLLITERAL(bool_of_string lit) }
| ['\"']['a'-'z' 'A'-'Z' '0'-'9' '_' '\r' '\n']*['\"'] as lit { STRINGLITERAL(lit) }
| ['\'']['a'-'z' 'A'-'Z' '0'-'9' '_' '\r' '\n']['\''] as lit { CHARLITERAL(lit) }
| ['a'-'z' 'A'-'Z']['a'-'z' 'A'-'Z' '0'-'9' '_']* as ident { ID(ident) }
| eof { EOF }
| _ 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 Parser.mly

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

%nonassoc NOELSE
%nonassoc ELSE

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

%start program
%type <int> program

%%

program:
   /* nothing */ { 0 }
 | program fdecl { 0 }
 | program vdecl { 0 }

/* e.g. func int some_func (int a, int b, float c) { Table foo; do_something; } */
fdecl:
   FUNC ptype ID LPAREN formals_opt RPAREN LBRACE vdecl_list stmt_list RBRACE
     { 0 }

/* List of all types */
ptype:
    INT             { 0 }
  | FLOAT           { 0 }
  | BOOL            { 0 }
  | CHAR            { 0 }
  | STRING          { 0 }
  | TABLE           { 0 }
  | CELL            { 0 }
  | LINE            { 0 }

/* Types that can be used in tables */
ttype:
    FLOAT     { 0 }
  | INT       { 0 }

/* Parameters of a function */
formals_opt:
    /* nothing */ { 0 }
  | formal_list   { 0 }

formal_list:
    ptype ID                     { 0 }
  | formal_list COMMA ptype ID   { 0 }

/* A table selection that returns a table e.g. [1~5, @] */
table_sel:
    LSQPAREN range_op COMMA range_op RSQPAREN { 0 }

/* a table selection that returns a line e.g. [4, @] */
line_sel:
    LSQPAREN INTLITERAL COMMA range_op RSQPAREN { 0 }
  | LSQPAREN range_op COMMA INTLITERAL RSQPAREN { 0 }

/* A table selection that returns a cell e.g. [4, 5] */
/* NB - also used in table initialization to set table size */
cell_sel:
    LSQPAREN INTLITERAL COMMA INTLITERAL RSQPAREN { 0 }

/* A line selection that returns a line -- selects a contiguous subset of a line */
line_of_line:
    LSQPAREN range_op RSQPAREN { 0 }

/* A line selection that returns a scalar -- a single cell value*/
scalar_of_line:
    LSQPAREN INTLITERAL RSQPAREN { 0 }

/* A range of cells can either be selected with the wildcard -- @ -- or *
/* by specifing the range with, for example, 4~6 */
range_op:
    WILDCARD { 0 }
  | INTLITERAL RANGE INTLITERAL { 0 }

/* List of Variables declared -- used in fdecl*/
vdecl_list:
    /* nothing */    { 0 }
  | vdecl_list vdecl { 0 }
```

```
/* we allow variable initialization to default (null or false) or specific values */
vdecl:
    vdecl1   {0}       /* default initialization*/
  | vdecl2   {0}       /* Explicit initialization value*/

/* 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" */
vdecl1:
    INT ID  SEMI    { 0 }
  | FLOAT ID  SEMI  { 0 }
  | STRING ID  SEMI { 0 }
  | BOOL ID  SEMI   { 0 }
  | CHAR ID  SEMI   { 0 }
  | TABLE ID  SEMI  { 0 }
  | LINE ID  SEMI   { 0 }
  | CELL ID  SEMI   { 0 }

/* Explicit initialization value */
vdecl2:
    INT ID ASSIGN INTLITERAL SEMI                { 0 }        /* int a = 4; */
  | INT ID ASSIGN NULL SEMI                      { 0 }        /* int a = NULL; */
  | INT ID ASSIGN ID cell_sel SEMI               { 0 }        /* int a = table1[3,4]; */
  | INT ID ASSIGN ID scalar_of_line SEMI         { 0 }        /* int a = line1[4]; */
  | FLOAT ID ASSIGN FLOATLITERAL SEMI            { 0 }        /* float b = 4.0; */
  | FLOAT ID ASSIGN NULL SEMI                    { 0 }        /* float b = NULL; */
  | FLOAT ID ASSIGN ID cell_sel SEMI             { 0 }        /* float b = table1[3,4]; */
  | FLOAT ID ASSIGN ID scalar_of_line SEMI{ 0 }               /* float b = line1[4]; */
  | STRING ID ASSIGN STRINGLITERAL SEMI          { 0 }        /* string s = "cat"; */
  | STRING ID ASSIGN NULL SEMI                   { 0 }        /* string s = NULL; */
  | CHAR ID ASSIGN CHARLITERAL SEMI              { 0 }        /* char c = 's'; */
  | CHAR ID ASSIGN NULL SEMI                     { 0 }        /* char c = NULL; */
  | BOOL ID ASSIGN BOOLLITERAL SEMI              { 0 }
  | TABLE ID ASSIGN LPAREN cell_sel COMMA ttype RPAREN SEMI { 0 }    /* table t = ([3,4], int); */
  | TABLE ID ASSIGN ID table_sel SEMI            { 0 }        /* table t = table1[1~2,3~4] */
  | TABLE ID ASSIGN NULL SEMI                    { 0 }        /* table t = NULL; */
  | LINE ID ASSIGN ID line_sel SEMI     { 0 } /* line l = table1[@,2];  line l = line1[1~3,2]; */
  | LINE ID ASSIGN ID line_of_line SEMI          { 0 }        /* line l = line1[1~4]; */
  | LINE ID ASSIGN  NULL SEMI                    { 0 }        /* line l = NULL; */
  | CELL ID ASSIGN ID cell_sel SEMI              { 0 }        /* cell c = table1[5,2]; */
  | CELL ID ASSIGN NULL SEMI                     { 0 }        /* cell c = NULL; */


/* List of Statements declared -- used in fdecl*/
stmt_list:
    /* nothing */  { 0 }
  | stmt_list stmt { 0 }

block_stmt:
    LBRACE stmt_list RBRACE { 0 }

stmt:
    expr SEMI { 0 }
  | RETURN expr SEMI { 0 }
```

```
   | IF LPAREN expr RPAREN block_stmt %prec NOELSE { 0 }
   | IF LPAREN expr RPAREN block_stmt ELSE block_stmt   { 0 }
   | FOR LPAREN expr SEMI expr SEMI expr RPAREN block_stmt { 0 }
   | WHILE LPAREN expr RPAREN block_stmt { 0 }

expr:
     INTLITERAL              { 0 }
   | FLOATLITERAL            { 0 }
   | CHARLITERAL             { 0 }
   | STRINGLITERAL           { 0 }
   | BOOLLITERAL             { 0 }
   | ID                      { 0 }
   | ID table_sel            { 0 }
   | ID line_sel             { 0 }
   | ID cell_sel             { 0 }
   | ID line_of_line         { 0 }
   | ID scalar_of_line       { 0 }
   | expr PLUS   expr        { 0 }
   | expr MINUS  expr        { 0 }
   | expr TIMES  expr        { 0 }
   | expr DIVIDE expr        { 0 }
   | expr EQ     expr        { 0 }
   | expr NEQ    expr        { 0 }
   | expr LT     expr        { 0 }
   | expr LEQ    expr        { 0 }
   | expr GT     expr        { 0 }
   | expr GEQ    expr        { 0 }
   | expr AND    expr        { 0 }
   | expr OR     expr        { 0 }
   | NOT expr                { 0 }
   | ID ASSIGN expr          { 0 }
   | ID LPAREN actuals_opt RPAREN { 0 }             /* Function Call */
   | LPAREN expr RPAREN      { 0 }

actuals_opt:
     /* nothing */   { 0 }
   | actuals_list    { 0 }

actuals_list:
     expr                      { 0 }
   | actuals_list COMMA expr { 0 }
```