TabPro:
A Table Generation language

Anureet Dhillon **Lakshmi Nadig** **Rajat Dixit**

19 December 2008

Contents

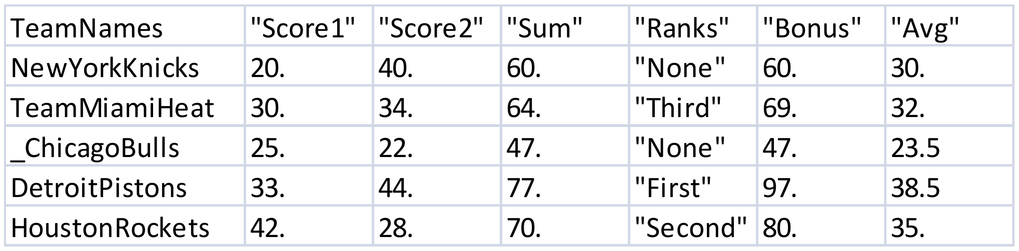
1.An Introduction to TabPro

1.1 Background

We live on sharing information and we try to do it in as simple way as it can be. One of the simplest ways of organizing the information is table. Tables are used around the world in various places, for example we need a table to organize a schedule, to display scores in various sports like basketball, bowling, baseball, etc. A table helps anyone to interpret the information very easily, since it becomes easier for us to visualize the information. Tables can also be used to show the balance available in the bank account of a person, etc. As we know there is several numbers of programming languages, we still feel a need of a language that enables user to create a tabular form of the data/information. Thus looking at the importance of the table, we developed a programming language, TabPro, which allows us to create and manipulate data in a tabular form.

1.2 Goals of TabPro

The objective of this language is to give the user both a flexible and simple way of manipulating and generating data in the form of a table. This language should aid the user in visualizing the data/information. The user should be able to define the organization of the data in a simple way and at the same time give them the power to manipulate the data before generating the final data in a table layout. To begin with, this language aims are generating tables in the form of comma separated files. The comma separated file can be opened using a spread sheet editor. A screen shot of the sample output is shown below. Further extension would include generation of tables in html files



TeamNames	"Score1"	"Score2"	"Sum"	"Ranks"	"Bonus"	"Avg"
NewYorkKnicks	20.	40.	60.	"None"	60.	30.
TeamMiamiHeat	30.	34.	64.	"Third"	69.	32.
_ChicagoBulls	25.	22.	47.	"None"	47.	23.5
DetroitPistons	33.	44.	77.	"First"	97.	38.5
HoustonRockets	42.	28.	70.	"Second"	80.	35.

2. Language Tutorial

This tutorial is for those people who want to learn to work in TabPro. Of course any knowledge of other programming languages or any general computer skill can be useful to better understand this tutorial, although it is not essential.

2.1 Preparing Tabpro

OCaml has to be installed on the user's computer. To make TabPro run, the user has to download the TabPro executable. To run a file test.tab, following command should be used (being in the current directory)-

```
ocamlrun TabPro test.tab
```

2.2 Declaring Variables

Declaring any variable in TabPro is simple. There can be four basic types of variables (as explained in LRM), and that can be declared as follow:

To simply declare a variable of type number-

```
num: i;
```

To initialize a variable of type number, you have to give the decimal value in the form of [0-9].[0-9]. For example-

```
i = 1.0;
```

To declare and initialize a variable of type string, you have to give the value in double inverted commas. For example-

```
str:name;

name = “Rajat”;
```

If you want to declare any column, then you must also mention its data type along with its heading.

```
col:c0<num:,"MyHeading">;
```

And if you want to give values for the declared column, you can use assign operator along with the declaration and a pair of curly braces - { }. For example-

```
col:c0<str:,"MyHeading"> = {“Sno.”}
```

2.3 Declaring and Calling Functions

To declare a function, we use a keyword function followed by function name and an argument list with their data type and return type at the end. Here is a small example-

```
function multiply(num:i, num:j) num:
{
num: k;
k =i* j;
return k;
}
```

To call this function, you need to declare a variable which is of the type similar to the return type of the function. For example here in our example it is done as-

```
num: t;

t = multiply(3,3);
```

The return type of the function in its definition should be the same as the type of the variable holding its value after calling it.

2.4 Working with Columns and indexes

Indexing on a column will return the element at that position. For example if you would like to access/retrieve the 3'rd element of a column mycol, you use mycol[3]

With indexing a new element can be appended to the array of the index is the length of the column, so if a column already has 3 elements, the fourth one can be appended by merely coding it as c[3]. This is especially useful when initializing a column with values.

An error is given if the column index is greater than the column length. The length of the column can be determined using c.size.

You can also apply operators +=, *+ , /= and -= on a range of columns in a particular row- For example- to add 13 to all elements of a column myCol

myCol[3-7] += 13;

Similarly other operators defined in the LRM can also be applied. (See LRM for details)

2.5 Generating Table

generate_table is used to generate the table. The example below demonstrates the generation of a table for the six columns c0, c1, c2, c3, c4 and c5.

generate_table c0, c1, c2, c3, c4, c5;

The example below will generate a time table for a few courses in fall semester for each day of the week-

```
col:Days<str:, “Days”> = {“Monday”, “Tuesday”, “Wednesday”, “Thursday”, “Friday”};
col:Time1 <str:, “11.00am”> = { “ COMS 4119 ”, “COMS 4156” , “COMS 4119 ”, “COMS 4156” , “  ”};

col:Time2<str:, “1.10pm”> = { “COMS 4115”, “ COMS 4111 ” , “COMS 4115”, “  COMS 4111” , “  ”};

col:Time3<str:, “2.40pm”> = { “COMS 4118”, “ COMS 4187 ” , “ COMS 4118”, “ COMS 4187 ” , “  ”};

generate_table Days, Time1, Time2, Time3
```

Thus, the output will look like-

```
Days        11.00 am        1.10 pm         2.40 pm

Monday      COMS 4119       COMS 4115       COMS 4118

Tuesday     COMS 4156       COMS 4111       COMS 4187

Wednesday   COMS 4119       COMS 4115       COMS 4118

Thursday    COMS 4156       COMS 4111       COMS 4187

Friday      NoData          NoData          NoData
```

2.6 Simple function

Let us now learn how to write a simple function in TabPro. Here we will be using a loop that will generate a Fibonacci Series in one column-

```
col:Fibonacci<num:,"Series">={1.0};
num: num1;
num: num2;

num: ans;

num1 = 0;
num2 =1.0;

num:loopvar = 2.0;

  Fibonacci[0] = num1;
  Fibonacci[1] = num2;

loop (loopvar < 8.0 )
{
   ans = num1 + num2;

   Fibonacci[i] = ans;
   num1 = num2;

   num2 = ans;

  loopvar = loopvar + 1.0;
}
generate_table  c0;
```

The above program explains the use of loop (iteration statement). The table generated will have just one column Fibonacii with heading Series and datatype num. There are other two variables of type num num1 and num2. The loop runs from 2 to 7 (6 times) and increments the value of num1 and num2. loopvar here is the loop variable and ans is the variable that is used to hold the addition of num1 and num2. The output of the program will contain 8 rows of column Fibonacci with heading 'Series'. Here is how the output looks like-

```
Series

0

1.0

1.0

2.0

3.0

5.0

8.0

13.0
```

2.7 More Examples

The example below demonstrated the use of conditional statements using the "if" syntax. This program generates a bill as a simple table

```
col:Items <str:,"ITEMS ">={ “Grocery”, “Grocery”, “Stationary”, “Grocery”, “Stationary”, “Misc”, “GRAND
TOTAL”};
col:Qnty <num:,"QUANTITY ">={ 2.0, 1.0, 3.0, 1.0, 4.0, 3.0};
col:Price <num:,"PRICE ">={ 12.0 10.0, 8.0, 15.0, 7.5, 7.0};
col:Amnt <num:,"AMOUNT ">;

col:Cmnt <str:,"COMMENTS ">;
num: total = 0.0;

num: loopvar = 0.0;

loop (loopvar < 6.0)
{
j = Qnty[loopvar];
k = Price[loopvar];
m = j * k;

                    Amnt[loopvar] = m;

                    if (m > 25)

                    {

                     str: comment = “10 % Discount”;

                     Cmnt [i] = comment;

                     m = m/10;

                     temp = m * 9;

                    Amnt [i] = temp;

                    }

total = total + Amnt[loopvar];
loovar = loopvar + 1;
}

num: totindex;

totindex = loopvar + 1;

Amnt[ totindex ]  = total;

generate_table Items, Qnty, Price, Amnt, Cmnt;
```

In this program, we are generating a bill for a grocery shop in the form of a table, where you can actually buy stationary and other stuff as well. It gives its customer a 10% Discount on every purchase if the amount payable of a single item is more than 25 dollars (no matter what is the quantity of the item you take). So, in the above example, a customer buys six different items, each of different in quantities, with different prices. Thus, we write the code to generate the appropriate bill.

Here we put a Boolean condition m > 25 in the if ( ) statement to check whether the amount payable to (loopvar)th item is more than or less than 25. If the amount is more than 25, it is eligible for the 10% discount and thus the new (discounted) amount overwrites the older amount, and thus we also put a tag of ‘10% discount’ in the Cmnt column. Here is the output generated-

```
ITEMS                QUANTITY   PRICE         AMOUNT     COMMENTS

Grocery                  2.0         12.0         24.0          nodata

Grocery                  1.0         10.0         10.0          nodata

Stationary               3.0         8.0          24.0          nodata

Grocery                  1.0         15.0         15.0          nodata

Stationary               4.0         7.5          27.0          10% Discount

Misc                    3.0          7.0          21.0          nodata

GRAND TOTAL             nodata       nodata      121.0          nodata
```

2.8 Exceptions and Errors

The following table shall help you to diagnose the error that is arising while you write a program in TabPro.

ERROR	POSSIBLE CAUSE	POSSIBLE SOLUTION
Divide By Zero Error	You would have done a division of two variables in which denominator has a null (zero) value or an expression like- x/0.0;	Try dividing the numerator by some non-zero value, otherwise avoid such division
Or Not Supported	Trying to perform a Boolean 'OR' operation over two operands. E.g. x = op1 or op2;	OR operator is not been supported by TabPro
And Not Supported	Trying to perform a Boolean 'AND' operation over two operands. E.g. x = op1 and op2;	AND operator is not been supported by TabPro
identifier not initialized	The identifier has not been initialized	Make sure that all the identifiers have been declared
Range start index expression not a number Or Index expression not a number	The starting index of the range that you are specifying in your statement is not of type number. E.g. 'c' being a String type, you write a statement like Col[c-9]	Use a valid statement to specify the range by using the number type in your statement E.g. Col[0-9]
Range end index expression not a number Or	The starting index of the range that you are specifying in your statement is not of type number. E.g. 'c' being a	Use a valid statement to specify the range by using the number type in your statement E.g.

Index expression not a number	String type, you write a statement like Col[0-c]	Col[0-9]

Id Not Found
 	
Or
 	
Undeclared Identifier
	The statement contains an identifier that is probably not defined. This may happen when you try operating on an undeclared identifier. E.g. your code is like-
	
	int:x;
	
	int:y;
	
	z = x + y;
		Define the identifier in your program with appropriate data type. E.g. the example code can be corrected as-
		
		int:x;
		
		int:y;
		
		int:z;
		
		z = x + y;

Incompatible assignment for index range	The data type specified in the index range is not number type.	The data type specified in the index range must be of the number type.
Incompatible assignment Or Not the right type	The data type on the right hand side of the assignment operator is not compatible to that on the left hand side.	Make sure the values that you are assigning are of the same type as that of the identifier on the left side.
Cannot determine length of this retVal	Cannot determine the length of the column list	Make sure the identifier corresponds to the column data type
index id out of range	If you are trying to operate on more number of columns than you have actually declared	Try reducing the index to less than or equal to what you have declared
Types not compatible in assignment, Expected Number type	The data type on the right hand side of the assignment operator is not of Number.	Make sure the values that you are assigning are of the same type as that of the identifier on the left side. Number type is expected
Types not compatible in assignment, Expected String type	The data type on the right hand side of the assignment operator is not compatible to that on the left hand side. String type is expected	Make sure the values that you are assigning are of the same type as that of the identifier on the left side. String type is expected
Types not compatible in assignment, Expected	The data type on the right hand side of the assignment operator is not compatible to	Make sure the values that you are assigning are of the same type as that of the

Number column	that on the left hand side. The type of the column is expected to be of Number type.	identifier on the left side. The type of the column should be of Number type.
Types not compatible in assignment, Expected String column	The data type on the right hand side of the assignment operator is not compatible to that on the left hand side. The type of the column is expected to be of String type.	Make sure the values that you are assigning are of the same type as that of the identifier on the left side. The type of the column should be of String type.
Column Index Not Found	You may have not specified the column index in the expression.	Specify a limited index for the column you are declaring
Not a column type	Type mismatch. Column type expected.	Column type is expected.
Undefined Function	Some function has not been declared in the program	Check whether all the functions are declared in your program
Wrong number of arguments passed to <function name>	You are trying to pass less or more number of the arguments to the function	Check the definition of the function and make sure the number of arguments in the function call is equal to that in the definition
Redefined Identifier	The identifier that you are trying to declare has already been declared in the program	Change the name of the identifier to some other name that has not been declared in the program
Content type not supported	Data type that you are declaring in the argument list of the function is wrong	Make sure you declare the correct data type in the argument list of the function
String list assigned to non str content column	The data type of the column is not of the String type	Check the definition of the column on which you are trying to operate and assign the appropriate values
Return type not recognized	Function declaration is missing the return type	Specify a proper return type for the function
Redefined function name	The function that you are trying to declare has already been declared in the program	Change the name of the function to some other name that has not been declared in the program

Loop Expression not evaluating to number	The identifier used in the loop statement is not of type Number	Make sure the identifier is of the type Number
If expression not evaluating to number	If statement is not syntactically correct	Make sure the syntax of the if statement is correct
Generate Statement allowed in main only	You might be using the generate function outside main	Make sure you use generate function in main only
Cannot generate table with Num/str type identifier	The list to generate the table contains the identifier having Num or Str type	Make sure the list have all the identifiers of type column
Cannot generate table with str type identifier	The list to generate the table contains the identifier having Str type	Make sure the list have all the identifiers of type column
Cannot generate table with Num type identifier	The list to generate the table contains the identifier having Num type	Make sure the list have all the identifiers of type column
Not a list	The set of arguments that you are specifying to generate a table is not a list	Make sure that the list of arguments is a list of columns
Undeclared identifier list ending with id	The list of columns contains an undeclared identifier	Make sure all the identifiers are declared
Other Parse errors	The file may not contain a semicolon as a statement separator.	Add semicolon

3. Reference Manual

3.1 Lexical Conventions

A TabPro program consists of a single translation unit stored in a file which is written using the ASCII character set. The file is scanned in a forward manner starting from the logical start of the file to the end of file. Tokens are separated from each other by using a white space or a new line.

All token are listed in the files in the appendix.

3.1.1 whitespace:

TabPro ignores whitespace. Whitespace characters consist of newlines, carriage returns, tabs and spaces. It could be also combination of the above mentioned characters.

3.1.2 Comments:

Single line comments are supported. The character ‘?’ introduces a comment, which terminates at the end of that line. Comments do not nest and they do not occur within a string or character literals.

3.1.3 Identifiers:

An identifier represents a variable name or a function name. Identifier is a combination of alphabets and digits where the first character has to be an alphabet. Special characters are not permitted. The maximum length of an identifier could be ten. Two identifiers are considered equal if the characters of their names match.

3.1.4 Keywords:

Below is the list of identifiers that has been reserved by TabPro for the use of keywords:

num	str	loop
return	function	generate_table
col	if	else
size	include	

3.1.5 Constants

The kinds of constants used in TabPro are listed below:

Number Constants: These include decimal integers which are a string of decimal digits from [0-9] and real numbers of the form [0-9][.][0-9].

String Constants: A string constant is enclosed in double quotes. The quotes are not considered as a part of the constant.

3.1.6 Declarations

TabPro supports the number type, string type and column declarations. More details on this are covered in 3.4.1

3.1.7 Operators:

(a) All basic arithmetic operators (+, - , *, /) are supported by TabPro. These operators work only on number types and number constants.

(b) Operators like '+=', '-=', '*=', '/=' are used to indicate the arithmetic operation followed by the assignment of the result of right side to the identifier of left side.

These assignment arithmetic operators can be applied to range of elements of a column if the column content type is a number type column. Example:

mycol[0-5]+=10.0

(c) '[]' operator is supported to access a column's elements.

Therefore, in order to access the 5th element of a column called mycol , one might use mycol[4].

(d) Relational operators like ==, !=, <, >, are supported for evaluating a relational expressions to a 1 or a 0. 1 corresponds to its true counterpart and 0 corresponds to false. These operators work accept only number types or number constants as their operands.

(The precedence of these operators is same as their counterparts in C language).

3.1.8 Separators:

A semicolon ";" is a statement separator which is the end of an executable statement.

3.1.9 Scope and Name Space:

TabPro supports static scoping. When a function is called a new local scope is created. So broadly there will be a global scope and a local scope (for every function).

TabPro supports a single name space i.e. only one identifier can have a particular name, be it a function or an identifier.

3.1.10 Built in functions and reserved keywords:

- **generate_table col1, col2, ….**

This is used to print the table in a comma separated file. The comma separated file will have the same name as the source file. The table will be generated for all columns listed after the generate_table keyword.

- **size**

This keyword indicates the size of the row/column in context. For instance, mycol.size would refer to the size of "mycol" column i.e. the number of columns in mycol.

3.2 Types:

TabPro supports the following data types:

1. num: Decimal integers are allowed which are a string of decimal digits from [0-9] or real numbers of the form [0-9][.][0-9].

2. str: Strings are allowed which are a collection of ASCII characters.

3. col: This data type is used to declare a column of elements. This is a inherently a list of elements. The list grows whenever the user appends a new element at the end of the list.For Instance, col: mycol<num:, “scores”> = {4,5,6,7}; declares a column of a table having four columns initialized to 4, 5, 6 and 7.

3.3 Expressions

Left-or right associative property of operators in expression is defined in the respective subsections. The precedence of the operators used in the expressions or sub-expressions is the same through out-highest precedence first. In an expression, if the order of evaluation of operator is not coming in the picture, the expression is independently evaluated.

The handling of exceptions like overflow, divide by zero check, and others in an expression is not defined by the language.

3.3.1 Primary Expressions

Primary expressions are identifiers, numbers and strings.

primary-expression
identifier
number
string constant
int constant
number constant

An identifier is a primary expression provided that it has been suitably declared with a specified type. An identifier basically refers a variable name or a function name (As discussed in 1.3).

A string literal is a primary expression. A string is basically a series of characters and/or integers and/or both.

A constant is a primary expression. Its can be either of the types discussed in 1.5.

It should be noted that parenthesized expressions (like (expression)) are not primary expressions and are not supported by our language.

3.3.2 Indexed expression

This is of the form <identifier>[<primary expression>] . For e.g. mycol[4] refers to the fifth element of the column mycol. A primary expression here denotes a number constant or a literal.

Note that this can also be the left hand operand for assignment expression.There are 3 cases in this

Case1: If the indexed position (primary expression) evaluates to a number between the start and end, that element is replaced by the evaluated expression on the right.

Case2: If the indexed position (primary expression) evaluates to a number that equals the length of the column, a new element is appended with the value as evaluated expression on the right.

Case3: If the indexed position (primary expression) evaluates to a number greater than the length of the column, an error message is displayed.

3.3.3 Index range expression

This is of the form <identifier>[<primary expression>-< primary expression >] = <general expression>

For e.g. mycol[4-7] refers to the range of fifth to the eighth elements of the column mycol. This expression can be used as a left hand operand for arithmetic assignment operators +=, -=, *= and /= if the column types are columns whose content types are number types. As an example in order to increment elements 5 though 8 of a column mycol by 10, the expression can be written as

mycol[4-7] += 10

3.3.4 Column Size expression

This is of the form <identifier>.size . This returns the length of the column identifier. If the identifier is not of the type column an error is displayed.

3.3.5 Function Calls

A function call is the identifier (function name) followed by parentheses, containing possibly empty, comma separated list of assignment expressions, which constitutes to the arguments to the function.

An example for a function call would be changeData (x)

The term argument is given for an expression passed by a function call while the term parameter is used for an input object received by the function definition.

For instance, my function(num:x,num:y):num is a function declaration with x and y as arguments and my function(4,6) has 4 and 6 as parameters to this function.

3.3.6 Expressions with Arithmetic Operators

Multiplicative and additive operators * , /,+,- are grouped left to right. The expressions on both sides of operators should evaluate to a number data type or a number constant. Arithmetic operators are not supported for string and column data types.

arithmetic-expression:
*arithmetic -expression * arithmetic –expression*
arithmetic -expression / arithmetic –expression
arithmetic-expression+arithmetic-expression
arithmetic -expression - arithmetic -expression

3.3.7 Expressions with Relational Operators

The relational operators group left-to-right, but a<b<c is not supported by our language. The result generated will be 0.0 if the expression is false and 1 if the expression is true. The == (equal to) and the != (not equal to)

operators are analogous to the relational operators except for their lower precedence and are used to compare the expressions on either side of the operator. The equality operator follows the same rule- if the expression is true then it will return value 1.0, and if it is false, it would return value 0.0.

relational-expression:
relational-expression<relational-expression
relational-expression>relational-expression
relational-expression==relational-expression
relational-expression != relational-expression

The operators used here hold usual meanings and are supported to compare two expressions.

3.3.8 Expressions with Assignment operators

The basic assignment expression is of the form

<identifier> = expression

The left operand must be modifiable and must not have an incomplete type, or must not be a function. The type of an assignment expression is that of its left operand, and the value is the value stored in the left operand after the assignment has taken place. In the simple assignment with =, the value of the expression replaces that of the object referred to by the lvalue. Both operands must evaluate to the same type.

The left operand can also be an indexed expression as described in 3.3.2

The assignment operator is also used in assigning initial values to a column data type. This is of the form

<column declaration> = { {comma separated string or number constants>}

3.3.9 Expression with Comma Operator

Parts of expressions are separated by a comma, since the comma is used as a separator while declaring a row. For example-

row:mycol<num:, “numbers”>={1,2,3,4,5,6,7}

declares a column and the elements of the columns “1,2,3,4,5,6,7” are number constants separated by the comma operator.

The comma is also used as a separator of function arguments and function parameters

3.4 Statements

3.4.1 Declaration Statement

A declaration statement can be either a simple declaration statement , a column declaration statement or a column declaration statement with initialization.The following types of declarations are supported :

num: <identifier name> to declare a number type

str: <identifier name> to declare the string type

col:<identifiername> < <content type >, <column heading>> to declare a column type.

Examples:

num:i

str: s

col:mycol<num: , “Column A”>

3.4.2 Block Statement

These include one or more statements that are nested within the curly braces. Block statements appear in Conditional and Iteration statements as explained in the sections below.

3.4.3 Conditional statements

The basic conditional statement consists of the if keyword, followed by an expression that evaluates 1 or 0, and statements that have to be enclosed within the open and close parenthesis even if it is just one statement

Abstract example:

if expression {

statement(s)

}

else {

statement(s)

}

3.4.4 return statement

The return statement in a function is optional and returns a value.

For example:

return <expression>.

3.4.5 Iteration Statements

loop(expression){

Block of statement(s)

}

This is our basic looping mechanism. The block statements will be executed as long as the expression evaluates to 1

3.4.6 *Function Declaration*

A function declaration declares a block of code that can be executed by a function call. A function declaration can also be made to define a block of statements to be performed on all columns of a row or on all rows of a passed column.

To define a function, start with the function keyword and follow it with an identifier to serve as the function's name, followed by an optional list of function arguments separated by a comma, each having a type declaration prefix. The return type could be num: or str: or col:<num:, "<heading> "> or col:<str:, "<heading> ">

A function with an empty body is not supported.

Abstract example:

```
function <identifier> (list of <type:identifier>) <return type >

{

statement(s);

}
```

A more concrete example:

```
function product(num:I, num:j) num:

{

k = I * j;

}
```

3.5 *Pre-Processor/Library Support*

The language will provide one library of functions through the preprocessor support.There are a set of functions written in TabPro and are available in "mathlib" which implements the functions average for column and number data types, square and power for number types. The user has to run the preprocessor on the source file including this file. The user can include these functions by using the include directive. The "include directive" syntax is as below:

```
include <libname> ;
```

For instance to include the "mathlib" functions provided with TabPro support, the user would add the following line to the source file.

```
include mathlib ;
```

The Interpreter will ignore all these include directive lines.

The user is expected to run the TabProPreProcess on the source file before running an interpretation of the file. The Tabpro preprocessor will include all functions in the include file into a new source file appended with append “PP”

The preprocessor implementation does not perform any checks with respect to the syntax of the library being included. Only the last encountered library will be included if the source file has multiple include statements. The verification will be done by the Interpreter when the user runs TabPro on the preprocessed file (.tabPP)

4. Project Plan

4.1 Team Responsibilities

Anureet Dhillon	Scanner implementation, LRM preparation,Grammar rules,Parser (Expressions),Parser Tests , Expressions in AST Interpreter (Expressions) , TableUtil - Table Generation in csv files, Interpreter Tests and Debugging Build Regression test env , Demo Test programs Final report preparation.
Lakshmi Nadig	Scanner & Parser Implementation , Scanner & Parser test framework, LRM preparation , Grammar rules, Interpreter (Statements , Call expression, & the flow of the all of the environment), AST , TableUtil (algo to generate a table from the list of lists) Printer functions , formulated the regression test plan, Final report preparation
Rajat Dixit	LRM preparation, grammar rules for statements, Parser Tests , Implemented Preprocessor functionality, wrote the library file, Final report preparation

4.2 Project Timeline

The following deadlines were set for key project development goals.

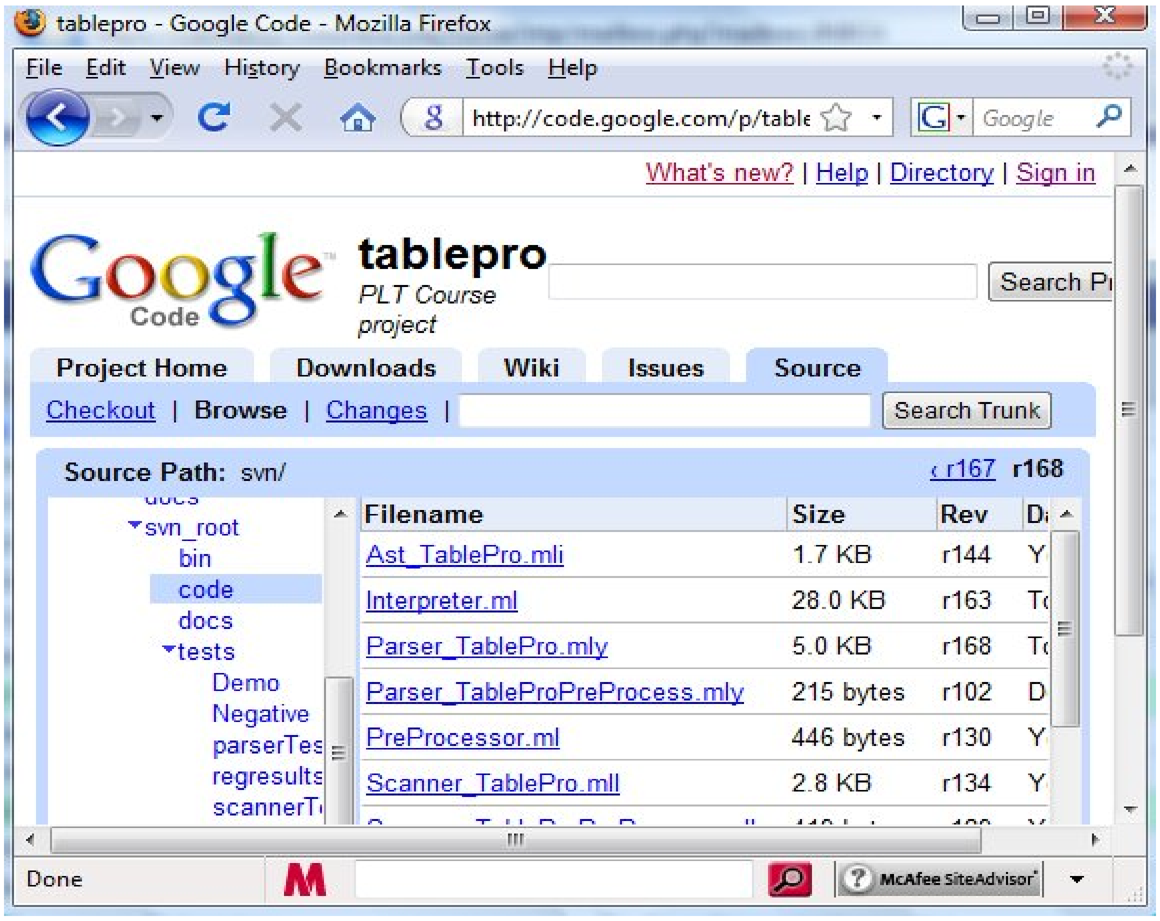
9-22-2008	Proposal
10-22-2008	LRM , Scanner working, Scanner tests, Parser Grammar
11-10-2008	Parser working, Parser Tests
11-22-2008	Symbol Tables and outline of AST walk through
11-30-2008	Interpreter's first working program
12-09-2008	Initial Interpreter tests
12-15-2008	Final Phase of testing
12-19-2008	Final report ready

4.3 Software Development Environment

TabPro was developed on Windows environment using Ocaml for Windows. Tools used were:

1. Ocamllex: An Ocaml tool for the scanner.
2. Ocamlyacc: An Ocaml tool for the parser.
3. Windiff : A comparison tool for Regression Testing.

The structure for Google code repository mirrors the development code organization as shown below:



Filename	Size	Rev	D
Ast_TablePro.mli	1.7 KB	r144	Y
Interpreter.ml	28.0 KB	r163	T
Parser_TablePro.mly	5.0 KB	r168	T
Parser_TableProPreProcess.mly	215 bytes	r102	D
PreProcessor.ml	446 bytes	r130	Y
Scanner_TablePro.mll	2.8 KB	r134	Y

The development environment for Tab Pro includes Tortoise SVN, which would be required for getting access to the Google code repository, windiff would be required for running the regression tests, and Ocaml for windows.

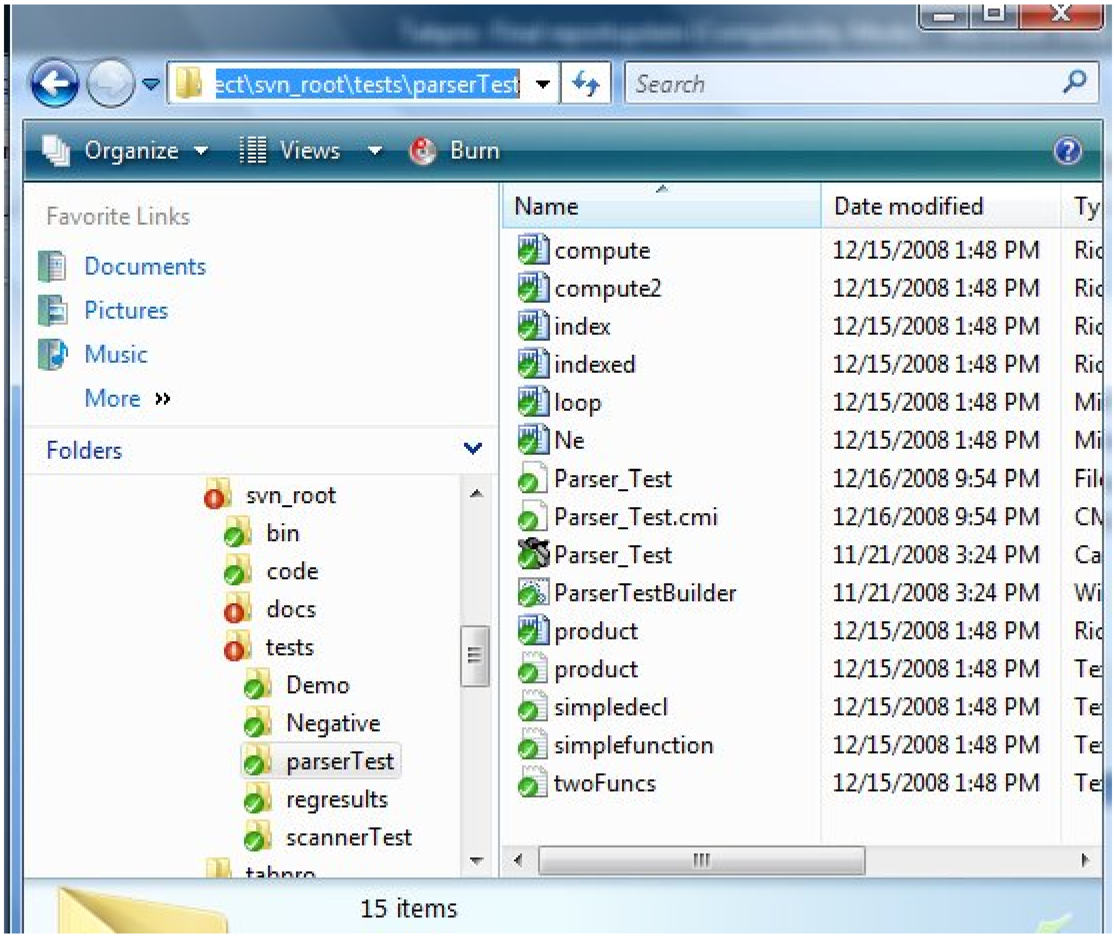
Following is the development folder structure:

Code: This contains all the source code

Bin: This contains the executables TabPro, TabProPreProcess and the windiff executable

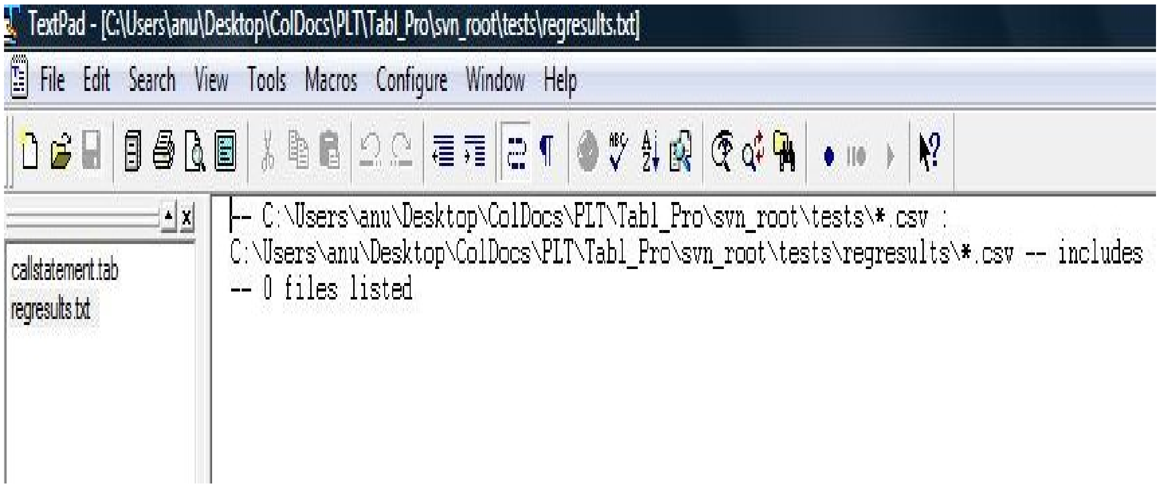
Docs: All the related documents

Tests: This folder contains the different kinds of tests – Scanner tests, Parser tests, Negative tests, Regression results, and the regression tests cases. A screen shot of the folder structure is shown below. The Scanner tests contain the code for testing the scanner and the batch file to build the scanner tests. The Parser tests contain the code, batch file to build the parser test and the parser test test-files.



Name	Date modified	Ty
compute	12/15/2008 1:48 PM	Ric
compute2	12/15/2008 1:48 PM	Ric
index	12/15/2008 1:48 PM	Ric
indexed	12/15/2008 1:48 PM	Ric
loop	12/15/2008 1:48 PM	Mi
Ne	12/15/2008 1:48 PM	Mi
Parser_Test	12/16/2008 9:54 PM	Fil
Parser_Test.cmi	12/16/2008 9:54 PM	CN
Parser_Test	11/21/2008 3:24 PM	Ca
ParserTestBuilder	11/21/2008 3:24 PM	Wi
product	12/15/2008 1:48 PM	Ric
product	12/15/2008 1:48 PM	Te
simpledecl	12/15/2008 1:48 PM	Te
simplefunction	12/15/2008 1:48 PM	Te
twoFuncs	12/15/2008 1:48 PM	Te

A screenshot of running windiff is attached below. It shows that after updating the code, no change in results occurred.


TextPad - [C:\Users\anu\Desktop\ColDocs\PLT\Tabl_Pro\svn_root\tests\regresults.txt]
File Edit Search View Tools Macros Configure Window Help
callstatement.tab
regresults.txt

```
|-- C:\Users\anu\Desktop\ColDocs\PLT\Tabl_Pro\svn_root\tests\*.csv :
C:\Users\anu\Desktop\ColDocs\PLT\Tabl_Pro\svn_root\tests\regresults\*.csv -- includes
-- 0 files listed
```

4.4 Project Log

The major milestones achieved are mentioned below:

9-15-2008	Discussion of project ideas of all team members and selection of one (TabPro).
9-17-2008	Discussion of basic language features.
9-22-2008	Submission of project proposal.
10-10-2008	In depth discussion of the language grammar.
10-15-2008	Work on scanner started. LRM documentation started.
10-18-2008	Scanner tested.
10-20-2008	Work on parser Grammar started.
10-22-2008	LRM finalized and submitted. Parser work in progress.
11-10-2008	Parser completed and tested.
11-15-2008	Meeting with TA to decide on AST structure.
11-22-2008	Symbol table completed.
11-25-2008	Major Expressions like Binop, Assignment and id completed.
11-27-2008	Expressions and Statements completed.
11-30-2008	Simple interpreter test run to print Hello World in tabular format & LRM updated to reflect new change
12-5-2008	Interpreter Tests for testing Expressions. Report documentation Started.
12-9-2008	Interpreter Tests for testing statements.
12-10-2008	Negative test cases for interpreter added.
12-12-2008	Functionality for generating output in csv file added.
12-13-2008	Functionality for regression testing using windiff added.
12-19-2008	Final Report Submitted.

5. Architectural Design

5.1 Architecture

TabPro design consists of two major parts, design of the front end and the back end. The front end consists of the lexer, parser and the AST walker. The backend consists of the implementation for data types, expression evaluation, statement evaluation, type checking, type conversion and generating the output for csv file. The block diagram representing a high level structure of the interpreter is given below.

The source program of the TabPro in form of .tab or .tabPP is fed to the lexer. The lexer converts this .tab file into a stream of tokens and passes it onto to the parser. The parser is responsible for analyzing the structure of the program and confirms whether the program written is in sync with the grammar rules of the language defined into the parser. The parser then translates the parsed program into a syntax tree. The interpreter relies on this parse tree where different data nodes are created as the syntax tree is read and symbol table is checked. The interpreter is mainly responsible for:

Expression Evaluation: Matching expressions, querying the local and the global symbol tables, checking type, converting types and returning the evaluated expression.

Statement Evaluation: Matching the kind of the statement, checking the scope, querying and updating symbol tables and calling the expression evaluation method as and when needed.

Generation of the table: After manipulations, the table is generated in the csv file format. It makes use of the Tableutil file which has functions for operating on the columns and printing them in tabular format.

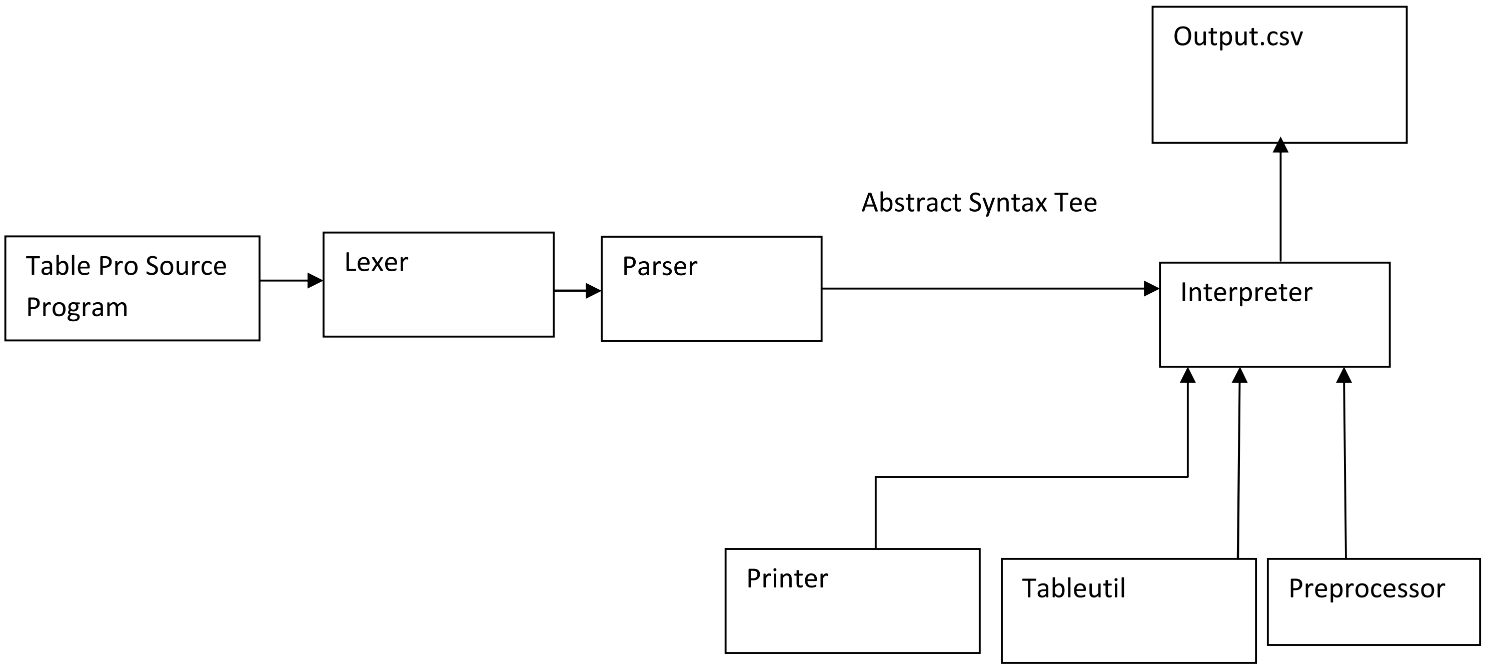


Figure 1: TabPro Architecture

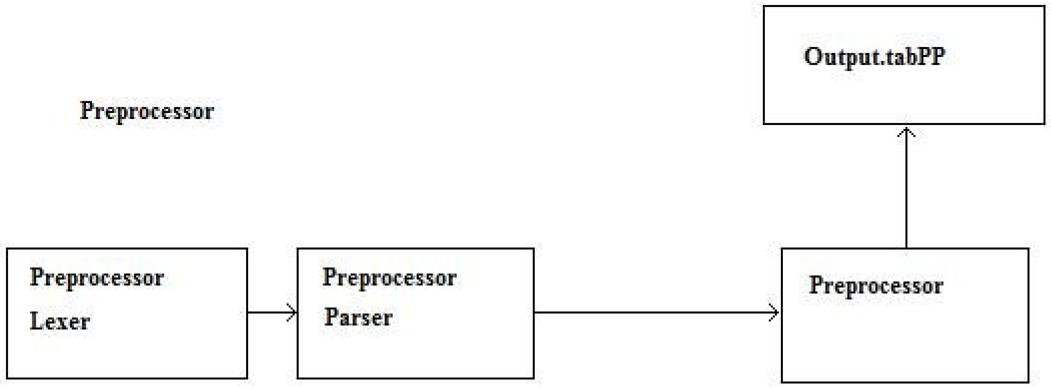


Figure 2: TabPro Preprocessor Overview

Everything in the program is considered to be a statement. The statement further expands to different kinds of statements as mentioned below. An expression statement can further be divided into different expressions written below.

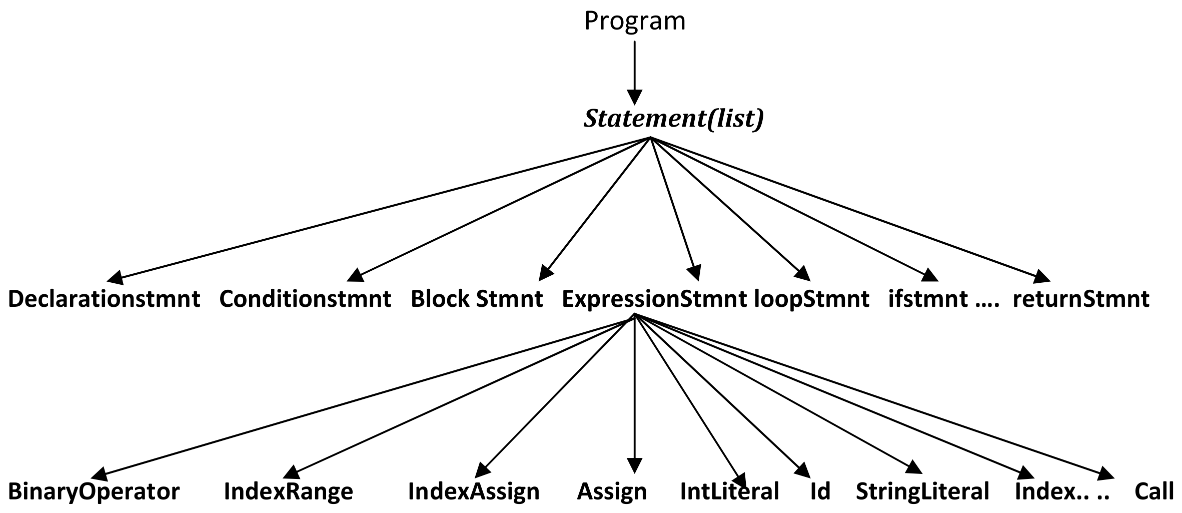


Figure 3: TabPro language overview

5.2 The Runtime Environment

The user has to download the executable, the file mylib (for some sample math functions) and OCaml for Windows.

6. Testing Plan

The testing plan is discussed below. All the test related files are attached to the appendix.

6.1 Goals

We had set the goal of testing the scanner, parser and interpreter separately at every step. We moved to the next stage only after ensuring that the respective components functioned correctly on all possible test cases fed to them. Interpreter was also tested on a number of negative test cases to ensure that the exceptions are being raised as desired.

6.2 Phase 1 – Scanner test

By writing a TabProLexer.ml file having the following code, we were able to test whether our scanner accepts legal tokens.

Code for testing Scanner:

```
let main () =

  try
  let lexbuf = Lexing.from_channel stdin in
  while true do
   Scanner_TableProTest.expr Scanner_TablePro.token lexbuf
  done
 with End_of_file -> exit 0


let _ = Printexc.print main ()
```

A number of tokens were fed in form of a simple test file written below. The out put of the current token being read was displayed on the standard console.

Input given to TabProLexer.ml in order to test some tokens.

```
if else abcdidentifier  loop col: num: string string1 str:  1.6 -5.8  jdhjashdjhs * )
generate_table  [  ] ( )
myarr[1] mycol[3-6] ggg { } (  )
```

6.3 Phase 2 – Parser Tests

To test the parser, we printed on the console whatever input program the user gives. After passing through the scanner and the parser, the input program was printed on the console so as to check whether the parser is functioning as desired.

Following test cases were fed so as to check the parser:

1. To test simple Binary operator Expression.

```
num: i;

i=0;
i = i + 1.0;
num: i;
str: t;
```

2. To test Function declaration statement and generate table statement

```
col:cdecl<num:,"heading">;

function changeData(num:x) num:
{

col:c0<num:,"heading">={ 1.0 , 2.0, 3.0};
col:c1<str:,"heading">={ "Hello",  "fname" , "lname" };

}
```

```
generate_table cnew, c0, c1;
```

3. To test the loop statement.

```
num: i;
col:c0<num:,"Id">={1.0};
num:j ;
i=0;
j =1.0;

loop (i < 10 )
{
  c0[i] = j;
  j = j+1;
  i = i+1;
}
```

4. To test if statement

```
col:c0<str:,"Myheading">={"Hello"};
num: i;
num:j ;
i=15;
j =0.0;

if (i < j )
{
  c0[j] = "less";
}
else
{
  c0[j] = "greater";
}
```

6.4 *Phase 3 – Interpreter Tests*

To test the interpreter, we tried to cover all the combination of different expressions and statements. Positive and negative test cases were written for the same and fed to the interpreter for testing.

1. The following program takes input as the marks of two subjects and displays the output in a CSV file as a table having four columns:

(a) Marks of Mathematics
(b) Marks of English
(c) Total Marks
(d) Average
(e) Percentage

The Total, average and Percentage are calculated in the ComputeMarks() function.

```
num: i;
num: j;
num: k;
```

```
num: l;
num: p;

col:c0<num:,"Mathematics">={ 100.0 , 90.0, 95.0};
col:c1<num:,"English">={ 88.0,  77.0 , 90.0 };
col:cnew<num:,"Total">;
col:cavg<num:,"Average">;
col:cper<num:,"Percentage">;
cnew = c0;

function ComputeTotal(num:x) num:
{
loop (i < 3)
{
j = c0[i];
k = c1[i];
j = j + k;
k = j/200;
p = k * 100;
  cnew[i] = j;
  cavg[i] = k;
  cper[i] = p;
  i = i+1;
}
}
i = 0.0;
ComputeTotal(i);

generate_table c0, c1,cnew,cavg,cper;
```

Output Generated in CSV File:

```
"Mathematics"  "English"      "Total"  "Average"      "Percentage"
100.     88.      188.     0.94     94.
90.      77.      167.     0.835    83.5
95.      90.      185.     0.925    92.5
```

2. To calculate speed given the time and distance:
The input given to the program is the time and speed and the output generated is the following:
(a) A coulmn representing the Speed
(b) A column representing the Time
(c) A coulmn representing the distance calculated from the first two columns.

```
num: i;
num: j;
num: k;

col:c0<num:,"Speed">={ 50.0 , 40.0};
col:c1<num:,"Time">={ 18.0,  20.0};
col:cdist<num:,"Distance">;
cdist = c0;

function distancetravelled(num:x) num:
{
```

```
loop (i < 2)
{
j = c0[i];
k = c1[i];
j = j * k;
  cdist[i] = j;
  i = i+1;
}
}
i = 0.0;
distancetravelled(i);

generate_table c0, c1,cdist;
```

The output Generated in CSV file is:

```
"Speed"      "Time"      "Distance"
50.       18.       900.
40.       20.       800.
```

Negative Test Cases:

1. The following code gives a divide by zero error:

```
num: i;

i=0;
i = i / 0.0;
```

2. The following code tries to generate a table from string type rather than a column type, therefore gives the following error:

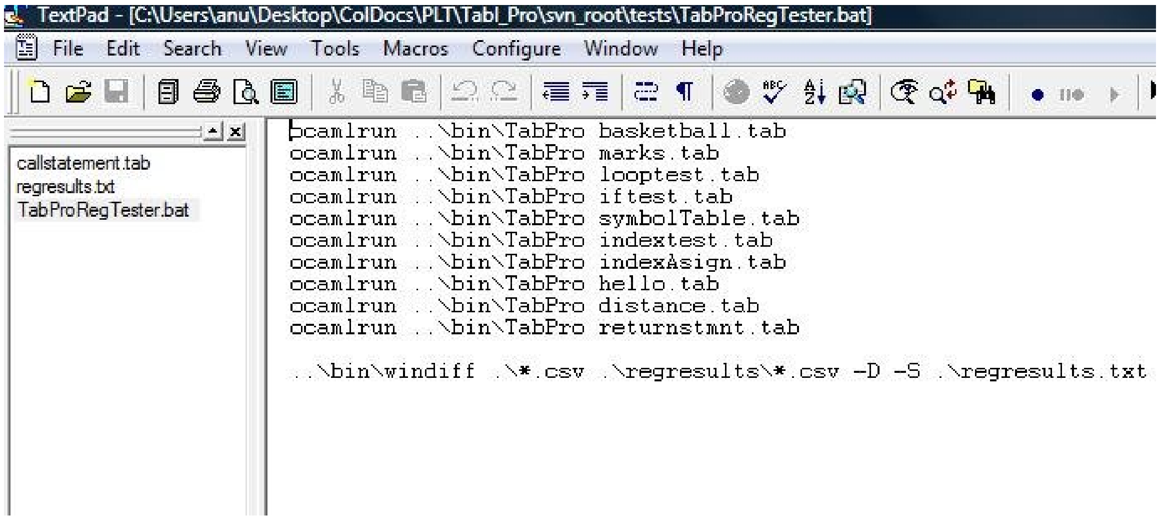
Cannot generate table with Str/ num type identifier "g".

```
col:c0<num:,"heading">={ 1.0 , 2.0, 3.0};
col:c1<str:,"heading">={ "Hello",  "fname" , "lname" };
col:c2<str:,"heading">={ "aa" };

num: i;
str: g;
g = "l";
i = 8-9;
c0[2] =7;

generate_table  c0, c1, g;
```

6.5 *Phase 4 – Regression Tests*

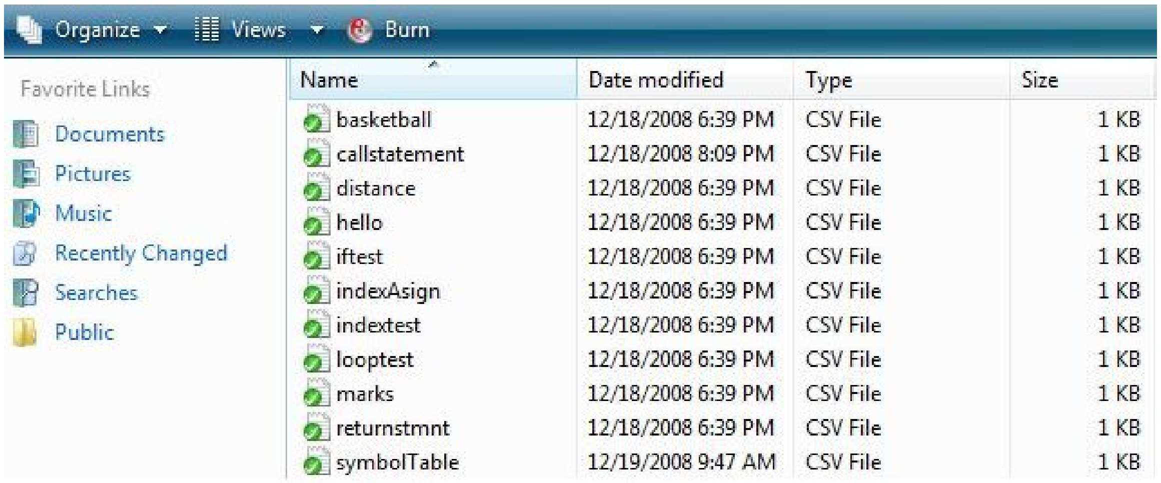
Regression Tests were conducted using the windiff tool in order to find changes(if any) in the new generated csv files.

Attached below is the screenshot for the Regression Test Builder File:



```
ocamlrun ..\bin\TabPro basketball.tab
ocamlrun ..\bin\TabPro marks.tab
ocamlrun ..\bin\TabPro looptest.tab
ocamlrun ..\bin\TabPro iftest.tab
ocamlrun ..\bin\TabPro symbolTable.tab
ocamlrun ..\bin\TabPro indextest.tab
ocamlrun ..\bin\TabPro indexAsign.tab
ocamlrun ..\bin\TabPro hello.tab
ocamlrun ..\bin\TabPro distance.tab
ocamlrun ..\bin\TabPro returnstmnt.tab

..\bin\windiff .\*.csv .\regresults\*.csv -D -S .\regresults.txt
```

Following CSV files were generated as a result of running the above mentioned batch file.



Name	Date modified	Type	Size
basketball	12/18/2008 6:39 PM	CSV File	1 KB
callstatement	12/18/2008 8:09 PM	CSV File	1 KB
distance	12/18/2008 6:39 PM	CSV File	1 KB
hello	12/18/2008 6:39 PM	CSV File	1 KB
iftest	12/18/2008 6:39 PM	CSV File	1 KB
indexAsign	12/18/2008 6:39 PM	CSV File	1 KB
indextest	12/18/2008 6:39 PM	CSV File	1 KB
looptest	12/18/2008 6:39 PM	CSV File	1 KB
marks	12/18/2008 6:39 PM	CSV File	1 KB
returnstmnt	12/18/2008 6:39 PM	CSV File	1 KB
symbolTable	12/19/2008 9:47 AM	CSV File	1 KB

7.Lessons Learned

Ocaml: The project was a very good opportunity to learn Ocaml. It was also a learning experience for functional programming paradigm.

Interpreters: The project gave an insight into Interpreter functioning and program flow. It helped understanding the AST concepts very well.

Team oriented development skills: This project also enhanced out team development skills. Both the pros and cons of team development were experienced.

8. Potential Enhancements

8.1 Short Term

1. Add a row type

2. Add more statistical library functions

3. Add functionality to the preprocessor to make sure this has only functions

8.2 Long Term

1. Add more statistical library functions

2. Enhance the preprocessor to handle a list of includes

3. Generate a html file for the table

4. Appendix

4.1 Code Listing for the main interpreter

4.1.1 Scanner_TablePro.mll

```
{
open Parser_TablePro

}
let digit = ['0'-'9']
let comma = [',']
let semicolon = [';']
let colon = [':']
let dot = ['.']
let exp = ['e''E']
let signednumber = ['-''+']?['0'-'9']*['.']['0'-'9']+
let signedint = ['-''+']?['0'-'9']*
let alphanum = ['a'-'z' 'A'-'Z' '0'-'9']
let alpha = ['a'-'z' 'A'-'Z' '_']
let numType = "num"
let strType = "str"
let rowType = "row"
let colType = "col"
let add = '+'
let subtract = '-'
let times = '*'
let divide = '/'
let equal = '='
let lessThan = '<'
let greaterThan = '>'
let equality =  "=="
let commentStart = '?'
let newLine = '\n' | '\r'
let if = "if"
let else = "else"
let leftParan = '('
let rightParan = ')'
let leftSqParan = '['
let rightSqParan = ']'
let blockBegin = '{'
let blockEnd = '}'
let loop = "loop"
let function = "function"
let size = "size"
let currIndex = "currIndex"
let col_heading = "col_heading"
let row_heading = "row_heading"
let row_limit = "row_limit"
let col_limit = "col_limit"
let col_sort_index = "col_sort_index"
let row_filter_condition = "row_filter_condition"
let return = "return"
let logand= "&&"
let logor = "||"
let quote = "\""
let generateTable = "generate_table"
let includedirective = "include"

rule token =
parse    [' ' '\t' '\r' '\n'] { token lexbuf }
        |  signednumber  as n { NUMBER(float_of_string n) }
        |  digit+ as i { INTCONST (int_of_string i) }
        |  (numType | strType )colon as n{SIMPLETYPE(n)}
        | ( colType ) colon as n{ COMPTYPE (n)}
        |  quote alpha (alphanum)* quote as s {STRCONST (s)}
        |  add {ADD}
        |  subtract {SUBTRACT}
        |  times {MULTIPLY}
        | divide {DIVIDE}
        | greaterThan {GREATERTHAN}
        | lessThan {LESSTHAN}
        | equality {EQUALITY}
              | logand {LOGICALAND}
              | logor {LOGICALOR }
        | equal {ASSIGN}
              |  commentStart {comment lexbuf}
              | includedirective { comment lexbuf }
              |  size {SIZE}
              |  currIndex {CURRINDEX}
              |  if {IF}
              |  else {ELSE}
              |  leftParan {LEFTPARAN}
              |  rightParan {RIGHTPARAN}
              |  leftSqParan {LEFTSQPARAN}
              |  rightSqParan {RIGHTSQPARAN}
              |  blockBegin {BLOCKBEGIN}
              |  blockEnd {BLOCKEND}
              |  loop {LOOP}
              | function {FUNCTION}
              | col_heading {COL_HEADING}
              | row_heading {ROW_HEADING}
              | row_limit {ROW_LIMIT}
              | col_limit {COL_LIMIT}
              | row_filter_condition {ROW_FILTER_CONDITION}
              | col_sort_index {COL_SORT_INDEX}
              | return {RETURN}
              | comma {COMMA}
              | semicolon {SEMICOLON}
        |  alpha (alphanum)* as name {IDENTIFIER(name)}
        |  alpha (alphanum)* as name {IDENTIFIER(name)}
        | generateTable {GENERATE_TABLE}
                | dot {DOT}
                | size {SIZE}
              | eof { EOF}
              | _ as char { raise (Failure("illegal character " ^ Char.escaped char)) }
and comment = parse
'\n'  { token lexbuf } (* End of comment *)
| _ { comment lexbuf } (* Eat everything else *)

{

}
```

4.1.2 Parser_TablePro.mly

```
%{ open Ast_TablePro %}
%token  EOF
%token  <float> NUMBER
%token  <int> INTCONST
%token  <string> IDENTIFIER
%token  <string> SIMPLETYPE
%token  <string> COMPTYPE
%token  COMMA
%token  ADD
%token  SUBTRACT
%token  MULTIPLY
%token  DIVIDE
%token  ASSIGN
%token  SIZE
%token  CURRINDEX
%token  IF
%token  ELSE
%token  LEFTPARAN
%token  RIGHTPARAN
%token  LEFTSQPARAN
%token  RIGHTSQPARAN
%token  LEFTPARAN
%token  BLOCKBEGIN
%token  BLOCKEND
%token  LOOP
%token  GREATERTHAN
%token  LESSTHAN
%token  EQUALITY
%token  LOGICALAND
%token  LOGICALOR
%token  FUNCTION
%token  COL_HEADING
%token  ROW_HEADING
%token  ROW_LIMIT
%token  COL_LIMIT
%token  ROW_FILTER_CONDITION
%token  COL_SORT_INDEX
%token  RETURN
%token  <string> STRCONST
%token  SEMICOLON
%token  BOGUS
%token  DOT
%token  SIZE
%token  GENERATE_TABLE

%left INTCONST
%right ASSIGN
%left GREATERTHAN LESSTHAN EQUALITY
%left LOGICALAND LOGICALOR
%left ADD  SUBTRACT
```

```
%left MULTIPLY DIVIDE


%start program
%type <Ast_TablePro.program> program
%%

expr:
          genExpr {$1 }


primaryexpr:
   NUMBER { Literal($1) }
| INTCONST { IntLiteral ($1)}
| IDENTIFIER { Id($1) }
| STRCONST { StrLiteral($1) }


genExpr:
  genExpr ADD genExpr { Binop($1, Add, $3) }
| genExpr SUBTRACT genExpr { Binop($1, Sub, $3) }
| genExpr MULTIPLY genExpr { Binop($1, Mult, $3) }
| genExpr DIVIDE genExpr { Binop($1, Div, $3) }
| genExpr GREATERTHAN genExpr {Binop($1, Greater, $3) }
| genExpr LESSTHAN genExpr {Binop($1, Less, $3) }
| genExpr EQUALITY genExpr {Binop($1, Equality, $3) }
| genExpr LOGICALAND genExpr {Binop($1, And, $3) }
| genExpr LOGICALOR genExpr {Binop($1, Or, $3) }
| IDENTIFIER ASSIGN genExpr {Assign( $1, $3) }
| indexExpr {$1 }
| indexRelExpr {$1 }
| IDENTIFIER LEFTPARAN argList RIGHTPARAN {Call ($1, $3) }
| NUMBER { Literal($1) }
| INTCONST { IntLiteral ($1)}
| IDENTIFIER { Id($1) }
| STRCONST { StrLiteral($1) }



indexExpr:

  IDENTIFIER LEFTSQPARAN primaryexpr RIGHTSQPARAN ASSIGN genExpr  { IndexAsn($1, $3 , $6) }
| IDENTIFIER LEFTSQPARAN primaryexpr RIGHTSQPARAN {Index($1, $3)}
| IDENTIFIER DOT SIZE {IndexSize($1)}



indexRelExpr:
         IDENTIFIER LEFTSQPARAN primaryexpr SUBTRACT primaryexpr RIGHTSQPARAN  ADD ASSIGN
genExpr {IndexRange($1, $3, $5, Add, $9)}
         | IDENTIFIER LEFTSQPARAN primaryexpr SUBTRACT primaryexpr RIGHTSQPARAN  SUBTRACT
ASSIGN genExpr {IndexRange($1, $3, $5,Sub,  $9)}
         | IDENTIFIER LEFTSQPARAN primaryexpr SUBTRACT primaryexpr RIGHTSQPARAN  MULTIPLY
ASSIGN genExpr {IndexRange($1, $3, $5, Mult, $9)}
         | IDENTIFIER LEFTSQPARAN primaryexpr SUBTRACT primaryexpr RIGHTSQPARAN  DIVIDE ASSIGN
genExpr {IndexRange($1, $3, $5, Div, $9)}

argList:  argList COMMA genExpr {List.rev($3::$1)}
           | genExpr {[$1]}




stmnt:
         sdeclstmnt{  $1 }
         | cdeclstmnt{  $1 }
```

```
            | cdeclAsnstmnt{ $1 }
            | exprstmnt {$1}
            | blockstmnt{$1 }
            | condstmnt{$1 }
            | iterstmnt{$1 }
            | returnstmnt{$1 }
            | funcdeclstmnt{ $1 }
            | gentablestmnt {$1 }


gentablestmnt: GENERATE_TABLE identifierList SEMICOLON {GenerateStmnt($2)};

identifierList: identifierList COMMA IDENTIFIER { $3::$1 }
                        | IDENTIFIER {[$1]};


sdeclstmnt:
            SIMPLETYPE IDENTIFIER SEMICOLON{ SDeclstmnt($1, $2) }

cdeclstmnt:
            COMPTYPE IDENTIFIER LESSTHAN SIMPLETYPE COMMA STRCONST GREATERTHAN
SEMICOLON{CDeclstmnt($1, $4, $6, $2)}

cdeclAsnstmnt:
             COMPTYPE IDENTIFIER LESSTHAN SIMPLETYPE COMMA STRCONST GREATERTHAN  ASSIGN
BLOCKBEGIN strconstList BLOCKEND SEMICOLON {CStrdeclAsnstmnt ($1, $4, $6, $2, $10)}
            | COMPTYPE IDENTIFIER LESSTHAN SIMPLETYPE COMMA STRCONST GREATERTHAN  ASSIGN
BLOCKBEGIN numberList BLOCKEND SEMICOLON {CNumdeclAsnstmnt ($1, $4, $6, $2, $10)}


strconstList: STRCONST COMMA strconstList {$1::$3}
                        | STRCONST {[$1] }


numberList: NUMBER COMMA numberList {$1::$3}
            | NUMBER {[$1]}

exprstmnt:
            expr SEMICOLON {Stmnt($1)}

blockstmnt:
  BLOCKBEGIN stmnt_list BLOCKEND {Block(List.rev($2)) }

stmnt_list:
                        stmnt_list stmnt {($2::$1)}
                        | stmnt {[$1]}


condstmnt:
             IF LEFTPARAN genExpr RIGHTPARAN BLOCKBEGIN stmnt_list BLOCKEND ELSE BLOCKBEGIN
stmnt_list BLOCKEND { If ($3, List.rev($6), List.rev($10))}


iterstmnt:
   LOOP LEFTPARAN genExpr RIGHTPARAN BLOCKBEGIN stmnt_list BLOCKEND{Loop ($3, List.rev($6))}

returnstmnt:
             RETURN IDENTIFIER SEMICOLON{ Return($2)}
            | RETURN SEMICOLON {Return("")}


funcdeclstmnt:
            FUNCTION IDENTIFIER LEFTPARAN fargList RIGHTPARAN SIMPLETYPE BLOCKBEGIN stmnt_list
BLOCKEND
             {Func_decl($2, $4, $6, List.rev $8) }
```

```
fargList:  fargList COMMA farg {List.rev($3::$1)}
              | farg {[$1]}

farg:   SIMPLETYPE IDENTIFIER {Sarg($1,$2)}
            | COMPTYPE IDENTIFIER LESSTHAN SIMPLETYPE COMMA STRCONST GREATERTHAN { Carg ($1,
$2, $4,$6)}


program :
                        /* nothing */ { [] }
              | stmnt_list { List.rev $1}
```

4.1.3 AST_TablePro.mll

```
type op = Add | Sub | Mult | Div | Greater| Less | Equality| And | Or

type expr = (* Expressions *)
  Literal of float (* 1.0 *)
| IntLiteral of int (* 5 *)
| StrLiteral of string (* "PLT" *)
| Id of string (* foo *)
| Binop of expr * op * expr (* a + b *)
| Assign of string * expr (* foo = 42 *)
| IndexAsn of string * expr * expr (* a[i] = 1.0 *)
| Index of string * expr (* row[4], row[j] *)
| IndexSize of string
| IndexRange of string * expr * expr * op * expr (* for rindex ranges *)
| Call of string * expr list(* foo(1, 25) *)

type stmnt=

  SDeclstmnt of string  * string
 | CDeclstmnt of string  * string * string  * string
 | CStrdeclAsnstmnt of string * string * string * string * string list
 | CNumdeclAsnstmnt of string * string * string * string * float list
| Stmnt of expr
| Block of stmnt list
| If of expr * stmnt list * stmnt list
| Loop of expr * stmnt list
| Return of string
| Func_decl of func_decl
| GenerateStmnt of string list

and

func_decl = string * formal list * string * stmnt list

and formal = Sarg of string * string | Carg of string * string * string * string

type program = stmnt list

type retVal = F of float | S of string | SLi of string list | FLi of float list

type t = Num | Str | Col of colHeading
and
colHeading =
{
 colName: string;
 colContentType: t;
}

type var_decl =
{
```

```
              vname: string;
              vtype: t;
              vVal: retVal;
}

type functionDetail =
{
 fname : string; (* Name of the function *)
 formals : formal list; (* Formal argument names *)
 retType : t ; (* saving return type*)
 body : stmnt list;
 retValue: retVal;

}
```

4.1.4 Interpreter.ml

```
open Ast_TablePro
open Printer
open Tableutil


let rec runprogram program  sourcefile=

let rec run locals globals functions env currfname statements =

(*print_endline (string_of_bool env);
print_endline (string_of_int (List.length statements));*)
let rec eval (locals, globals, functions, env) = function
Literal(n) -> (F(n)) , (locals, globals, functions, env)
| Binop (e1 , op , e2) ->
 let val1, (locals, globals, functions, env)  =
 eval (locals, globals, functions, env) e1
in
let val2, (locals, globals, functions, env)  =
eval (locals, globals, functions, env) e2
in
let boolean i = if i then 1.0 else 0.0
in
let v =
(match (op , val1 , val2) with
               (Add , F f1 , F f2) -> F(f1 +. f2)
               | (Sub , F f1 , F f2) -> F(f1 -. f2)
               | (Mult , F f1 , F f2) -> F(f1 *. f2)
               | (Div , F f1 , F f2) ->
                 if int_of_float f2 = 0 then
                           raise (Failure("Divide by Zero"))
                 else
                           F(f1 /. f2)
               | (Greater , F f1 , F f2) -> F (boolean((f1> f2)))
               | (Less , F f1 , F f2) -> F (boolean((f1< f2)))
               | (Equality, F f1 , F f2) -> F (boolean((f1 = f2)))
               | (Or, _, _) -> raise (Failure("Or not supported"))
                | (And, _, _) -> raise (Failure("Or not supported"))
                | (_, _, _) -> raise (Failure("Multiple binary combination
  or identifier not initialized"))

)

              in v , (locals, globals, functions, env)

| IndexRange (col, ste, ene , op, e1) ->

let st =
   let evst,(locals, globals, functions, env) =
   eval(locals, globals, functions, env)  ste in
   (match evst with F f -> int_of_float f
   | _ -> raise(Failure("Range start index
   expression not a number")) )
in
let en =
   let even,(locals, globals, functions, env) =
   eval(locals, globals, functions, env)  ene in
   (match even with F f -> int_of_float f
| _ -> raise(Failure("Range end index
expression not a number")) )

in
let rhsVal, (locals, globals, functions, env) =
eval (locals, globals, functions, env) e1
in
      let rec performloop  currIndex
      looplocals loopglobals =
              let assign   aindex =
                 let declList =
                 if (NameMap.mem col looplocals) then
                 (NameMap.find col looplocals)
                 else if (NameMap.mem col loopglobals)
                 then (NameMap.find col loopglobals)
                 else raise (Failure ("Id not found"))
                            in
                            let colList = declList.vVal in
               (match colList with
                FLi fl -> let res =
              (match rhsVal with
                          F f ->
                          let colArray =
                          Array.of_list fl
                                in
                                let val1 =
                                F (colArray.(aindex)) in
                                       let valAssign =
                                       (match
                                       (op , val1 , rhsVal) with
                                                    (Add , F f1 , F f2) -> (f1 +. f2)
                                                    | (Sub , F f1 , F f2) -> (f1 -. f2)
                                                    | (Mult , F f1 , F f2) -> (f1 *. f2)
                                                    | (Div , F f1 , F f2) ->
                                                    if int_of_float f2 = 0 then
                                                                raise (Failure("Divide by Zero"))
                                                      else
                          (f1 /. f2)
                          | (_, _, _) ->
                          raise (Failure("Binop combination not supported"))
             )
             in
             let replaceList  = ignore(colArray.(aindex) <- valAssign);
             FLi(Array.to_list  colArray)
             in
               if (NameMap.mem col looplocals) then
                            let localsadded = NameMap.add
                            col ({vname= declList.vname ;vtype=
                            declList.vtype; vVal= replaceList}) looplocals
                            in (localsadded , loopglobals)
               else if (NameMap.mem col loopglobals) then
                            let globalsadded =
                            NameMap.add col
                            ({vname= declList.vname ;vtype= declList.vtype;
                            vVal= replaceList}) loopglobals
                            in (looplocals , globalsadded)
               else raise (Failure ("undeclared identifier"))
|              _ -> raise ( Failure ("Incompatible assignment for index range"))
     )
    in res
              | _ -> raise (Failure ("Not the right type"))
              )
              in

                if currIndex <= en then
                        let  (al, ag) = assign currIndex
                        in let currIndex = currIndex+1
                        in performloop  currIndex al ag
                else
                        (looplocals, loopglobals)
               in
                     let (l, g) = performloop st locals globals
                     in rhsVal, (l, g, functions, env)




    | IndexAsn (id, e1,e2) ->
                     let ev,(locals, globals, functions, env)=
                     eval(locals, globals, functions, env)  e2
                      in
                      let ev1,(locals, globals, functions, env) =
                      eval(locals, globals, functions, env)  e1
                      in
                      let v1 =
                       (match ev1 with F f -> int_of_float f
                                | _ -> raise(Failure("Index expression not a number"))
                        )
                      in
                       let declList =
                         if (NameMap.mem id locals) then (NameMap.find id locals)
                         else if (NameMap.mem id globals) then (NameMap.find id globals)
                         else raise (Failure ("Id not found"))
                        in
                      let colList = declList.vVal
                      in
                      let colListlen =
                      (match colList with
                              FLi fl->List.length fl
                              | SLi sl ->List.length sl
                              | _ ->
                              raise (Failure ("Cannot determine length of this retVal"))
                      )
                      in
                      let appendNew =
                     let newCollist =
                          (match colList with
                              FLi fl ->(match ev with
                                              F f -> FLi(fl @ [f])
                                              | _ -> raise (Failure
                                              ("Incomaptible assignment"))
                                         )
                              | SLi sl ->(match ev with
                                              S s -> SLi(sl @ [ s ])
                                              | _ -> raise (Failure
                                              ("Incomaptible assignment"))
                                          )

                              | _ -> raise (Failure ("Not the right type"))
                              )in
                       if (NameMap.mem id locals) then
                       let localsadded =
                       NameMap.add id
                       ({vname= declList.vname ;vtype=
                       declList.vtype; vVal= newCollist})
                       locals
                       in (localsadded , globals)
                       else if (NameMap.mem id globals) then
                       let globalsadded =
                       NameMap.add id
                       ({vname= declList.vname ;vtype=
                       declList.vtype; vVal= newCollist})
                       globals
                       in (locals , globalsadded)
                       else raise (Failure ("Id not found"))
              in
             let (new_locals, new_globals) =
              if v1 = colListlen then
                appendNew

              else if v1 < colListlen then
              let assign  = (match colList with
                     FLi fl ->
                     let res =
                      (match ev with
                              F f -> let colArray =Array.of_list fl
                                  in
                                      let replaceList  =
                                      ignore(colArray.(v1) <- f);
                                      FLi(Array.to_list  colArray)
                                      in
                                         if (NameMap.mem id locals) then
                                         let localsadded =
                                         NameMap.add id
                                         ({vname= declList.vname ;
                                         vtype= declList.vtype;
                                         vVal= replaceList}) locals
                                                 in (localsadded , globals)
                                         else if (NameMap.mem id globals) then
                                                 let globalsadded = NameMap.add id
                                                 ({vname= declList.vname ;
                                                 vtype= declList.vtype;
                                                 vVal= replaceList}) globals
                                                 in (locals , globalsadded)
                                         else raise (Failure ("Id not found"))
             |   _ -> raise ( Failure ("Incompatible assignment for index range"))
     )
    in res
             | SLi sl -> let res =
              (match ev with
                     S s -> let colArray =Array.of_list sl
                          in
                              let replaceList  = ignore(colArray.(v1) <- s);
                              SLi(Array.to_list  colArray)
                              in
                               if (NameMap.mem id locals) then
                               let localsadded =
                               NameMap.add id
                               ({vname= declList.vname ;
                               vtype= declList.vtype; vVal= replaceList}) locals
                               in (localsadded , globals)
                               else if (NameMap.mem id globals)
                               then
                               let globalsadded =
                               NameMap.add id
                               ({vname= declList.vname ;
                               vtype= declList.vtype; vVal= replaceList}) globals
                               in (locals , globalsadded)
```

```
                                        else raise (Failure ("Id not found"))
                              |  _ ->
                              raise ( Failure ("Incompatible assignment for index range"))
                )
                  in res


                | _ -> raise (Failure ("Not the right type"))
   )
                   in  assign
else raise (Failure ("index id out of range" ))
in ev, (new_locals, new_globals, functions, env)

| Assign(id, e1) ->
              let ev,(locals, globals, functions, env)=
              eval(locals, globals, functions, env)  e1
              in
              if NameMap.mem id locals then
              let declis = (NameMap.find id locals)
              in
              (match declis.vtype with
              Num -> (match ev with
              F f ->
                     ev, (  (NameMap.add id ({vname= declis.vname ;
                     vtype= declis.vtype; vVal= ev}) locals ),
                     globals, functions, env)
              | _ -> raise (Failure (" Types not compatible
                            in assignment,Expected Number type"))
              )
              | Str ->(match ev with
              S s -> ev, ((NameMap.add id ({vname=
              declis.vname ;vtype= declis.vtype; vVal= ev}) locals),
              globals, functions, env)
              | _ -> raise (Failure
              (" Types not compatible in assignment, Expected String type"))
              )
              | Col c ->( match c.colContentType with
                 Num -> (match ev with
                                          FLi fl -> ev,
                                          ((NameMap.add id
                                          ({vname= declis.vname ;
                                          vtype= declis.vtype; vVal= ev})
                                          locals), globals, functions, env)
                                          | _ -> raise
                                          (Failure (" Types not
                                          compatible in assignment,
                                          Expected Number column"))
                                    )
   | Str -> (match ev with
                              SLi sl -> ev,
                              ((NameMap.add id ({vname= declis.vname ;
                              vtype= declis.vtype; vVal= ev})
                              locals), globals, functions, env)
                              | _ -> raise (Failure (" Types not compatible in assignment,
                              Expected String column"))
                 )
   | _ -> raise (Failure (" Types not compatible in assignment"))

  )
)


else if NameMap.mem id globals then
              let declis = (NameMap.find id globals)
              in
              (match declis.vtype with
Num -> (match ev with
              F f -> ev, (locals , (NameMap.add id
              ({vname= declis.vname ;vtype= declis.vtype;
              vVal= ev}) globals),
              functions, env)
              | _ -> raise (Failure ("
              Types not compatible in assignment,
              expected number type"))
              )
| Str ->(match ev with
              S s -> ev, (locals , (NameMap.add id
              ({vname= declis.vname ;vtype= declis.vtype;
              vVal= ev}) globals),
              functions, env)
              | _ -> raise (Failure (" Types not compatible
              in assignment,expected string type"))
              )
| Col c ->( match c.colContentType with
                 Num -> (match ev with
                               FLi fl -> ev, (locals ,
                               (NameMap.add id
                               ({vname= declis.vname ;vtype= declis.vtype; vVal= ev})
                               globals), functions, env)
                               | _ -> raise (Failure
                               (" Types not compatible in assignment,expected number
                               column"))
                         )
                 | Str -> (match ev with
                               SLi sl -> ev, (locals ,
                               (NameMap.add id
                               ({vname= declis.vname ;vtype= declis.vtype; vVal= ev})
                               globals), functions, env)
                               | _ -> raise
                               (Failure (" Types not compatible in assignment,expected
                               string column"))
                         )
                 | _ -> raise (Failure (" Types not compatible in assignment"))

                )

)
else raise (Failure("Identfier not declared" ^id ))

| IntLiteral (e1) -> F(float_of_int e1) ,(locals, globals, functions, env)

| Id (id) ->

let declList =
if (NameMap.mem id locals) then (NameMap.find id locals)
else if (NameMap.mem id globals) then (NameMap.find id globals)
else raise (Failure ("Id not declared"))
in
declList.vVal , (locals, globals, functions, env)

| StrLiteral (e1) -> S(e1) ,(locals, globals, functions, env)
| Index (id, e1) ->  let ev,(locals, globals, functions, env)
= eval(locals, globals, functions, env)  e1
   in
              let mval =  (match ev with
                                F f -> int_of_float f
                                | _  -> raise (Failure ("Index is not a number" ))
                                )
                          in
                          let declist =
                                        if NameMap.mem id locals then
```

```
                                                (NameMap.find id locals)
                                         else if NameMap.mem id globals then
                                                (NameMap.find id globals)
                                                else raise (Failure ("column index not found" ))
                                      in
                                      let v =
                                      (match (declist.vVal) with
                                       FLi f1 -> let arr = Array.of_list f1
                                                  in F(arr.(mval))

                                       |SLi s1 -> let arr = Array.of_list s1
                                                  in S(arr.(mval))

                                       | _ -> raise (Failure ("Not a column type"))

                                      )

               in v,(locals, globals, functions, env)
        | IndexSize (id) ->
          let declList =
                    if (NameMap.mem id locals) then (NameMap.find id locals)
                    else if (NameMap.mem id globals) then (NameMap.find id globals)
                    else raise (Failure ("Id not declared"))
                    in
                      let colList = declList.vVal
                      in
                      let colListlen =
                                      (match colList with
                                                FLi fl->List.length fl
                                                | SLi sl ->List.length sl
                                                | _ -> raise (Failure
                                                ("Cannot determine length of this retVal"))
                                                )
                              in
                      F(float_of_int colListlen),(locals, globals, functions, env)
        | Call(f, elist) ->
                     let fdecl =
                    (*ignore(print_string "Inside Call");
                    ignore(Printer.print_functions functions);
                    ignore(print_string f);*)
                    try NameMap.find f functions
                    with Not_found ->
                    raise (Failure ("undefined function " ^ f))
                    in
                    let slist = fdecl.body
                     in
                        let fargtype formal =
                         ( match formal with
                              Sarg(x, y) -> vdecl_from_str (x)
                              | Carg (p,q,r,s)->
                              let conttype = vdecl_from_str (r)
                              in
                                 Col ({colName=s; colContentType=conttype})
                         )
                        in
                        let fargname formal =
                         (match formal with
                              Sarg(x, y) ->  (y)
                              | Carg(p, q, r, s) ->  (q)
                         )
                        in
                        let evalex ex =
                               let evalres ,
                               (locals , globals, functions, env) =
                               eval (locals, globals, functions, env) ex
                               in evalres
                    in
                    let fargdeclList =
                                  let fargdecl l e f =
                                           ({vname=(fargname f);
                                           vtype=(fargtype f);vVal= evalex e})::l
                           in List.fold_left2 fargdecl [] elist fdecl.formals
                     in
                     let initialfunclocals =
                              let oneformaladdedmap  argsmap fargdecl=
                                NameMap.add fargdecl.vname fargdecl argsmap
                           in
                                  try List.fold_left
                                  oneformaladdedmap NameMap.empty  fargdeclList
                                  with Invalid_argument(_) ->
                                  raise (Failure
                                  ("wrong number of arguments passed to " ^ fdecl.fname))

                           in
                           let (_, globals, functions, _) =
                           run initialfunclocals globals functions true f slist
                           in
                           let fd = NameMap.find f functions
                            in   fd.retValue ,(locals, globals, functions, env)
in
let rec  exec_stmnt (locals, globals,
functions, env, currfname) = function

  Block(slist) ->  List.fold_left exec_stmnt (
  locals, globals, functions, env, currfname) slist
| SDeclstmnt(s1 ,s2) -> (* populate symbol table *)

if( env ) then
             let new_locals =
             if NameMap.mem s2 locals then raise (Failure "Redefined Identifier")
             else NameMap.add s2 ({vname=s2 ; vtype= (vdecl_from_str s1);
             vVal=S("0")}) locals
             in  (new_locals, globals, functions, env, currfname)
else
             let new_globals =
             if NameMap.mem s2 globals then raise (Failure "Redefined Identifier")
             else NameMap.add s2 ({vname=s2 ; vtype=(vdecl_from_str s1);
             vVal=S("0")})  globals
             in  (  locals, new_globals, functions, env, currfname)

| CDeclstmnt(s1 ,s2, s3, s4) ->(* populate symbol table *)
             (*print_string "CDeclStmnt in "; print_endline(string_of_bool env );
             print_locals locals ; print_globals globals;*)
             let conttype = vdecl_from_str s2
             in
             let contLi =
             (match conttype with
              Str -> SLi([])
              | Num -> FLi([])
              | _ -> raise (Failure ("Content type not supported"))
             )
             in
             if( env ) then
                           let new_locals =
                           if NameMap.mem s4 locals then raise
                           (Failure "Redefined Identifier")
                           else NameMap.add s4
                           ({vname=s4 ; vtype= (Col
                           ({colName=s3; colContentType= conttype}));
                           vVal=contLi}) locals
                           in  (new_locals, globals,
                           functions, env, currfname)
```

```
            else
                        let new_globals =
                        if NameMap.mem s4 globals then raise
                        (Failure "Redefined Identifier")
                        else NameMap.add s4 ({vname=s4 ;
                        vtype=
                        (Col ({colName=s3; colContentType= conttype}));  vVal=contLi})
                        globals
                        in  (  locals, new_globals, functions, env, currfname)

| CStrdeclAsnstmnt(s1,s2,s3,s4,l) ->
            if ((vdecl_from_str s2) != Str )
            then raise (Failure ("String list assigned
            to non str content column" ))
            else
            if( env ) then
            let new_locals =
            if NameMap.mem s4 locals then raise
            (Failure "Redefined Identifier")
            else NameMap.add s4 ({vname=s4 ;
            vtype= (Col ({colName=s3;
            colContentType= (vdecl_from_str s2)}));
            vVal=SLi(l)}) locals
            in  (new_locals, globals, functions, env, currfname)
            else
            let new_globals =
            if NameMap.mem s4 globals then raise (Failure "Redefined Identifier")
                        else
                           NameMap.add s4
                           ({vname=s4 ; vtype=
                           (Col ({colName=s3; colContentType=
                           (vdecl_from_str s2)}));  vVal=SLi(l)})  globals
                        in  (  locals, new_globals, functions, env, currfname)

| CNumdeclAsnstmnt(s1,s2,s3,s4,l) ->
            if ((vdecl_from_str s2) != Num )
            then raise (Failure ("String list assigned to non str content column" ))
            else
              if( env ) then
                        let new_locals =
                        if NameMap.mem s4
                        locals then raise
                        (Failure "Redefined Identifier")
                        else NameMap.add s4
                        ({vname=s4 ; vtype=
                        (Col ({colName=s3; colContentType=
                        (vdecl_from_str s2)}));  vVal=FLi(l)}) locals
                        in  (new_locals, globals, functions, env, currfname)
               else
                        let new_globals =
                        if NameMap.mem s4 globals then raise
                        (Failure "Redefined Identifier")
                        else NameMap.add s4
                        ({vname=s4 ; vtype= (Col ({colName=s3;
                        colContentType= (vdecl_from_str s2)}));  vVal=FLi(l)})
                        globals
                        in  (  locals, new_globals, functions, env, currfname)

|   Func_decl (name, flist, rettype, slist) ->
            let retVal =
            (match rettype with
              Sret(s) -> let ret_t = vdecl_from_str s in
                (match ret_t with
                        Str -> S("")
                        | Num -> F(0.0)
                        | _ -> raise (Failure ("Return type not recognized"))
                )
              | Cret(c, content_s) -> let content_t = vdecl_from_str content_s in
                        (match content_t with
                        Str -> SLi([])
                        | Num -> FLi([])
                        | _ -> raise (Failure ("Return type not recognized"))
                        )

            )
            in
            let ret_t  =
            (match rettype with
              Sret(s) -> vdecl_from_str s
              | Cret(c, content_s) -> Col ({colName ="" ;
              colContentType=vdecl_from_str content_s })
            )
            in
            let fdecl = {fname = name; formals = flist; retType =
            ret_t; body = slist; retValue=retVal}
            in
            let new_functions =
                        if NameMap.mem name functions then raise
                        (Failure "Redefined function name")
                        else NameMap.add name fdecl functions
                        in (*ignore(Printer.print_functions new_functions);*)
                        ( locals, globals, new_functions, env, currfname)

|   Stmnt(e) ->
                        let v, (locals, globals, functions, env) =
                        eval (locals, globals, functions, env)  e
                        in  (locals, globals, functions, env, currfname)



| Loop (e, slist) ->

    let rec loop locals globals functions env currfname slist e =
    let  v, (locals, globals, functions, env)  =
    eval (locals, globals, functions, env) e in

    (match v with
   F f  ->     if f <> 0.0 then
                  let ( locals, globals, functions, env, currfname)  =
                  List.fold_left exec_stmnt
                  ( locals, globals, functions, env , currfname) slist
                  in  loop  locals globals functions env currfname slist e
                        else ( locals, globals, functions, env, currfname)

   | _ ->
            raise (Failure ("Loop Expression not evaluating to number "))
     )
     in loop locals globals functions env  currfname slist e


| Return (s) ->
if (s != "") then
let retValue =
            if NameMap.mem s locals then
                        (NameMap.find s locals).vVal
            else if NameMap.mem s globals then
                        (NameMap.find s globals).vVal
            else raise (Failure ("undeclared identifier " ^ s))
                        in
                                    let ofdecl = NameMap.find currfname functions
                                    in
                                    let new_functions =
                                        NameMap.add ofdecl.fname
                                        ({fname = ofdecl.fname;
                                        formals = ofdecl.formals;
                                        retType = ofdecl.retType;
                                        body = ofdecl.body; retValue=retValue;}) functions
                                        in
                                        ( locals, globals, new_functions, env, currfname)
else
let retValue = S("")
                        in
                        let ofdecl = NameMap.find currfname functions
                        in
                        let new_functions = NameMap.add ofdecl.fname
                        ({fname = ofdecl.fname;
                        formals = ofdecl.formals;
                        retType = ofdecl.retType;
                        body = ofdecl.body; retValue=retValue;}) functions
                        in
                                        ( locals, globals, new_functions, env, currfname)



| If (e, slist1, slist2) ->

   let v, (locals, globals, functions, env) =
   eval (locals, globals, functions, env) e in

   (match (v) with
              ( F f ) ->
                        if f <> 0.0 then
                        (List.fold_left exec_stmnt
                        ( locals, globals, functions, env, currfname) slist1 )
                        else (List.fold_left exec_stmnt
                        ( locals, globals, functions, env, currfname) slist2)
              | _ ->
                        raise (Failure
                        ("If expression not evaluating to number "))


    )

| GenerateStmnt (slist) ->
if (String.compare "main" currfname  != 0) then
raise (Failure ("Generate Statement  allowed in main only" ))
else
let checkColType s =
            if ((NameMap.mem s globals)!=true) then
                        false
            else if ( (NameMap.find s globals).vtype
            != Num) then
              let b =
                        if( (NameMap.find s globals).vtype != Str) then
                        true
                        else raise (Failure
                        ("Cannot generate table with Num/str type identifier " ^ s))
              in b
            else raise (Failure ("Cannot
            generate table with Num/str type identifier " ^ s))
in
   let checked s = checkColType s
   in
            let pickUpColList s =
                        if checked s then
                        let list =
                           (match (NameMap.find s globals).vVal
                           with
```

```
                    SLi sl -> (*print_string "no converting";
                    List.iter print_string sl;*)
                            let li =
                            ( match (NameMap.find s globals).vtype with
                                        Col c -> (c.colName)::sl
                                        | _ -> raise (Failure
                                        ("Cannot generate
                                        table with str type identifier " ^ s))
                            )
                            in li
                    | FLi fl ->
                            let li =
                            ( match (NameMap.find s globals).vtype with
                            Col c -> c.colName::
                            (Tableutil.convertFloatListToStringList fl)
                            | _ -> raise (Failure
                            ("Cannot generate table with Num type identifier " ^ s))
                            )
                            in li

                    | _ -> raise (Failure( "Not a list ")))
                 in list
                 else raise
                 (Failure ("undeclared identifier list ending with " ^ s))

           in
             let addList outerList s =
             (pickUpColList s)::outerList
             in
                       let lists = List.fold_left addList [] slist
                       in
                               let paddedlists =
                               Tableutil.padLists lists
                               in
                       let x =
                       Tableutil.metafoo (List.rev paddedlists) sourcefile
                       in ( locals, globals, functions, env, currfname)


in
let ( locals ,globals, functions, env, currfname) =
List.fold_left exec_stmnt
( locals ,globals, functions, env, currfname) statements

in ( locals ,globals, functions, env)

in
let ( locals ,globals, functions, env) =
try run NameMap.empty NameMap.empty NameMap.empty false "main" program
with Not_found -> raise (Failure ("Could not run Program"))
in ( locals ,globals, functions, env);
```

4.1.5 Printer.ml

```
open Ast_TablePro

module NameMap = Map.Make(struct
  type t = string
  let compare x y = Pervasives.compare x y
  end)

let rec string_of_expr = function
    Literal(l) -> string_of_float l
  | IntLiteral(l) -> string_of_int l
  | StrLiteral(s) -> s
  | Id(s) -> s
  | IndexSize(s) -> s
  | Binop(e1, o, e2) -> string_of_expr e1 ^ " " ^(match o with Add -> " + " ^ string_of_expr e2 | Sub -> "-"  ^
string_of_expr e2| Mult -> "*" ^ string_of_expr e2 | Div -> "/" ^ string_of_expr e2
  | Equality ->  " == " ^ string_of_expr e2 | And -> " Log && " ^ string_of_expr e2 | Less -> "<"^ string_of_expr e2  |
Greater -> ">" ^ string_of_expr e2 | Or -> " Log Or "^ string_of_expr e2 )
  | Assign(v, e) -> v ^ " = " ^ string_of_expr e
  | IndexAsn ( s, e1 ,e2) -> s ^ "[" ^ string_of_expr e1  ^ " ] = " ^ string_of_expr e2
  | Call(f, el) -> f ^ "(" ^ String.concat ", " (List.map string_of_expr el) ^ ")"
  | Index(s, e) -> s ^ "[" ^ string_of_expr e ^ "]"
  | IndexRange(s, i, j, op, e) -> s ^ "[" ^ string_of_expr i ^ "-" ^ string_of_expr j ^ "]" ^
                                    (match op with Add -> " + " ^ string_of_expr e | Sub -> "-"  ^ string_of_expr e
                                    | Mult -> "*" ^ string_of_expr e | Div -> "/" ^ string_of_expr e  | _ ->"Op not supported")



let rec string_of_formal  = function
Sarg (s1 , s2) -> s1 ^" * "^ s2
| Carg (s1 , s2, s3, s4) -> s1 ^" * "^ s2  ^  s3 ^" * "^ s4

let rec string_of_formals list = "" ^ String.concat "" (List.map string_of_formal list)


let rec string_of_stmt = function
    SDeclstmnt (s1 , s2) -> "Declared " ^ s1 ^ s2
  |  CDeclstmnt (s1 , s2, s3, s4) -> "Declared " ^ s1 ^ s2 ^ s3 ^ s4
  |  CNumdeclAsnstmnt ( s1, s2, s3, s4, l) -> "Declared " ^ s1 ^ s2 ^ s3 ^ s4
  |  CStrdeclAsnstmnt ( s1, s2, s3, s4, l) -> "Declared " ^ s1 ^ s2 ^ s3 ^ s4
  |  Block(stmts) -> "{\n" ^ String.concat "" (List.map string_of_stmt stmts) ^ "}\n"
  | Stmnt(expr) -> string_of_expr expr ^ ";\n";
  | Return(s) -> "return " ^ s ^ "\n";
  | If(e, slist1 , slist2) -> "if (" ^ String.concat "" (List.map string_of_stmt slist1) ^ String.concat "" (List.map
string_of_stmt slist2)
  | Loop(e1, slist) -> "loop " ^ string_of_expr e1  ^ " Loop begin " ^ String.concat "" (List.map string_of_stmt slist)  ^ "
Loop end "
  | GenerateStmnt(slist) -> "Generate table for " ^ (List.fold_left (fun s1 s2 -> s1 ^ s2)  ""  slist)
  | Func_decl(s1, list1 , s2, list2) -> string_of_fdecl s1 list1 s2 list2

              and
               string_of_fdecl  s1  list1  s2 list2 =
                s1 ^ s2 ^ "(" ^ string_of_formals list1^ ")" ^ "\n{\n" ^
                String.concat "" (List.map string_of_stmt list2) ^  "}\n"



let string_of_program (slist) = print_string "in printer\n";
  String.concat "" (List.map string_of_stmt slist) ^ "\n"


let rec string_of_col_type t =
              (match t with Num -> " num "
              | Str -> "str"
              | _ -> raise (Failure ("unsupported column type " )))

let string_of_Col c = "Col Of"^ c.colName^(string_of_col_type c.colContentType)


let rec string_of_vdecl_type t =
              (match t with Num -> " num "
              | Str -> "str"  | Col  c -> string_of_Col c)




let rec string_of_vdecl  s1  t  =
                              " Name: " ^ s1 ^ " Type: " ^ (string_of_vdecl_type t)

let vdecl_from_str s =
              (match s with "num:" -> Num
                                | "str:"-> Str
                                | _     -> raise (Failure ("undefined type " ^ s)))

let string_of_retval retvalue =
              (match retvalue with
                          F f -> ("float of " ^ string_of_float f )
                              | S s -> ("string of" ^ s)
                              | _-> "A list of values")



let rec print_locals locals  =
   let f k e =  print_endline ("Key: " ^ k ^ "Vdecl: "^ (string_of_vdecl e.vname e.vtype ) ^ string_of_retval e.vVal)
      in
              print_endline "Locals"; NameMap.iter f locals

(* let print_globals (globals) =
    let f k e =  print_endline ("Key: " ^ k ^ "Type: "^ e)
       in
              print_endline "globals"; NameMap.iter f globals *)

let rec print_globals globals  =
   let f k e =  print_endline ("Key: " ^ k ^ "Vdecl: "^ (string_of_vdecl e.vname e.vtype ))
      in
              print_endline "Globals"; NameMap.iter f globals


let rec string_of_fdecl  s1  list1  s2 list2  =
                              " Name: "^ s1 ^ " RetType: " ^ s2 ^ "(" ^ string_of_formals list1^ ")" ^ "\n{\n" ^
                              String.concat "" (List.map string_of_stmt list2) ^  "}\n "

let rec string_of_function  k e =
                            let a = string_of_fdecl e.fname e.formals (string_of_vdecl_type e.retType) e.body
                            in
                                          k ^ a

  let print_functions (functions) =
    let f k e =  print_endline (string_of_function k e)
              in
                            NameMap.iter  f functions
```

4.1.6 TableUtil.ml

```
open Printf

let file = "Tabout.csv"
let oc = open_out file

let rec convertFloatListToStringList l =
                    List.rev(List.fold_left (fun li e -> (string_of_float e)::li) [] l )

let rec printheads l  oc=
                    match l with
                    [] -> ()
                    | [a] -> print_string ((List.hd a) ^ "      ");fprintf oc "%s\n" (List.hd a);()
                    | hd::tl -> print_string ((List.hd hd) ^ "     "); fprintf oc "%s\t" (List.hd hd) ;printheads tl oc

let rec removeTop l =
          match l with
                    [] -> []
                    | [a] -> (match a with
                               [] -> []
                              | [b]-> []
                              | hd1::tl1 ->  [List.tl a]
                              )
                    | hd::tl ->  (match hd with
                                        [] -> []
                                        | [a]-> []
                                        | hd1::tl1 -> List.tl hd::(removeTop tl)
                                 )

let rec metafoo l sourcefile= print_endline "";
          let filename = sourcefile ^ ".csv";
                              in
                              let file = filename
                              in
                              let oc = open_out file
                    in
          let rec metafoo1 l=

          match l with
          [] -> ()
          | hd::tl ->  printheads l oc; metafoo1 (print_endline"";removeTop l);()
          in
          match l with
          [] -> ()
          | hd::tl ->  printheads l oc; metafoo1 (print_endline"";removeTop l);()

let rec maxlength l =
          let getMaxlength n li =
                    if List.length li > n  then List.length li
                    else  n;
          in List.fold_left getMaxlength 0 l


let rec padLists l =
          let maxl = maxlength l
          in
          let rec padonelist  li =   (*print_string "Max"; print_int maxl;*)
             if List.length li = maxl then li
             else if List.length li < maxl then padonelist (li@["nodata"])
             else raise (Failure ("maxlength not working"))
             in
             let foldfunc     retLi li=
                          let ret = (padonelist li)::retLi
                          in ret
             in
                          List.fold_left foldfunc []  l
```

4.1.7 Tabpro.ml

```
open Printer

 let _ =  if ((Array.length Sys.argv) = 1 )
         then let _ = print_string "Source File not provided"
 in ();
         else if ((Array.length Sys.argv) > 2 )
         then
         let _ = print_string "Only one source file as an argument is supported."
 in ();
         else let sourceFile = Sys.argv.(1)
         in
           let inChannel = open_in sourceFile
           in
              let lexbuf = Lexing.from_channel inChannel
              in
                let len = String.length sourceFile
                in
                  let tlen = len - 4
                  in
                    let substr = String.sub sourceFile 0 tlen
                    in
                      let strfile  = substr
                      in
                        let program = Parser_TablePro.program Scanner_TablePro.token lexbuf in ignore
(Interpreter.runprogram program                strfile)
```

4.2 Code Listing for the Preprocessor

4.2.1 Scanner_TableProPreProcess.mly

```
{
open Parser_TableProPreProcess

}

let includename = "include"
let semicolon = [';']
let alphanum = ['a'-'z' 'A'-'Z' '0'-'9']
let alpha = ['a'-'z' 'A'-'Z' '.']

rule token =
parse    [' ' '\t' '\r' '\n']  { token lexbuf }
        |  includename  { INCLUDE  }
        | semicolon { SEMICOLON }
        |  alpha (alphanum)* as name {IDENTIFIER(name)}
        | _ { TEXT }
        | eof { EOF}


{

}
```

4.2.2 Scanner_TableProPreProcess.mly

```
%{   %}
%token  EOF
%token  SEMICOLON
%token  INCLUDE
%token  TEXT
%token  <string>IDENTIFIER


%start program
%type <string> program
%%

program  : INCLUDE IDENTIFIER SEMICOLON { $2 }
```

4.2.3 PreProcessor.ml

```
let rec includemylib includefile  sourceFile  destFile =

 let oc = open_out destFile
 in
        let copy src  =
                                   let ic = open_in src in
                                     try
                                       while true do
                                         let s = input_line ic in
                                         output_string oc s;
                                         output_char oc '\n';
                                       done
                                     with End_of_file ->
                                       close_in ic;
        in
        let _ = copy includefile; copy  sourceFile; close_out oc;
        in ();
```

4.2.4 TabProPreProcess.ml

```
open Printer

 let _ =  if ((Array.length Sys.argv) = 1 )
          then let _ = print_string "Source File not provided"
 in ();
          else if ((Array.length Sys.argv) > 2 ) then
          let _ = print_string "Only one source file as an argument is supported."
 in ();
          else let sourceFile = Sys.argv.(1)
          in
            let inChannel = open_in sourceFile
            in
              let lexbuf = Lexing.from_channel inChannel
              in
```

```
let includename = Parser_TableProPreProcess.program Scanner_TableProPreProcess.token lexbuf
in
   let preprocessOut = sourceFile ^ "PP"
   in ignore(print_string preprocessOut );ignore(print_string includename );
   ignore (PreProcessor.includemylib includename  sourceFile preprocessOut)
```

4.3 *Other Test Code*

Code for testing the Parser:

```
open Ast_TablePro
open Printer
let _ =
let lexbuf = Lexing.from_channel stdin in
let program = Parser_TablePro.program Scanner_TablePro.token lexbuf in
let listing = Printer.string_of_program program in
print_string listing
```

ParserTestBuilder.bat

```
ocamlc ..\..\code\Ast_TablePro.mli
ocamlc -c -I ..\..\code -dllpath ..\..\code  ..\..\code\printer.ml
ocamlyacc ..\..\code\Parser_TablePro.mly
ocamlc -c -I ..\..\code -dllpath ..\..\code ..\..\code\Parser_TablePro.mli
ocamlc -c -I ..\..\code -dllpath ..\..\code ..\..\code\Parser_TablePro.ml
ocamllex  -o ..\..\code\Scanner_TablePro.ml ..\..\code\Scanner_TablePro.mll
ocamlc -c  -I ..\..\code -dllpath ..\..\code ..\..\code\Scanner_TablePro.ml

ocamlc -c -I ..\..\code -dllpath ..\..\code Parser_Test.ml
ocamlc -o Parser_Test  -I ..\..\code -dllpath ..\..\code Scanner_TablePro.cmo  Parser_TablePro.cmo printer.cmo
Parser_Test.cmo
```

4.4 *Regression test script*

TabProRegTest.bat

```
ocamlrun ..\bin\TabPro basketball.tab
ocamlrun ..\bin\TabPro marks.tab
ocamlrun ..\bin\TabPro looptest.tab
ocamlrun ..\bin\TabPro iftest.tab
ocamlrun ..\bin\TabPro symbolTable.tab
ocamlrun ..\bin\TabPro indextest.tab
ocamlrun ..\bin\TabPro indexAsign.tab
ocamlrun ..\bin\TabPro hello.tab
ocamlrun ..\bin\TabPro distance.tab
ocamlrun ..\bin\TabPro returnstmnt.tab

..\bin\windiff .\*.csv .\regresults\*.csv -D -S .\regresults.txt
```

regreults.txt (File generated by windiff showing no changes in the output)

```
-- C:\Users\anu\Desktop\ColDocs\PLT\Tabl_Pro\svn_root\tests\*.csv :
C:\Users\anu\Desktop\ColDocs\PLT\Tabl_Pro\svn_root\tests\regresults\*.csv -- includes left-only,differing files
-- 0 files listed
```

4.5 Test Cases

1. Demo.tab

```
function average(num:x, num:y) num:
{
num:sum;
sum = x + y;
avg=sum/2;
return ret;
}

function square(num:x) num:
{
num:ret;
ret= x * x;
return ret;
}

function power(num:x, num:y) num:
{
num:ret;
ret = 0.0;
num:i;

if(y == 1.0)
{
ret= x;
}

else
{
        loop(i<y)
        {
        ret = ret + x;
        }
}
return ret;
}


function colaverage(col:col1<num:,"c1">, col:col2<num:,"c2"> )col:<num:>
{
        col:cavg<num:,"average">;
        num: length1;
        num: x;
        num: y;
        num: sum;
        x = 0;
        sum = 0;
        y = 0;
        length1 = c1.size;
        x = 0;
        loop(x< length1){
        y = col1[x]+col2[x];
        sum = sum + y;
        cavg[x] = sum;
        x = x+1;
        sum = 0;
        }
        y = length1 -1;
        cavg[0-y] /= 2;
        return cavg;
}
include mylib;
num: i;
num: j;
num: k;
num: k1;
num: first;
num: second;
num: k2;
num: k3;
num: colsize;

col:c0<str:,"TeamNames">={"NewYorkKnicks" , "TeamMiamiHeat", "_ChicagoBulls", "DetroitPistons",
"HoustonRockets"};
col:c1<num:,"Score1">={ 20.0 , 30.0, 25.0, 33.0, 42.0};
col:c2<num:,"Score2">={ 40.0 , 34.0, 22.0, 44.0, 28.0};
col:ctotal<num:,"Sum">;
col:cbonus<num:,"Bonus">;
col:caverage<num:,"Avg">;
col:cwinner<str:,"Ranks">={ "None","None","None","None","None"};

colsize = c0.size;
function FindWinner(num: b) num:
{
first = 0;
second = 0;
k = 0;
  loop(k<3){
  b = 0;
  j  = b+1;
              loop(j < colsize){
                             if(ctotal[b]<ctotal[j]){
                                          b = b+1;
                                          j = b+1;
                               }
                             else{
                             j = j+1;
                             }
               }
               k = k+1;
               if(first == 0){
                            cwinner[b] = "First";
                            first = ctotal[b];
                            ctotal[b] = 0;
                            k1 = b;
               }
               else{
                            if(second == 0){
                            cwinner[b] = "Second";
                            k2 = b;
                            second = ctotal[b];
                            ctotal[b] = 0;
                            }
                            else{
                             cwinner[b] = "Third";
                             k3 = b;
                            }

               }

}

                             ctotal[k1] = first;
                             ctotal[k2] = second;
}
```

```
function bonusfun(num: bonus) num:{
            cbonus[k1-k1] += 20 ;
            cbonus[k2-k2] += 10;
            cbonus[k3-k3] += 5;
}

function bonusfun1(col:cpbonus<num:,"PBonus">)col:<num:>{
            num: x ;
            x = 0;

            loop (x < 5)
            {
             j = ctotal[x];
              cpbonus[x] = j;
              x = x+1;
            }
            cpbonus[k1-k1] += 20 ;
            cpbonus[k2-k2] += 10;
            cpbonus[k3-k3] +=5;


            return cpbonus;
}


function ScoreCalculation(num: x) num:
{
loop (x < 5)
{
j = c1[x];
k = c2[x];
j = j + k;
  ctotal[x] = j;
  x = x+1;
}
}
i = 0.0;
ScoreCalculation(i);
FindWinner(i);
i = 0;
cbonus = bonusfun1(ctotal);
caverage = colaverage(c1,c2);
generate_table c0, c1,c2,ctotal,cwinner,cbonus,caverage;

num: i;
num: j;
num: k;

col:c0<num:,"Speed">={ 50.0 , 40.0};
col:c1<num:,"Time">={ 18.0,  20.0};
col:cdist<num:,"Distance">;
cdist = c0;
```

Demo Output

TeamNames	"Score1"	"Score2"	"Sum"	"Ranks"	"Bonus"	"Avg"
NewYorkKnicks	20.	40.	60.	"None"	60.	30.
TeamMiamiHeat	30.	34.	64.	"Third"	69.	32.
_ChicagoBulls	25.	22.	47.	"None"	47.	23.5
DetroitPistons	33.	44.	77.	"First"	97.	38.5
HoustonRockets	42.	28.	70.	"Second"	80.	35.

2. IndextTest.tab

```
num: i;
str: t;

col:c0<num:,"heading">={ 1.0 , 2.0, 3.0};
col:c1<str:,"heading">={ "Hello",  "fname" , "lname" };
col:c2<str:,"heading">={ "World" };
col:cnew<num:,"heading">;
cnew = c0;


col:cdecl<num:,"heading">;

function changeData(num:x) num:
{

 x = x + 1;
 str: temp;
 temp ="lakshmifromtemp";
  c1[x] =  temp;
}
i = 1.0;
changeData(i);

num:end;

end=2;

num:start;
start=0.0;


c0[start-end] += 10;


num:j;
j=3;
c0[j] = 20.0;
j = j+1;
c0[j] = 21;

num: oi;
str: s1;

generate_table cnew, c0, c1, c2;
```

3. marks.tab

```
num: i;
num: j;
num: k;
num: l;
num: p;

col:c0<num:,"Mathematics">={ 100.0 , 90.0, 95.0};
col:c1<num:,"English">={ 88.0,  77.0 , 90.0 };
col:c2<num:,"History">={ 80.0 , 90.0, 75.0};
col:cnew<num:,"Total">;
col:cavg<num:,"Average">;
col:cper<num:,"Percentage">;
cnew = c0;

function ComputeTotal(num:x) num:
{
```

```
loop (i < 3)
{
j = c0[i];
k = c1[i];
j = j + k;
k = j/200;
p = k * 100;
  cnew[i] = j;
  cavg[i] = k;
  cper[i] = p;
  i = i+1;
}
}
i = 0.0;
ComputeTotal(i);

generate_table c0, c1,cnew,cavg,cper;
```

4. distance.tab

```
function distancetravelled(num:x) num:
{

loop (i < 2)
{
j = c0[i];
k = c1[i];
j = j * k;
  cdist[i] = j;
  i = i+1;
}
}
i = 0.0;
distancetravelled(i);

generate_table c0, c1,cdist;
```

5. HelloWorld.tab

```
num: y;
col:c1<str:,"heading">={ "Hello"};
col:c2<str:,"heading">={ "World"};
col:c3<num:,"heading">={ 1.0 , 2.0};

num: i;

generate_table c1, c2, c3 ;
```

6. ifTest.tab

```
col:c0<str:,"Myheading">={"Hello"};
num: i;
num:j ;
i=15;
j =0.0;
if (i < j )
{
  c0[j] = "less";
}
else
{
  c0[j] = "greater";
}

generate_table  c0;
```

7. IndexAssign.tzb

```
col:c0<num:,"heading">={ 1.0 , 2.0, 3.0};
col:c1<str:,"heading">={ "Hello",  "fname" , "lname" };
col:c2<str:,"heading">={ "World" };
c1[1] = "Anu";

generate_table  c0, c1, c2;
```

8. loopTest.tab

```
num: i;


col:c0<num:,"Id">={1.0};
num:j ;
i=0;
j =1.0;

loop (i < 10 )
{
  c0[i] = j;
  j = j+1;
  i = i+1;
}

generate_table  c0;
```

9. returnstatement.tab

```
num:t;
col:c0<num:,"returnstatement">={ 1.0 , 2.0, 3.0};
function product(num:i, num:j) num:
{
num: k;
k =i* j;
return k;
}

t = product(2,3);
c0[1] = t;
generate_table c0;
```

10. symbolTable.tab

```
num: i;
str: t;
col:c2<num:,"Score2">={ 40.0 , 34.0, 22.0, 44.0, 28.0};
function product(num:i, num:j) num:
{
        num: fi;
        str: ft;
}
```

11. iffile.tab

```
col:c0<str:,"Myheading">={"Hello"};
num: i;
num:j ;
i=15;
j =0.0;
```

```
if (i < j )
{
  c0[j] = "less";
}
generate_table  c0;
product( i,t);
num: oi;
str: s1;

generate_table c2;
```

4.6 *Negative* ***Test Cases:***

1. Divide by Zero

```
num: i;

i=0;
i = i / 0.0;
```

2. generating a table with a string instead of a column.

```
col:c0<num:,"heading">={ 1.0 , 2.0, 3.0};
col:c1<str:,"heading">={ "Hello",  "fname" , "lname" };
col:c2<str:,"heading">={ "aa" };

num: i;
str: g;
g = "l";
i = 8-9;
c0[2] =7;

generate_table  c0, c1, g;
```

3. undeclared identifier

```
generate_table  c0, c1, c2;
col:c0<num:,"heading">={ 1.0 , 2.0, 3.0};
col:c1<str:,"heading">={ "Hello",  "fname" , "lname" };
col:c2<str:,"heading">={ "aa" };
num: i;
i = 8-9;
c0[2] =7;
```

4. Index assignment to an incompatible type.

```
col:c0<num:,"heading">={ 1.0 , 2.0, 3.0};
col:c1<str:,"heading">={ "Hello",  "fname" , "lname" };
col:c2<str:,"heading">={ "World" };
c1[1] = 4.0;
generate_table  c0, c1, c2;
```

5. Index not dexlared

```
col:c0<num:,"heading">={ 1.0 , 2.0, 3.0};
col:c1<str:,"heading">={ "Hello",  "fname" , "lname" };
col:c2<str:,"heading">={ "World" };
num: i;
i = 9;
c0[j] =7;
generate_table  c0, c1, c2;
```

6. Index out of range

```
col:cfirst<num:,"heading">={ 1.0 , 2.0, 3.0};
col:csecond<str:,"heading">={ "Anu",  "Lakshmi" , "Rajat" };
col:cthird<str:,"heading">={ "World" };
num: i;
i = 9;
cfirst[i] =7;

generate_table  cfirst, csecond, cthird;
```

7. Index not a number type.

```
col:c0<num:,"heading">={ 1.0 , 2.0, 3.0};
col:c1<str:,"heading">={ "Hello",  "fname" , "lname" };
col:c2<str:,"heading">={ "World" };
c0["a"-1] /= 2;

generate_table  c0, c1, c2;
```

8. Index out of range

```
col:c0<num:,"heading">={ 1.0 , 2.0, 3.0};
col:c1<str:,"heading">={ "Hello",  "fname" , "lname" };
col:c2<str:,"heading">={ "aa" };
num: i;
i = 8-9;
c0[i] =7;

generate_table  c0, c1, c2;
```