Table Generation Language TabPro

Project LRM

Rajat Dixit UNI: rd2392
Anureet Dhillon UNI:ad2660
LakshmiNadig UNI:ln2206

Contents

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.
Various tokens present in the file could be:

- Keywords
- Identifiers or variables
- Operators
- Constants

Tokens are separated from each other by using a white space or a semicolon.

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.

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.

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.

1.4 Keywords:

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

num	string	loop
return	function	row
col	if	else
col_heading	row_heading	row_limit
col_limit	col_sort_index	row_filter_condition
size	currIndex	generate_table

1.5 Constants

The kinds of constants used in TabPro are listed below:

(a) Number Constants: These are declared using the keyword num: datatype in Tabpro. These include decimal integers which are a string of decimal digits from [0-9] and real numbers of the form [0-9][.][0-9.

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

1.6 Declarations

TabPro declares an identifier when it is first encountered in a file. A declaration is made by placing the <datatype: identifierName> combination on the left side of the assignment operator. For instance, string: tabstring = "greetings!" declares a string type identifier. Here the data type could be num, string, row or col.

1.7 Operators:

(a) All basic arithmetic operators (+, - , *, /) are supported by TabPro. These operators work on row and column level as well. For instance num: abc = def+5;
(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 also be applied to range of rows and columns as in myrow[3-5] += 5.
(c) '[]' operator is supported to access row or column's elements.
Therefore, in order to access the 5th element of a row, one might use myrow[5].
(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.
(e) logical operators like &&, || are used to join relational expressions. && operator is the logical AND and || operator represents the logical OR.
(The precedence of these operators is same as their counterparts in C language).

1.8 Separators:

While declaring a row or a column, in order to separate two elements "," is used. For instance, row: myrow = {3,88,45,33,12,77,59,48} declares a row.
Also, TabPro recognizes a semicolon ";" as the line separator which is the end of an executable statement.

1.9 Scope and Name Space:

Curly braces are required to mark the scope of a function or a looping construct. TabPro supports a single name space i.e. only one identifier can have a particular name, be it a function or an identifier.

1.10 Built in functions and reserved keywords:

(a) col_heading
This is a reserved word; a special list that holds the label values and types for the elements of the respective column. This is a required field that the programmer has to set.

(b) row_heading
This is a reserved word; a special list that holds the label values for the rows. This is optional and need not be specified by the programmer.

(c) row_limit

This is a reserved word that defines the limit for the row i.e. the maximum number of rows that can be present in the table.

(d) col_limit

This is a reserved word that defines the limit for the column i.e. the maximum number of columns that can be present in the table.

(e) col_sort_index

This is a reserved word; a setting that can be used by the developer to set the column by which the rows need to be sorted.

(f) row_filter_condition

This is a reserved word; a setting that can be used by the developer to set the filtering condition by which the rows will be filtered before final display.

(g) generate_table(arg)
This built in function is used to print the table on the console or the file depending upon the argument. "arg" can be 0,1 or 2 for standard console, file and both respectively. If arg is 1 or 2, the generated table would be stored in a file having the name as that of the table itself.

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

2. Types:

TabPro supports the following data types:

(a) **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]. These can be preceded by a – to indicate negative numbers. The range of these numbers will be the same as a float type.
(b) **string:** Strings are allowed which are a collection of ASCII characters.
(c) **row**: This data type is used to declare a row of elements.

For Instance, row: myrow = {4,5,6,7}; declares a row of a table having four columns.

(d) col: This data type is used to declare a column of elements.
For Instance, col: mycol = {4,5,6,7}; declares a column of a table having four rows initialized to 4, 5, 6 and 7.

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.

3.1 Primary Expressions

Primary expressions are identifiers, numbers and strings.

primary-expression
identifier
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 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.

3.2 Postfix Expressions

The operators in postfix expressions group left to right.

postfix-expression[expression]
postfix-expression(argument-expression-list-optional)

3.2.1 Indexed Interpretation

A postfix expression followed by an expression in square brackets is a postfix expression denoting a subscript that specifies a range of columns. Thus, if the type of an expression or sub-expression is “myrow [5]” then it is interpreted as the 5th column of the row named ‘myrow’. Similarly, if an expression is written like “myrow of integer1, integer2”, for any row ‘myrow’, then the value of the expression is interpreted as a ranged index, i.e. for myrow- column ranging from integer1 to integer2, and the type of the expression is same as defined by the user.

For e.g. myrow[4-6] *= 3 means the values of column number 4, 5 and 6 in 'myrow' will be multiplied by 3.

3.2.2 Function Calls

A function call is a postfix expression, followed by parentheses, containing possibly empty, comma separated list of assignment expressions, which constitutes to the arguments to the function.

num:a = my function ( );
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 Expressions with Arithmetic Operators

Multiplicative and additive operators * , /,+,- are grouped left to right. The expressions on both side of operators should evaluate to a number data type.
Airthmetic operators are not defined for string data types.

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

3.4 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 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, and if it is false, it would return value 0.

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

relational-expression==relational-expression
relational-expression != relational-expression
expresison
The operators used here hold usual meanings and are supported to compare two expressions.

3.5 Expressions with Logical And Operator

logical-AND-expression:
logical-AND-expression && *expression*
The && operator groups left to right. It returns 1 if both its operands compare unequal to zero, 0 otherwise. && guarantees left-to-right evaluation: the first operand is evaluated, including all side effects; if it is equal to 0, the value of the expression is 0. Otherwise, the right operand is evaluated, and if it is equal to 0, the expression's value is 0, otherwise 1.
The operands need to have same arithmetic type.

3.6 Expressions with Logical Or Operator

logical-OR-expression:
logical-OR-expression || *expression*
The '||' operator groups left-to-right. It returns 1 if either of its operands compare unequal to zero, and 0 otherwise. Unlike '||' follows left-to-right evaluation: the first operand is evaluated, including all side effects; if it is unequal to 0, the value of the expression is 1. Otherwise, the right operand is evaluated, and if it is unequal to 0, the expression's value is 1, otherwise 0.
The operands need to have same arithmetic type.

3.7 Expressions with Assignment operators

There are several assignment operators; all group right-to-left.
assignment-expression:
unary-expression *assignment-operator assignment-expression*
assignment-operator:
= , *= , /= , += , -=
All require an lvalue as left operand, and the lvalue must be modifiable: it 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 have the same arithmetic type.

4. Statements

Statements can be one of the many kinds of statements described int eh sections below:
Statement:
normalStmnt

blockStmnt
condStmnt
iterStmnt
returnStmnt
funcDeclStmnt

4.1 Normal Statement

These statements include expressions statements and declaration statements ending with a semicolon ; . These include declarative statements and expression statements.

normalStmnt:
declarativeStmnt
expressionStmnt

4.2 Block Statement

These include one or more normal statements that are nested within the curly braces.

blockStmnt: { statement_list }

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)
}
```

condStmnt:
if (expression) statement
if (expression) statement else{ statement }

4.4 return statement

return exits out of a scope (block of code) and optionally returns a value and may
only be defined inside the scope of a function definition. Abstract example: return <expression>. The return statement in a function is optional.

returnStmnt:
return expression

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

<row/column variable>[<start index>-<end-index>]

This will perform the following block of statement(s) for the range of columns in the row or range of rows in a column. This can also be used with operators as defined in expressions . Examples:

To perform a block of statement(s) for a range of rows in myCol

```
myCol[3-6] {
Block of Statement(s)
}
```

To add 2 to all columnd of myRow:

```
Myrow[2-5] += 2;
```

iterStmnt:

identifier [constant – constant] block_statement

loop (expression) block_statement

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.

Abstract example:

```
function <identifier> (list of <type:identifier>) <return type:return dientifier>
{
statement(s);
}
```

A more concrete example:

```
function product(num:I, num:j) num:k
{
k = I * j;
}
function sqr(row:myRow) row:resultRow
{
 resultrow = myRow * myRow;
}
```

funcDeclStmnt:

function identifier (argument_list) declaration_specifier

5. Library Support

The language will provide one library for some common statistical functions. This will be the “statistics” library which will support the statistical functions like finding mean, median, mode and variance. The library functions would be implemented in TabPro.

6. Grammar rules

constant:
- *number*
- *string*

declaration_specifier:
- *numtype identifier*
- *stringType identifier*

expression:
- *primary-expression*
- *postfix-expression*
- *arithmetic-expression:*
- *relational-expression:*
- *logical-AND-expression:*
- *assignment-expression*

primary-expression
- *identifier*
- *constant*

postfix-expression
- *postfix-expression[expression]*
- *postfix-expression(argument-expression-list-optional)*

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

relational-expression:
- *relational-expression < relational-expression*
- *relational-expression > relational-expression*
- *relational-expression <= relational-expression*

relational-expression >= relational-expression
relational-expression == relational-expression
relational-expression != relational-expression
expresison

logical-AND-expression:
logical-AND-expression && expression
logical-AND-expression && expression

assignment-expression:
identifier = assignment-expression
identifier[constant] = assignment-expression
expression

statement:
normalStmnt
blockStmnt
condStmnt
iterStmnt
returnStmnt
funcDeclStmnt

normalStmnt:
declarativeStmnt
expressionStmnt

blockStmnt: { statement_list }

if (expression) statement
if (expression) statement else{ statement }

iterStmnt:
identifier [constant - constant] block_statement
loop (expression) block_statement

condStmnt:
if (expression) statement
if (expression) statement else{ statement }

returnStmnt:
return expression

funcDeclStmnt:
function identifier (argument_list) declaration_specifier

argument_list:
argument_list , arg

arg :
declaration_specifier identifier

7. Lexer Tokens

```
{

}
let digit = ['0'-'9']
let comma = [',']
let semicolon = [';']
let colon = [':']
let dot = ['.']
let exp = ['e''E']
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 mathOp = "+" | "-" | "*" | "/"
let assign = '='
let compOperator = "<" | ">" | "=="
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"

rule token =
parse   | [' ' '\t' '\r' '\n'] { token lexbuf }
        | ( signedint)+ (dot digit*)?  as n { NUMBER(float_of_string n) }
        |  numType colon {NUMTYPE}
        |  strType colon {STRTYPE}
        |  rowType colon {ROWTYPE}
        |  colType colon {COLTYPE}
        |  mathOp { MATHOPERATOR }
        |  assign {ASSIGN}
        |  compOperator as op{COMPARATOR(op)}
            |  commentStart _ newLine {COMMENT}
            |  size {SIZE}
            |  currIndex {CURRINDEX}
            |  if {IF}
            |  else {ELSE}
            |  leftParan {LEFTPARAN}
            |  rightParan {RIGHTPARAN}
            |  leftSqParan {LEFTSQPARAN}
            |  rightSqParan {LEFTPARAN}
            |  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}
        |  alpha (alphanum)* as name {IDENTIFIER(name)}
        | _  { ERROR}
            | eof { raise End_of_file }

    {

    }
```