Middle East Technical University | Department of Computer Engineering

CENG 444

Language Processors

Spring 2023-2024

Project I - Part II

Due date: 25 April 2024, Thursday, 23.59

1 Problem Definition

In this assignment, you are required to write a C++ program that performs a syntactic conformance test of a fictitious programming language Text Query Language (TQL). This so-called text query language will also be the basis of the forthcoming assignments on semantic analysis and code generation. The problem particular to this assignment is the implementation of a syntax checker which requires definitions for scanner and parser, and the implementation of a command line tool that reports syntactic conformity of a given TQL module.

1.1 Text Query Language (TQL)

The so-called Text Query Language (TQL) is an imperative language with basic expression evaluation capacity extended with TQL specific types and operators. The type system on which the language was built on has a limited extensibility on its particular type, **table**. The basic data types are string, number, boolean, and text. The flow control structures of the language include while-loop, if and if-else statements, expression statements, and empty statement. There will be a minimal set of intrinsic functions that can be called from the expressions. Variables are scoped to the compound statements, and organized in hierarchical order. Compound statements are the statements enclosed by curly braces. `let` statements are used the declare variables.

Note that the semantic checks are not required even though some semantics at some level are described through the rest of this section. Forthcoming assignments will remove the points that may be vague at the moment. Just focus on syntactic analysis!

1.1.1 Types

TQL is a statically typed, type-strong language. Table 1 lists the types allowed in a TQL program. The names in the Type column are also the keywords that will be used in a TQL program. Type-equality of simple types is strict, in other words, string-type equals string-type, number-type equals number-type, boolean-type equals boolean-type. Type equality check of the table type has additional conditions.

Type level	**Type**	**Description**
Simple	number	Type to represent IEEE 754 double precision floating point value.
Simple	boolean	Type to represent boolean values.
Simple	string	String of characters.
Object	table	Table object where each column belongs to a simple type. A table object must have at least one column. Two tables are type-equal if they have an equal number of columns and each respective column are of the same type. Each column has a name (id) that can be referred to in expression contexts.

Table 1: Types available in TQL

1.1.2 Statements

A TQL program can consist of zero or more statements. The statement forms are as follows:

- **Evaluation:** An evaluation is an expression followed by a semicolon.
- **While loop:** A while loop starts with the `while` keyword followed by an expression in parenthesis. The expression type must be boolean. A statement must follow the expression in parenthesis.
- **If statement:** An if statement starts with the `if` keyword followed by an expression in parenthesis. The expression type must be boolean. A statement must follow the expression in parenthesis.
- **If-else statement:** The syntax of an if-else statement can be defined as the form of the if-statement followed by the `else` keyword and a statement.
- **Let statement:** Let statement declares a variable in the current scope. It starts with the `let` keyword. The identifier of the variable must follow. An assignment symbol `=` comes after the identifier and an expression is coded. The type of the variable is inferred from the type of the expression, and its initial value is set as the result of the expression. The statement ends with a semicolon.
- **Append statement:** Append statement adds a row to the target table. The statement starts with `append` keyword and is followed by the list of the expressions separated with a comma. The statement ends with a semicolon following the final expression.
- **Compound statement:** A compound statement is defined as statements (zero or more) enclosed by curly braces.

Do not get into details like type checking for the statements that require expressions. This assignment is on syntactic checks only.

1.1.3 Expressions

The form of the expressions in TQL is mostly conventional. The table-related operators and identifiers may be found non-conventional but they are easy to understand.

In this assignment you are required to check only for syntactic conformance. **Do not get into semantic details like type checking, type evaluation, precedence, associativity, constant folding, parameter checking, declarative integrity, and similar.**

- **Operators**
  Operators introduced in Table 4 are used with operands, which may be expressions, identifiers, and literals, to build well-formed expressions.

- **Parentheses**
  An expression can be enclosed in parentheses to change/ensure evaluation order. See the precedence and associativity of the operators in Table 4 to understand the default evaluation order.

- **String Literals**
  The string literals are in the form of regular and exceptional characters enclosed in double quotes. Exceptional characters and their string encodings are shown in Table 2.

Exceptional Character	Encoding in string
double quote(")	\"
back-slash(\)	\\
new-line	\n

Table 2: Exceptional characters and their string encodings

A character, regardless of whether it is exceptional or regular can be coded with the help of byte coding. Any byte except zero can be encoded in a string of characters by using the hexadecimal form, which is \xhh, where h is a hexadecimal digit. Capital and small letters can be used in hexadecimal digit specification.

- **Numeric Literals**
  The numeric literals can be composed of the valid combinations of whole part (W), decimal separator (.), and fractional part (F). Whole part can be either zero, or a sequence of digits starting with a non-zero digit. The fractional part is a nonempty sequence of digits, not ending with zero. Table 3 lists valid examples of W,".", and F combinations.

W	.	F	Example number
1	1	1	1.25
1	1	1	0.25432
0	1	1	.25
0	1	1	.0134
1	0	0	123
1	0	0	0

Table 3: Valid number combinations

A number can have an optional exponential part prefixed by 'e' or 'E'. When supplied, an optional sign that can be either '+' or '-' may follow. Then, a sequence of digits must be given. The first digit must be non-zero.

- **Boolean Literals**
  `true` and `false`

- **Table Literals**
  Table literals are declared as `@[<string>]` where `<string>` is the name of the text file associated with the table object.

- **Identifiers**
  There are two kinds of identifiers that can be referred to from an expression context. The first kind is the regular identifier, which is similar to the variable or type references you are used to. The second kind is prefixed with `$` sign. The `$` prefixed identifiers are special and only valid when referred in the expressions that are the second operand of an operator that takes a table as the first operator, which are `select` and `mutate`. `$<positive integer>.<field name>` is the form to use in `mutate` and `$<field name>` is the form for the `select` operator.

  - `mutate` **example**
    `Temp=Students*Courses[number $1.studentId, number $5.courseId]`
    The cross product of `Students` and `Courses` is calculated resulting in several fields by the `*` operator. Then the mutation is performed to reduce and rename the column names. The result is assigned to `Temp` variable.

  - `select` **example**
    `Temp/($studentId=100 && ($courseId>=100 || &courseId<200))`
    This example follows the previous one. `Temp` is scanned line by line with the help of the expression and a new table object is created with the rows qualified.

- **Function Calls**
  The function call is formed as `<identifier>` followed by a list of expressions separated by commas. The list of expressions may be empty. See the Table 5 and the shared example source.

- **Mutation** A mutation is formed as `<expression>` followed by a `<column-list>` enclosed by "`[]`" brackets. Refer to the example source in Appendix Section [4.3].

1.1.4 Expression Lists

An expression list is formed by at least one expression. Additional expressions may be coded with a preceding comma to form a multi-expression list. Expression lists are constructs that are used with the function call operator. See the list of operators in Appendix Section 4.

1.1.5 Column Lists

A column list is formed by at least one identifier. Additional column declarations may be coded with a preceding comma to form a multi-column list. Column lists are constructs that are used with the mutate operator. See the list of operators in Appendix Section 4.

A column declaration may have one of the following forms:

- Form 1:
  `<simple type>`

- Form 2:
  `<simple type><identifier>`

1.2 Processor Output

The processor you develop, `project01syn`, will accept the source file that will be syntactically analyzed from the command line. The command must contain only one parameter. No parameter or more than one parameter cases must be reported as error. The processor must create an output file `project01syn.txt` containing 12 lines with collected frequencies in case it recognizes the input as a syntactically valid program. The strict ordering of the lines is given as follows:

```
<Total number of variable declarations, not let statements>
<Total number of the evaluation statements>
<Total number of the while statements>
<Total number of the if statements>
<Total number of the if-else statements>
<Total number of the append statements>
<Total number of the compound statement>
<Total number of the table literals>
<Total number of the function calls>
<Maximum number of possibly select operations>
<Total number of mutation operations>
<Total number of materializations>
```

In case the input file is found syntactically non-compliant, the output file `project01syn.txt` will contain a single line as follows:

```
Line <line number>: Syntax error.
```

2 Specifications & Hints

- Keep in mind that the language processor you will work through the rest of the term is TQL.
- In TQL, white spaces can be inserted between tokens to increase human-level readability.
- In TQL, comments can be inserted into any line starting with double slashes "//". Comments are terminated with an end-of-line character.
- Use `flex` and `bison` to generate `C++`, not `C` file. `C` implementation will **not** be accepted.
- Never modify the files that are automatically generated by `flex` and `bison`. You may prefer consulting the recitation material and the sample project on syntactic analysis.
- In case you use Eclipse IDE, take care following the directions below, which are easy as these are compliant with defaults:
  - Place all of your source files in the project folder, not in any subfolder.
  - Develop your solution in a way that the input and output text files (.txt) will be placed in the project folder.
  - Do not make any changes to project options that designate executable target folder other than Debug. The evaluation process will use Debug build.
  - Make sure that make utility succeeds building when run (`make clean` followed by `make all`) in the Debug folder of the project.
  - Submit your project folder (not the workspace) as a .zip file in compliance with the naming conventions noted above. The project folder is the folder containing the .project folder.

- If you breach from the advice of using Eclipse C/C++ complying with the directions provided, your presence and participation in the evaluation may be required. In this case, please describe precisely the steps to build and run the program. Additionally, provide shell scripts for building and running with the support of a README file that contains directions for building and running, you may also provide a makefile if needed.

3 Regulations

- **Implementation:** You should use `flex` and `bison` with `C++` to write your program. Make sure that your program will accept the name of the input file. In case the program is run without a parameter or with more than one parameter your program must terminate immediately with an appropriate error message sent to the standard output.
- **Testing:** Your programs will be tested on an Ubuntu (22.04) machine. It is strongly advised that you set up a virtual machine if you do not have access to a machine with Ubuntu or any other Linux distro.
- **Debugging:** You are advised to use Eclipse IDE for C/C++ for debugging and tracing your program. This will come in handy, especially for the later stages of the project.
- **Evaluation:** The evaluation will be based on the correct diagnosis of 8 different input files, one point for each.
- **Submission:** You need to submit all relevant files you have implemented (.cpp, .h, .l, .y, etc.) as well as a README file with instructions on how to run, a shell script and a makefile to run in a single .zip file named `<studentID.zip>`. If you are using Eclipse IDE, please submit your project folder as a .zip file named `<studentID.zip>`
- This project is expected to be an individual effort, but discussions are always welcome. Please do not hesitate to use the mail group for your questions.

4 Appendix

4.1 Operators

Table 4 lists the operators that will be available. For each operator, form (coding syntax), possible applications on types, type of the result, associativity, and precedence are given. It should be noted that:

- The `<type1>` and the `<type2>` in relational operators must be type-equal.
- The relational operators can be applied to simple types.
- The merge operator can be applied to two tables on the condition that the tables are type-equal.

OP	Short Name	Form	Application	Result Type	Assoc.	Prec.
=	Assign	Binary, infix	id=<expression>	Type of <expression>	RL	0
&&	Boolean AND	Binary, infix	<boolean> && <boolean>	<boolean>	LR	1
||	Boolean OR	Binary, infix	<boolean> || <boolean>	<boolean>	LR	1
==	Equal	Binary, infix	<type1> == <type2>	<boolean>	LR	2
!=	Not equal	Binary, infix	<type1> != <type2>	<boolean>	LR	2
<	Less than	Binary, infix	<type1> < <type2>	<boolean>	LR	2
>	Greater than	Binary, infix	<type1> > <type2>	<boolean>	LR	2
<=	Less than or equal	Binary, infix	<type1> <= <type2>	<boolean>	LR	2
>=	Greater than or equal	Binary, infix	<type1> >= <type2>	<boolean>	LR	2
+	Add	Binary, infix	<number> + <number>	<number>	LR	3
+	Concatenate	Binary, infix	<string> + <string>	<string>	LR	3
+	Concatenate	Binary, infix	<string> + <number>	<string>	LR	3
+	Merge	Binary, infix	<table> + <table>	<table>	LR	3
-	Subtract	Binary, infix	<number> - <number>	<number>	LR	3
*	Multiply	Binary, infix	<number> * <number>	<number>	LR	4
*	Cross product	Binary, infix	<table> * <table>	<table>	LR	4
/	Divide	Binary, infix	<number> / <number>	<number>	LR	4
/	Select	Binary, infix	<table> / <boolean>	<table>	LR	4
->	Materialize	Binary, infix	<table> / <string>	<number>	LR	4
-	Negate	Unary, prefix	-<number>	<number>	RL	5
!	Boolean negate	Unary, prefix	!<boolean>	<boolean>	RL	5
()	Call function	n-ary, postfix	id(<expression-list>)	Return type of function	LR	6
[]	Mutate	n-ary, postfix	<table>[column-list]	<table>	LR	6

Table 4: Operators available in TQL

4.2 Intrinsic Functions

Table 5 lists the available intrinsic functions in TQL.

Prototype	**Description**
number len(string str)	Returns the number of the characters in string.
string num2str(number n)	Converts n to string.
number str2num(string str)	Converts str to number.
number bool2num(boolean b)	Converts b to number.
boolean num2bool(number n)	Converts n to boolean.
boolean str2bool(string str)	Converts str to boolean.
string bool2str(boolean b)	Converts b to string.
number rowcount(table t)	Calculates the number of rows in a table.
number sum(table t, number n)	Calculates the sum of the field fieldId in all rows of t.
number sumofsquares(table t, number n)	Calculates the sum of the field fieldId squared in all rows of t.
number max(table t, number n)	Calculates the maximum of the field fieldId in all rows of t.
number min(table t, number n)	Calculates the minimum of the field fieldId in all rows of t.
number sum(table t, string fieldId)	Calculates the sum of the field fieldId in all rows of t.
number sumofsquares(table t, string fieldId)	Calculates the sum of the field fieldId squared in all rows of t.
number max(table t, string fieldId)	Calculates the maximum of the field fieldId in all rows of t.
number min(table t, string fieldId)	Calculates the minimum of the field fieldId in all rows of t.
string rename(string fileName)	Renames the file specified by fileName.
string copy(string sourceFileName, string destFileName)	Copies the source file specified by sourceFileName to the destination file specified by destFileName.
boolean delete(string fileName)	Deletes the file specified by fileName.
boolean clearWorkArea()	Clears the files in the work area.
number int(number n)	Returns the integer (whole) part of a number.
string mid(string str, number startIndex, number count)	Calculates substring using the parameters.
number random()	Returns a random number r with the condition that `0<=r<1`.

Table 5: Intrinsic functions available in TQL

4.3 Example Source

Below is an example source code in TQL.

```
remove("output.txt");

let b=@["grades.txt"][string studentId, string courseId, number grade],
    output=@["output.txt"][number tableNo, string description],
    k=0;

let i=1;

while (i<=10)
{
   let t=b / ($courseId==100+i);

   t->("t"+i+".txt");

   let r=rowcount(t);
   append output, i, "Processed " + t + "lines\n";

   if (r>0)
   {
      let s=sum(t, "grade");

      if (s/n>=50)
         append output, i, "Successful!";
   }

   i=i+1;
}
```