Middle East Technical University | Department of Computer Engineering

# CENG 444

Language Processors

Spring 2023-2024

## Project II - Intermediate Representation

Due date: 31 May 2024, Friday, 23.59

# 1 Problem Definition

In this assignment, you are required to write a C++ program that performs the following by accepting source codes written in the fictitious programming language Text Query Language (TQL):

- **Augmenting Graphical IR:** Partially constructed control flow structures, variable lists, and expression trees will be augmented, completed, and presented.
- **Checking the validity:** Numerous checks to ensure compliance of the given source will be performed and results will be reported.
- **Fixing up Linear IR:** Incomplete intermediate code consisting of a sequence of the stack machine instructions will be improved and presented.

You are supplied with a starting project source that is capable of recognizing a syntactically correct TQL source. This supplementary code has the capacity to construct graphical IR, to emit intermediate code as linear IR, to represent variables and members, and to accumulate messages. It also provides with a basis for the capacities including but not limited to the generation of main output, representation of types, variables, and members.

The syntactic structure and semantics of a TQL program are slightly different from the one you dealt with in the latest assignment. Differences are as follows:

- **Definition of column list form:** There is one uniform syntactic rule for the column lists.
- **Expression lists and the comma operator:** The comma operator is introduced to change the way of building up the expression lists. See the appendix for the updated list of the operators.
- **Definition of runtime functions and intrinsic functions:** The "intrinsic functions" are changed to "runtime functions". Intrinsic functions are defined as the functions that act behind the scenes to implement operators. See the relevant appendix for details.

# 2 Processor Output

The language processor you develop will generate an improved JSON file that delivers the three main components (the graphical IR, the linear IR, and the list of messages) given in the introduction to the problem. Additionally, it will generate a text file to present the linear IR in a more readable format.

The test material given for the previous assignment and the expected outputs are accessible in the GitHub repository to clarify the properties of the output and its relationship with the three main components. The material accessible from the link also contains the grammar files for your attention.

## 2.1 Augmented Graphical IR

You are given an infrastructure that generates the graphical IR. You are required to implement components that pass over the relevant structures to perform the following:

- **Stack load evaluation:** Each node in an expression three has an impact on the evaluation stack. For example, a double precision floating point constant has an impact of +1 on the stack size while a hypothetical function having three parameters has an impact of -2. The stack load of the root node is the net stack load of the expression, which needs to be handled carefully.
- **Type evaluation:** Each node in an expression three has a type descriptor that should be evaluated by examining the child nodes. Leaf nodes of an expression must be generated in a way to ensure they have correct type descriptors. The result of the type evaluation will be in various places like type inference for the let-assignments, parameter type checking, and operator applicability.
- **Constant folding:** The operators having constant children may be reduced by the language processor. Your program must reduce the nodes having constants for string, number, and boolean types.
- **Variable tables:** Augment each compound statement and select operator node with table of variables. For each entry in the variable table calculate and store a local address index.

## 2.2 Validity Check

Implement a pass or passes over graphical IR so that a given source code can be validated. The linear IR will be generated only when the given input is found as error free. Table 1 shows what will be checked and reported when detected. Discussions for each of the errors will be made in the lab hours.

| | |
|---|---|
| Improper operand. | When the operands and the operator found are not applicable this error message will be generated. Operator applicability is defined in the table of the operators in the appendix. |
| Call to unknown function. | When a call operator is detected the identifier data must be used to identify the function. This error message is generated when the function is not identified. The table of the available functions is given in the appendix. |
| Parameter type mismatch. | This error message is generated when the type of an expression node as an actual parameter does not match the type of the corresponding formal parameter. The table of the available functions is given in the appendix. |
| Unknown variable reference. | This error message is generated if an identifier cannot be resolved. |
| Multiple definition for variable. | This error message is generated if an identifier has already been defined. |
| Wrong number of parameters. | The number of the parameters to a specified function is fixed by definition. See the table of the available functions given in the appendix to determine the number of the parameters correctly. |
| Member index out of range. | This error occurs when a mutation operand refers to a column number that falls out of the range defined by the table operand. |
| Expression must be boolean. | The condition expression that is found in “if” or “while” statement must be boolean. Similarly, the second operand of the select operator must be boolean. |
| Expression must be string. | The table literal expression must be string. |
| Improper type for append table. | The append statements for expression must be a table. |

Table 1: Validity checks and errors to be reported.

## 2.3 Linear IR Fix-up

The supplemental code generates a sequence of instructions naïvely by using expression tree opcodes. You are expected to:

- Replace the instructions for the constants with typed push `(OP_PUSH)` instructions.
- Replace the arithmetic instructions with code templates containing calls to intrinsic functions `(OP_CALL)`. A special care must be taken for the implementation of select as it requires a conditional loop template.
- Improve the id references with consistent variable offsets.
- Ensure variable retrieval instructions `(OP_ID)` have correct offsets.
- Ensure function calls have correct function index numbers.

# 3 Regulations

- **Implementation:** You are required to use the supplemental code and apply modifications as you see fit. You are responsible for applying modifications and patches to the supplemental components when issued.
- **Attendance:** Many details will be clarified in the lab hours. Additional validity checks and improvements on IR may be asked as identified during the discussions. Attendance is crucial for successfully completing this assignment.
- **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.
- 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 the make utility succeeds in 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.

# 4 Appendix

## 4.1 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.

### 4.1.1 Types

TQL is a statically typed, type-strong language. Table 2 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 2: Types available in TQL

### 4.1.2 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 variables in the current scope. It starts with the `let` keyword by a variable assignment or a series of variable assignments separated by commas. In each variable assignment, 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 an expression. Use of the comma operator at the root level separates the actual parameters to form an argument list where the first argument must be a table. The tuple formed by the rest of the parameters must be type-compliant with the field set of the table.

- **Compound statement:** A compound statement is defined as statements (zero or more) enclosed by curly braces.

### 4.1.3 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.

- **Operators**
  Operators introduced in Table 5 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 5 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 3.

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

Table 3: 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 4 lists valid examples of W,".", and F 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`

| 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 4: Valid number combinations

- **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 an expression list enclosed by parentheses. The list of expressions is skipped for the functions having no formal parameters.

- **Mutation** A mutation is formed as `<expression>` followed by a `<column-list>` enclosed by "`[]`" brackets. Refer to the example source in Appendix Section [4.5].

### 4.1.4 Variable Scopes

Each compound statement establishes a scope for the variables defined within. Additionally, any select operator defines a scope by using the fields in its table operand.

It is possible to override a symbol defined in higher scopes. Any reference to a variable will be resolved according to the hierarchy of scopes prioritizing the innermost.

### 4.1.5 Expression Lists

An expression list is constructed by the comma operator which responds all of the requirements of parameter passing in a function call. See the list of operators (Table 5).

### 4.1.6 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 mutation operator. See the list of operators (Table 5).

At least one column declaration must be made. Any additional column declaration must be preceded by a comma. A column declaration has the form `<simple type><identifier>`.

## 4.2 Operators

Table 5 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 noted below 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. |
|---|---|---|---|---|---|---|
| `,` | Comma | Binary | | Multi | LR | 0 |
| `=` | Assign | Binary, infix | `id=<expression>` | Type of `<expression>` | RL | 1 |
| `&&` | Boolean AND | Binary, infix | `<boolean> && <boolean>` | `<boolean>` | LR | 2 |
| `\|\|` | Boolean OR | Binary, infix | `<boolean> \|\| <boolean>` | `<boolean>` | LR | 2 |
| `==` | Equal | Binary, infix | `<type1> == <type2>` | `<boolean>` | LR | 3 |
| `!=` | Not equal | Binary, infix | `<type1> != <type2>` | `<boolean>` | LR | 3 |
| `<` | Less than | Binary, infix | `<type1> < <type2>` | `<boolean>` | LR | 3 |
| `>` | Greater than | Binary, infix | `<type1> > <type2>` | `<boolean>` | LR | 3 |
| `<=` | Less than or equal | Binary, infix | `<type1> <= <type2>` | `<boolean>` | LR | 3 |
| `>=` | Greater than or equal | Binary, infix | `<type1> >= <type2>` | `<boolean>` | LR | 3 |
| `+` | Add | Binary, infix | `<number> + <number>` | `<number>` | LR | 4 |
| `+` | Concatenate | Binary, infix | `<string> + <string>` | `<string>` | LR | 4 |
| `+` | Concatenate | Binary, infix | `<string> + <number>` | `<string>` | LR | 4 |
| `+` | Merge | Binary, infix | `<table> + <table>` | `<table>` | LR | 4 |
| `-` | Subtract | Binary, infix | `<number> - <number>` | `<number>` | LR | 4 |
| `*` | Multiply | Binary, infix | `<number> * <number>` | `<number>` | LR | 5 |
| `*` | Cross product | Binary, infix | `<table> * <table>` | `<table>` | LR | 5 |
| `/` | Divide | Binary, infix | `<number> / <number>` | `<number>` | LR | 5 |
| `/` | Select | Binary, infix | `<table> / <boolean>` | `<table>` | LR | 5 |
| `->` | Materialize | Binary, infix | `<table> / <string>` | `<number>` | LR | 5 |
| `-` | Negate | Unary, prefix | `-<number>` | `<number>` | RL | 6 |
| `!` | Boolean negate | Unary, prefix | `!<boolean>` | `<boolean>` | RL | 6 |
| `()` | Call function | n-ary, postfix | `id(<expression-list>)` | Return type of function | LR | 7 |
| `[]` | Mutate | n-ary, postfix | `<table>[column-list]` | `<table>` | LR | 7 |

Table 5: Operators available in TQL

## 4.3 Runtime Functions

Table 6 lists the available runtime functions in TQL.

| Index | Prototype | Description |
|---|---|---|
| 1 | double len(string *str) | Returns the number of the characters in string. |
| 2 | string* num2str(double n) | Converts n to string. |
| 3 | double str2num(string *str) | Converts str to number. |
| 4 | double bool2num(bool b) | Converts b to number. |
| 5 | bool num2bool(double n) | Converts n to boolean. |
| 6 | bool str2bool(string *str) | Converts str to boolean. |
| 7 | string* bool2str(bool b) | Converts b to string. |
| 8 | double rowcount(TQLTable *t) | Calculates the number of rows in a table. |
| 9 | double sum(TQLTable *t, double n) | Calculates the sum of the field fieldId in all rows of t. |
| 10 | double sumofsquares(TQLTable *t, double n) | Calculates the sum of the field fieldId squared in all rows of t. |
| 11 | double max(TQLTable *t, double n) | Calculates the maximum of the field fieldId in all rows of t. |
| 12 | double min(TQLTable *t, double n) | Calculates the minimum of the field fieldId in all rows of t. |
| 13 | double sum(TQLTable *t, string *fieldId) | Calculates the sum of the field fieldId in all rows of t. |
| 14 | double sumofsquares(TQLTable t, string *fieldId) | Calculates the sum of the field fieldId squared in all rows of t. |
| 15 | double max(TQLTable t, string *fieldId) | Calculates the maximum of the field fieldId in all rows of t. |
| 16 | double min(TQLTable t, string *fieldId) | Calculates the minimum of the field fieldId in all rows of t. |
| 17 | string* rename(string *fileName) | Renames the file specified by fileName. |
| 18 | string* copy(string *sourceFileName, string *destFileName) | Copies the source file specified by sourceFileName to the destination file specified by destFileName. |
| 19 | bool delete(string *fileName) | Deletes the file specified by fileName. |
| 20 | bool clearWorkArea() | Clears the files in the work area. |
| 21 | double int(double n) | Returns the integer (whole) part of a number. |
| 22 | string* mid(string *str, double startIndex, double count) | Calculates substring using the parameters. |
| 23 | double random() | Returns a random number r with the condition that `0<=r<1`. |

Table 6: Runtime functions available in TQL

## 4.4 Intrinsic Functions

Table 7 lists the functions that will be used while implementing the append statement, select operator, mutation operator, merge operator, cross product operator, and materialize operator.

| Index | Prototype | Description |
|---|---|---|
| -1 | void append(TQLTable *, ...) | Appends a line to the table given as the first parameter. |
| -2 | boolean fetchNext(TQLTable *) | Prepares values of iterator variables by reading a line from the table parameter. It returns true if it can read and store values. It returns false if it reaches the end of the table or fails to store iterator variables. |
| -3 | TQLTable *mutate(TQLTable *, TQLVarList *) | Redefines the column of a table. |
| -4 | TQLTable *merge(TQLTable *, TQLTable *) | Creates a new table by merging two tables. Note that the input tables must be type equals. |
| -5 | TQLTable *crossProduct(TQLTable *, TQLTable *) | Creates a new table as the cross product of the input tables. |
| -6 | TQLTable *materialize(TQLTable *) | Creates a file as contents of a table. It returns the parameter as the result. |

Table 7: Intrinsic functions that TQL language processor uses

## 4.5 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";

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

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

   i=i+1;
}
```