COMP3173 Compiler Construction
Course Project Requirements
2021 Fall
Relational Algebra Analyzer

Relational algebra is the foundation of relational databases. For those who want to master SQL, they need to start with relational algebra. Unfortunately, relation algebra is not easy for beginners. It is relatively abstract comparing to arithmetic. Even worse, beginners cannot even learn from mistakes because they do not have an efficient method to verify that whether their expressions are correct or not.
In order to help these beginners, you need to design and implement an analyzer for relational algebra expressions using **C programming language**. This analyzer is expected to find all kinds of errors in the expressions, and calculate the outcome if no errors. To help you with the design and implementation, we split the project into multiple phases.

The first phase is **lexical analysis**. In this phase, you need to design and implement a lexer following these requirements.

1. Design a DFA over the alphabet Σ which includes the following symbols. See Table 1.

Symbols	Ascii	Meaning
!	33	Exclamation
"	34	Double quotation
'	39	Single quotation
(	40	Left parentheses
)	41	Right parentheses
*	42	Star
,	44	Comma
−	45	Dash
.	46	Full stop
/	47	Slash
$[0-9]$	48-57	Numbers
<	60	Left angle brackets
=	61	Equal sign
>	62	Right angle brackets
$[A-Z]$	65-90	Letters in the upper case
_	95	Underline
$[a-z]$	97-122	Letters in the lower case

Table 1: Alaphbet

2. The DFA accepts the following strings. See Table 2.

Token	Regular Expression	Remark
identifier	"α^+ ", for $\alpha \in [a-z] \cup [A-Z]$	User defined relation names or attribute names, cotained by "
text_literial	'$\alpha^+\beta^*$ ', for $\alpha \in [a-z] \cup [A-Z]$ and $\beta \in [0-9]$	Constant strings, start with at least one letter and contained by '.
int_literial	'$0\|(\alpha\beta^*)$ ', for $\alpha \in [1-9]$ and $\beta \in [0-9]$	The integer can be a single 0, or start with a non-zero digit.
float_literial	'$0\|(\beta\alpha^*).(0\|\alpha^*\beta)$', for $\alpha \in [0-9]$ and $\beta \in [1-9]$	The digits after the point can be a single 0, or end with a non-zero digit.
project	*pi*	Relational operator, project
select	*sig*	Relational operator, select
time	$*$	Relational operator, cartesian product
divide	/	Relational operator, division
rename	*rho*	Relational operator, rename
join	*join*	Relational operator, join
assignment	*as*	Relational operator, assignment
aggregation	*agg*	Relational operator, aggregation
maximum	*max*	Aggregation function, max
minimum	*min*	Aggregation function, min
minus	$-$	Relational operator, set minus
union	*cup*	Relational operator, set union
intersect	*cap*	Relational operator, set intersect
negation	!	Logical operator
conjunction	*and*	Logical operator
disjunction	*or*	Logical operator
bra	(	Separator, open parentheses
ket	)	Separator, close parentheses
comma	,	Separator, comma
subscript	_	Separator, subscript
integer	*int*	Domain type, integer
text	*str*	Domain type, text string
float	*float*	Domain type, floating point
smaller	$<$	Relation connective, is LHS smaller than RHS?
equal	$==$	Relation connective, is LHS equal to RHS?
larger	$>$	Relation connective, is LHS larger than RHS?

Table 2: Tokens

3. The lexer is implemented as a c library, named "lexer.h" and "lexer.c".

4. The library has a function "next_token" which reads the input string of relational expression and returns the first token (token name) in the input.

5. The function uses a pointer, always pointing to the first unprocessed symbol in the input. When a token is found and returned by "next_token", the pointer is shifted to the next unprocessed symbol.

6. The function returns an error flag when lexecal errors are found.

Suppose the input expression is "sigpla" and we use a pointer p always pointing at the first unprocessed symbol. Initially, p is pointing at symbol "s". When "next_token" is called, it reads one symbol at a time and tries to find a token. After reading the first three symbols "sig", the function call should stop and return, because "sig" is a lexeme of the token type *select*. Then, the pointer p points "p", which is the first unprocessed symbol in the remaining input expression. Next, the function "next_token" is called again. And the function call returns an error flag when it sees the second symbol (in this round) "l", because no lexem starts with "pl".

The implementation also needs another source file called "analyzer.c" for the "main" function, which

- reads expressions from "in.txt", one expression on each line;
- use a loop to make function calls on "next_token";
- outputs the outcome of each expression to "out.txt", one outcome on each line.
- if the expression has no lexecal error, the outcome of each expression is a stream including
  - some token names followed by the corresponding lexeme included in "<>" (excluding the quotation marks) if the token is one of "$identifier$", "$text\_literial$", "$int\_literial$", or "$float\_literial$"; or
  - direcly lexemes, separated by an empty "space" if the token is noneof the above;
- if the expression has lexecal errors, the outcome is "Lexecal error."

See "in.txt" and "out.txt" in the package for example. To help your design and implementation, you need to know the followings.

- This phase is lexical analysis only. DO NOT think about syntax, like "are the parentheses properly paired?" We will handle syntax in the next phase.
- The tokens are nicely designed. No token is a prefix of another one (like prefix code). Think about why.
- You are recommended to use a transition table, even it can be quite large. Once the table is constructed, you solve everything once and for all.
- A symbol table is used to store identifiers and literials. We do not discuss the symbol table here to save your efforts. But if you are interested, you are highly encouraged to implement one using some data structures, even with indexing.
- As we agreed in lectures, different groups will implement different parts of the relational algebra. The allocation is given in "Project Allocation.xls". Your language only includes the token marked by "×".

Other phases are coming soon.
To be continued...