22K-4176 | 22K-4424 | 22K-4253

LQL Language (CC Project)

1) REGEX Grouping Table (Token Classes)

Token Class	Regex/Description	Example
NUMBER	-?\d+(\.\d+)?	42, -3.14
DOLLARD	\$0	$0
ARROW	=>	=>
COMP	>=	<=
GT	>	>
LT	<	<
ASSIGN	=	=
OP	+	-
ID/KEYWORD	[a-zA-Z_][a-zA-Z0-9_]*	x, y, mylist
LBRACK/RBRACK	\[, \]	[,]
LPAREN/RPAREN	\(, \)	(,)
COMMA	,	,
COMMENT	@[^\n]*	@ comment
WS	[\t]+	space, tab
NEWLINE	\r?\n+	newline

[KEYWORDS like "List, filter, sort, etc" are recognized via ID regex but are converted to KEYWORD type in the lexer.]

2) DFA / Transition Table for lexical analysis

State	Input Class	Next State	Accept?	Token Type
S0	letter/_	S_ID	No	-
S0	Digit	S_NUM	No	-
S0	-	S_MINUSNUM	No	-
S0	$	S_DOLLAR	No	-
S0	>	S_GT	No	-
S0	<	S_LT		COMP/GT/LT
S0	=	S_ASSIGN	No	
S0	+, -, *, /, %	S_OP		OP
S0	[,], (,), ,	S_PUNC		PUNCT
S0	@	S_COMMENT	No	COMMENT
S0	whitespace/\n	S_WS	No	WS/NEWLINE

Example SUB-DFA for NUMBERS (S_NUM):

- Digits → loop in S_NUM
- . → S_DECIMAL
- S_DECIMAL + digits → S_NUM (accept as NUMBER)

Example Sub-DFA for Identifiers (S_ID):

- letter/_ → loop in S_ID.
- . → S_DECIMAL
- END → accept as ID (or keyword if matches list)

3) SAMPLE TOKEN STREAM TABLE

- For this source:

```
list mylist [1,2,3]
x = 5+2
print x
```

Token Type	Value	Line	Column
KEYWORD	list		1
ID	mylist	1	6
LBRACK	[	1	13
NUMBER	1	1	14
COMMA	,	1	15
NUMBER	2	1	17
COMMA	,	1	18
NUMBER	3	1	20
RBRACK	]	1	21
ID	x	2	1
ASSIGN	=	2	3
NUMBER	5	2	5
OP	+	2	7
NUMBER	2	2	9
KEYWORD	print	3	1
ID	x	3	7
EOF	None	3	8

4. Syntax Grammer

Program → StmtList
StmtList → Stmt StmtList | ε
Stmt → ListDecl | SortStmt | PrintStmt | FilterStmt
| MapStmt | SetStmt | ListOpStmt
ListDecl → "list" ListIdentifier "=" ListSource
ListSource → "[" Array "]" | ListIdentifier | FilterStmt |
MapStmt | SetStmt | ListOpStmt
SortStmt → "sort" ListIdentifier ("asc" | "desc")
FilterStmt → "filter" ListIdentifier ComparisonOp Number
MapStmt → "map" ListIdentifier "to" "=>" Expr
SetStmt → ListIdentifier SetOperator ListIdentifier
ListOpStmt → ListOrExpr
ListOrExpr → ListXORExpr ("or" ListXorExpr)*
ListXorExpr → ListANDExpr ("xor" ListAndExpr)*
ListAndExpr → ListAddExpr ("and" ListAddExpr)*
ListAddExpr → (ListMulExpr)*
("+" | "-")
ListMulExpr → ("*" | "/") ListPrimary
ListPrimary → ListIdentifier | Number | "[" Array "]"
| "(" ListOpStmt ")"
PrintStmt → "print" (ListIdentifier | ScalarOpExpr)
ScalarOpExpr → Operation ListIdentifier
Operation → "mean" | "sum" | "median" | "variance" | "std" | "min" |
"max" | "count"
Expr → OrExpr

OrExpr → XorExpr ("or" XorExpr)*

XorExpr → AndExpr ("xor" AndExpr)*

AndExpr → AddExpr ("and" AddExpr)*

AddExpr → MulExpr (("+" | "-") MulExpr)*

MulExpr → Primary (("*" | "/") Primary)*

Primary → Number | "$0" | "(" Expr ")" | "-" Primary

Array → Number ("," Number)* | ε

Number → "-"? Digit+ ("." Digit+)?

Digit → [0-9]

ComparisonOp → "==" | "!=" | ">" | ">=" | "<" | "<="

SetOperator → "union" | "intersection" | "difference"

ListIdentifier → [a-zA-Z_][a-zA-Z0-9_]*

Day: Date:

GRAMMAR 1

list x = [1, 2, 3]
list y = [2, 4, 5]
list res = x + y
print res

PARSE TREE:

Program
- StmtList
  - Stmt
    - ListDecl
      - "list"
      - ListIdentifier
        - "x"
      - "="
      - ListSource
        - "[" Array "]"
          - "["
          - Array
            - Number
              - "1"
            - ","
            - Number
              - "2"
            - ","
            - Number
              - "3"
          - "]"
  - StmtList
    - Stmt
      - ListDecl
        - "list"
        - ListIdentifier
          - "y"
        - "="
        - ListSource
          - "[" Array "]"
            - "["
            - Array
              - Number
                - "2"
              - ","
              - Number
                - "4"
              - ","
              - Number
                - "5"
            - "]"
    - StmtList
      - Stmt
        - ListDecl
          - "list"
          - ListIdentifier
            - "res"
          - "="
          - ListSource
            - ListOpStmt
              - ListOrExpr
                - ListXorExpr
                - ListAndExpr
                - ListAddExpr
                - ListMulExpr
                - ListPrimary
                - ListIdentifier
                - "x"
              - "+"
              - ListMulExpr
                - ListPrimary
                - ListIdentifier
                - "y"
      - StmtList
        - Stmt
          - PrintStmt
            - "Print"
            - ListIdentifier
              - "res"
        - StmtList
          - ε

Day: Date:

GRAMMAR 2:

```
list x = [1,2,3]
list res = x+2
print res
```

PARSE TREE:

```
Program
└StmtList
  ├Stmt
  │└ListDeclaration
  │ ├"list"
  │ ├ListIdentifier
  │ │└"x"
  │ ├"="
  │ └ListSource
  │   └"[" Array "]"
  │     ├"["
  │     ├Array
  │     │ ├Number
  │     │ │└"1"
  │     │ ├","
  │     │ ├Number
  │     │ │└"2"
  │     │ ├","
  │     │ └Number
  │     │   └"3"
  │     └"]"
  │
  └StmtList
    ├Stmt
    │└ListDeclaration
    │ ├"list"
    │ ├ListIdentifier
    │ │└"res"
    │ ├"="
    │ └ListSource
    │   └ListOptStmt
    │     └ListOrExpr
    │       └ListXorExpr
    │         └ListAndExpr
    │           └ListAddExpr
    │             ├ListMulExpr
    │             │└ListPrimary
    │             │  └ListIdentifier
    │             │    └"x"
    │             ├"+"
    │             └ListMulExpr
    │               └ListPrimary
    │                 └Number
    │                   └"2"
    │
    └StmtList
      ├Stmt
      │└PrintStmt
      │ ├"print"
      │ └ListIdentifier
      │   └"res"
      └StmtList
        └ε
```

- Example Code Snippt

```
list nums  = [1,2,3]
list evens = [2,4,6]
print x
```

Symbol Table

Name	Type	Scope	Value	Line Declared
nums	list	global	[1,2,3]	1
evens	list	global	[2,4,6]	2

Note: Scope is global because these variables are defined at toplevel

- Example with Nested Scope

```
list nums  [1,2,3]
map nums x => x+1
```

Symbol Table

Name	Type	Scope	Value	Line
nums	list	global	[1,2,3]	1
x	number	map-lambda	-	2

Reflection on Compiler Construction Project

During this project, I gained hands-on experience in the key phases of compiler construction: lexical analysis, syntax analysis, and semantic analysis. Implementing the lexer allowed me to understand how source code is transformed into a stream of tokens using regular expressions, and how careful handling of token priorities prevents ambiguity in parsing. Creating the parser and building parse trees reinforced the importance of grammar design and how context-free rules translate into hierarchical structures representing program logic. The semantic analysis phase, particularly building the symbol table & managing scopes, taught me the necessity of tracking type information & variable visibility for correct program interpretation.

Through this project, I also learned the value of designing modular, testable components. Each compiler phase could be developed & tested independently, which made debugging more systematic. Working on edge cases, such as nested scopes & invalid tokens, highlighted the importance of robust error handling & clear feedback for users.

If I were to improve this project, I would focus on enhancing the parser to support more complex expressions & nested statements, and implement type checking & runtime

evaluation for expressions to make the compiler closer to a fully functional interpreter. Additionally integrating visualizations for token streams, parse trees, and symbol tables would improve understanding & documentation. Finally, better automation of tests for all phases would save time & ensure correctness for larger source code inputs.

Overall, this project significantly strengthened my understanding of compiler theory & practical implementation, bridging gap b/w formal language concepts and real-world programming tools. I feel more confident now in designing language processors & ~~ang~~ analyzing the structure & semantics of code systematically