Homework Assignment

Implementing a Scanner, Parser, and AST for a Google Query Language

Introduction

In this assignment, you will implement basic components of a compiler for a small, domain-specific language designed to express structured Google search queries using keywords, directives, and set operations.

The language supports the declaration of queries, conditional execution, loops over query lists, and assignment of query results. Students will deepen their understanding of tokenization (lexical analysis), parsing (syntax analysis), and abstract syntax tree (AST) construction for context-free grammars with operator precedence.

Language Summary

The language allows you to:

- Declare queries with keywords, directives (key-value pairs), and logical operations.
- Compose query sets using set operations: union ("++"), difference ("--"), intersection ("**").
- Control execution with "IF" conditions and "FOR" loops.

Query structures are enclosed in '<' and '>' symbols to clearly mark their beginning and end. Terms inside a query can be simple words, directives, or combined using logical operators.

Operator precedence must be respected:
- Juxtaposition (space between terms) binds stronger than '|'.

Students are expected to study and properly handle precedence and associativity in their parser.

Grammar (Shortened Form)

```
program -> declarations commands

declarations -> declaration
          | declarations declaration

declaration -> "QUERY" query_name "=" query ";"
          | "QUERY" query_name "=" list_of_queries ";"
          | "RESULT_OF_QUERY" identifier ";"
```

```
commands -> command
        | commands command

command -> "EXEC" query_name ";"
        | "IF" condition "BEGIN" commands "END"
        | "FOR" identifier "IN" list_of_queries "BEGIN" commands "END"
        | assign_command ";"

assign_command -> identifier "=" "EXEC" query_name ";"
             | identifier "=" identifier set_operator identifier

condition -> "EMPTY" "(" identifier ")"
          | "URL_EXISTS" "(" identifier "," STRING ")"
          | "NOT_EMPTY" "(" identifier ")"

list_of_queries -> "[" query_list "]"

query_list -> query
           | query_list "," query

query -> "<" terms ">"

terms -> term
      | terms term

term -> TERM
    | directive
    | operator term
    | term "|" term

operator -> "+"
         | "-"
         | "*"

set_operator -> "++"
             | "--"
             | "**"

directive -> KEY ":" VALUE

TERM -> WORD | STRING
```

identifier -> WORD
KEY -> WORD
VALUE -> WORD | STRING
STRING -> "string literal"
WORD -> [a-zA-Z0-9_]+

Tasks

1. Implement a Scanner (Lexer)

- Write a scanner that reads the source text and converts it into a stream of tokens.
- The scanner must correctly recognize keywords (QUERY, EXEC, etc.), operators (+, -, *, |, ++, --, **), brackets (<, >, [,], (,)), identifiers, and string literals.
- Handle spaces and comments (if any) appropriately.

2. Implement a Parser

- Write a parser that builds a parse tree according to the grammar rules.
- Pay special attention to operator precedence:
  - Juxtaposition (space) binds stronger than the '|' (alternative operator).
- Report syntax errors clearly if the input does not conform to the grammar.

Hint: You may choose between hand-writing the parser (recursive descent) or using parser generator tools (e.g., ANTLR, Yacc/Bison).

3. Build an Abstract Syntax Tree (AST)

- Define AST node classes for all important constructs: Program, Declarations, Commands, Queries, Terms, etc.
- The AST should represent the logical structure of the program, not the syntactic details (e.g., no brackets or separators in the tree).
- Implement the parser so that instead of building a raw parse tree, it constructs the corresponding AST during parsing.

Additional Notes

- Respect operator precedence and associativity when building the AST.
- You may optionally extend the language to support parentheses '()' inside queries for custom grouping (bonus points).
- Think carefully about the distinction between tokens, syntax structure, and abstract structure (AST).

Example

Example Program:

QUERY basic = < apple banana | orange >;
QUERY advanced = <filetype:pdf apple +fruit -tree >;

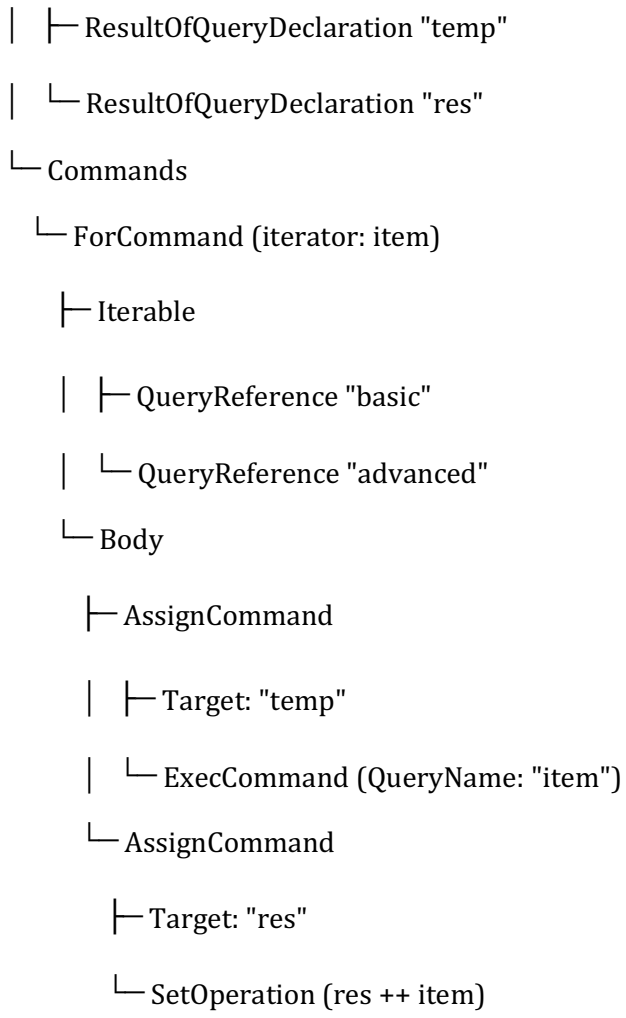
```
RESULT_OF_QUERY temp;
RESULT_OF_QUERY res;


FOR item IN [basic, advanced] BEGIN
  temp = EXEC item;
  res = res ++ item;
END
```

Corresponding AST Structure:

AST Structure (simplified):

```
Program
├─ Declarations
│  ├─ QueryDeclaration "basic"
│  │  └─ Query
│  │     └─ Or
│  │        ├─ Juxtaposition
│  │        │  ├─ Term "apple"
│  │        │  └─ Term "banana"
│  │        └─ Term "orange"
│  ├─ QueryDeclaration "advanced"
│  │  └─ Query
│  │     └─ Juxtaposition
│  │        ├─ Directive (filetype:pdf)
│  │        ├─ Term "apple"
│  │        ├─ Plus(Term "fruit")
│  │        └─ Minus(Term "tree")
```


├─ ResultOfQueryDeclaration "temp"
└─ ResultOfQueryDeclaration "res"
└─ Commands
└─ ForCommand (iterator: item)
├─ Iterable
├─ QueryReference "basic"
└─ QueryReference "advanced"
└─ Body
├─ AssignCommand
├─ Target: "temp"
└─ ExecCommand (QueryName: "item")
└─ AssignCommand
├─ Target: "res"
└─ SetOperation (res ++ item)