**main.py**

```python
"""
Deterministic Finite Automata (DFA) and Non-deterministic Finite Automata (NFA) Overview:

In language parsing, DFAs and NFAs are widely used for lexing and recognizing patterns within input code. A DFA has
a strict, single-path nature from one state to the next for each character input, meaning there is only one possible
transition for each character in each state. DFAs are fast and efficient as they don't need to backtrack, which makes
them ideal for predictable, structured patterns like keywords and operators.

On the other hand, NFAs are more flexible in that they can transition to multiple states at once based on a single
input, or even none (epsilon transitions). This enables NFAs to handle ambiguous or overlapping patterns more easily.
In practical applications, NFAs are often converted to DFAs, as DFAs are more performant, though NFAs can provide a
more compact representation. Both automata are implemented here to parse an extensive language grammar, giving it
flexibility in handling various constructs.
"""

import re

# DFA for parsing tokens
class DFA:
    def __init__(self):
        self.states = {}
        self.final_states = set()
        self.current_state = None

    def add_state(self, name, is_final=False):
        self.states[name] = {}
        if is_final:
            self.final_states.add(name)

    def add_transition(self, from_state, input_char, to_state):
        if from_state in self.states:
            self.states[from_state][input_char] = to_state

    def set_start_state(self, state):
        self.current_state = state

    def process_input(self, input_string):
        for char in input_string:
            if char in self.states[self.current_state]:
                self.current_state = self.states[self.current_state][char]
            else:
                return False
```

```
        return self.current_state in self.final_states

# NFA for parsing tokens
class NFA:
    def __init__(self):
        self.states = {}
        self.start_state = None
        self.final_states = set()

    def add_state(self, name, is_final=False):
        self.states[name] = {}
        if is_final:
            self.final_states.add(name)

    def add_transition(self, from_state, input_char, to_states):
        if from_state in self.states:
            self.states[from_state].setdefault(input_char, []).extend(to_states)

    def set_start_state(self, state):
        self.start_state = state

    def process_input(self, input_string, current_states=None):
        if current_states is None:
            current_states = {self.start_state}
        for char in input_string:
            next_states = set()
            for state in current_states:
                if char in self.states[state]:
                    next_states.update(self.states[state][char])
            current_states = next_states
        return bool(current_states & self.final_states)


dfa = DFA()
dfa.set_start_state("START")
dfa.add_state("KEYWORD", is_final=True)
dfa.add_state("NUMBER", is_final=True)
dfa.add_state("IDENTIFIER", is_final=True)

GRAMMAR_RULES = [
    "S -> if E then S",
    "S -> while E do S",
    "S -> for IDENTIFIER in E do S",
    "S -> S ; S",
    "E -> E + E",
    "E -> E * E",
    "E -> E - E",
    "E -> E / E",
    "E -> ( E )",
    "E -> NUMBER",
```

```
    "E -> IDENTIFIER",
    "E -> true",
    "E -> false",
    "E -> IDENTIFIER == IDENTIFIER",
    "E -> IDENTIFIER != IDENTIFIER",
    "E -> IDENTIFIER >= IDENTIFIER",
    "E -> IDENTIFIER <= IDENTIFIER",
    "E -> IDENTIFIER < IDENTIFIER",
    "E -> IDENTIFIER > IDENTIFIER",
    "S -> IDENTIFIER = E",
    "S -> IDENTIFIER = NUMBER",
    "S -> return E",
    "S -> function IDENTIFIER ( PARAMS ) { BODY }",
    "PARAMS -> IDENTIFIER",
    "PARAMS -> PARAMS , IDENTIFIER",
    "BODY -> S",
    "BODY -> BODY S",
    "S -> { S }",
    "S -> IDENTIFIER ( ARG_LIST )",
    "ARG_LIST -> E",
    "ARG_LIST -> ARG_LIST , E",
    "S -> if E then S else S",
    "S -> break",
    "S -> continue",
    "S -> IDENTIFIER [ E ] = E",
    "S -> for IDENTIFIER = E to E do S",
    "S -> while ( E ) S",
    "S -> do S while ( E )",
    "S -> switch ( E ) { CASES }",
    "CASES -> case NUMBER : S",
    "CASES -> case IDENTIFIER : S",
    "CASES -> CASES case NUMBER : S",
    "S -> var IDENTIFIER = E",
    "S -> let IDENTIFIER = E",
    "S -> const IDENTIFIER = E",
    "E -> ! E",
    "E -> - E",
    "S -> import IDENTIFIER from STRING",
    "S -> export IDENTIFIER",
    "E -> IDENTIFIER ? E : E",
]


nfa = NFA()
nfa.set_start_state("START")
nfa.add_state("EXPR", is_final=True)
nfa.add_state("TERM")

def parse_input(input_string):
    if dfa.process_input(input_string):
```

```
        print("DFA accepted the input.")
    elif nfa.process_input(input_string):
        print("NFA accepted the input.")
    else:
        print("Input rejected by both DFA and NFA.")


import re

# Define token types using regular expressions
TOKEN_SPECIFICATION = [
    # Single-line comments
    ('COMMENT', r'//[^\n]*'),
    ('KEYWORD', r'\b(if|else|while|for|return|function)\b'),  # Added 'function'
    # Integer or decimal number
    ('NUMBER', r'\d+(\.\d*)?'),
    ('STRING', r'"[^"]*"'),                                   # String literal
    ('BOOLEAN', r'\b(true|false)\b'),                         # Boolean values
    ('IDENTIFIER', r'[A-Za-z_][A-Za-z_0-9]*'),                # Identifiers
    ('BOOL_OP', r'\b(and|or|not)\b'),                         # Boolean operators
    # Boolean comparisons
    ('BOOL_COMP', r'==|!=|<=|>=|<|>'),
    # Assignment operator
    ('ASSIGN', r'='),
    ('OPERATOR', r'[+\-*/%&|]+'),                             # Operators
    ('LPAREN', r'\('),                                        # Left parenthesis
    # Right parenthesis
    ('RPAREN', r'\)'),
    ('LBRACE', r'\{'),                                        # Left brace
    ('RBRACE', r'\}'),                                        # Right brace
    ('SEMICOLON', r';'),                                      # Semicolon
    ('COLON', r':'),                                          # Colon
    ('COMMA', r','),                                          # Comma
    # Skip over spaces and tabs
    ('WHITESPACE', r'[ \t]+'),
    ('NEWLINE', r'\n'),                                       # Line endings
]

# Compile regular expressions
TOKENS_RE = '|'.join(f'(?P<{pair[0]}>{pair[1]})' for pair in TOKEN_SPECIFICATION)


class ParseTreeNode:
    def __init__(self, node_type, value=None):
        self.node_type = node_type  # Type of the node
        self.value = value          # Value of the node
        self.children = []          # Child nodes

    def add_child(self, child_node):
        self.children.append(child_node)
```

```
    def __repr__(self):
        return f'{self.node_type}({self.value})'


# Tokenizer
class Token:
    def __init__(self, type, value, line, column):
        self.type = type
        self.value = value
        self.line = line
        self.column = column

    def __repr__(self):
        return f'{self.type}({self.value})'


def tokenize(code):
    line_num = 1
    line_start = 0
    tokens = []
    for match in re.finditer(TOKENS_RE, code):
        kind = match.lastgroup
        value = match.group(kind)
        column = match.start() - line_start
        if kind == 'WHITESPACE' or kind == 'COMMENT':
            continue
        elif kind == 'NEWLINE':
            line_num += 1
            line_start = match.end()
        else:
            tokens.append(Token(kind, value, line_num, column))
    return tokens


# Parser for Control Structures and expressions
class ControlStructureParser:
    DATA_TYPES = [
        'int', 'float', 'double', 'char', 'string', 'bool', 'void', 'byte',
        'short', 'long', 'decimal', 'object', 'list', 'set', 'map', 'array',
        'function', 'date', 'time', 'datetime', 'buffer', 'stream', 'enum',
        'struct', 'class', 'interface', 'tuple', 'json', 'xml', 'html',
        'url', 'path', 'regex', 'pointer', 'reference', 'native', 'async',
        'generator', 'promise', 'callback', 'task', 'module'
    ]

    def __init__(self, tokens):
        self.tokens = tokens
        self.pos = 0
        self.scope = {}
```

```
243
244     def peek(self):
245         return self.tokens[self.pos] if self.pos < len(self.tokens) else None
246
247     def advance(self):
248         token = self.peek()
249         self.pos += 1
250         return token
251
252     def expect(self, token_type):
253         token = self.advance()
254         if token is None or token.type != token_type:
255             raise ParserError(f'Expected {token_type}, got {token.type if token else "EOF"}')
256         return token
257
258     def parse_while_statement(self):
259         self.expect('KEYWORD')  # "while"
260         self.expect('LPAREN')
261         self.parse_boolean_expression()
262         self.expect('RPAREN')
263         self.expect('LBRACE')
264         self.parse_statement_list()
265         self.expect('RBRACE')
266
267     def parse_if_statement(self):
268         self.expect('KEYWORD')  # "if"
269         self.expect('LPAREN')
270         self.parse_boolean_expression()
271         self.expect('RPAREN')
272         self.expect('LBRACE')
273         self.parse_statement_list()
274         self.expect('RBRACE')
275
276         # Check for optional 'else' block
277         if self.peek() and self.peek().value == 'else':
278             self.advance()  # "else"
279             if self.peek() and self.peek().type == 'KEYWORD' and self.peek().value == 'if':
280                 self.parse_if_statement()  # Else-if (nested if)
281             else:
282                 self.expect('LBRACE')
283                 self.parse_statement_list()
284                 self.expect('RBRACE')
285
286     def parse_statement_list(self):
287         while self.peek() and self.peek().type != 'RBRACE':
288             self.parse_statement()
289
290     def parse_statement(self):
291         token = self.peek()
292         if token.type == 'COMMENT':
```

```
            self.advance()  # Skip comments
        if token.type == 'KEYWORD' and token.value == 'if':
            self.parse_if_statement()
        elif token.type == 'KEYWORD' and token.value == 'while':
            self.parse_while_statement()
        elif token.type == 'IDENTIFIER' and token.value in self.DATA_TYPES:
            self.parse_variable_declaration()
        elif token.type == 'KEYWORD' and token.value == 'function':
            self.parse_function_declaration()
        else:
            raise ParserError(f'Unexpected token {token.value} in statement')

    def parse_function_declaration(self):
        self.expect('KEYWORD')  # "function"
        func_name = self.expect('IDENTIFIER')  # Function name
        self.expect('LPAREN')  # Opening parenthesis

        # Parse parameters
        params = []
        while self.peek() and self.peek().type != 'RPAREN':
            param_type = self.expect('IDENTIFIER')  # Expect a type
            param_name = self.expect('IDENTIFIER')  # Expect a variable name
            params.append((param_type.value, param_name.value))

            if self.peek() and self.peek().type == 'COMMA':
                self.advance()  # Consume comma

        self.expect('RPAREN')  # Closing parenthesis
        self.expect('LBRACE')  # Opening brace for function body
        self.parse_statement_list()  # Parse function body
        self.expect('RBRACE')

        print(f"Function {func_name.value} declared with parameters: {params}")
    def parse_variable_declaration(self):
        data_type_token = self.expect('IDENTIFIER')
        if data_type_token.value not in self.DATA_TYPES:
            raise ParserError(f'Unknown data type: {data_type_token.value}')

        var_name = self.expect('IDENTIFIER')
        if self.peek().type == 'ASSIGN':
            self.advance()
            self.parse_expression()

        self.expect('SEMICOLON')

    def parse_expression(self):
        return self.parse_assignment()

    def parse_assignment(self):
        """Handles variable assignment and return expressions."""
```

```
        left = self.parse_equality()
        if self.peek() and self.peek().type == 'ASSIGN':
            self.expect('ASSIGN')
            right = self.parse_assignment()  # Right side of assignment
            return ParseTreeNode('ASSIGNMENT', (left, right))  # Return an assignment node
        return left

    def parse_equality(self):
        """Handles equality comparisons (==, !=)."""
        left = self.parse_comparison()
        while self.peek() and self.peek().type in ('BOOL_COMP'):
            operator = self.advance()
            right = self.parse_comparison()
            left = ParseTreeNode('EQUALITY', (left, operator, right))  # Create a node for equality
        return left

    def parse_comparison(self):
        """Handles comparison operations (<, <=, >, >=)."""
        left = self.parse_term()
        while self.peek() and self.peek().type in ('BOOL_COMP'):
            operator = self.advance()
            right = self.parse_term()
            left = ParseTreeNode('COMPARISON', (left, operator, right))  # Create a node for comparison
        return left

    def parse_term(self):
        """Handles addition and subtraction."""
        left = self.parse_factor()
        while self.peek() and self.peek().type in ('OPERATOR'):
            operator = self.advance()
            right = self.parse_factor()
            left = ParseTreeNode('TERM', (left, operator, right))  # Create a node for term
        return left

    def parse_factor(self):
        """Handles multiplication and division."""
        left = self.parse_unary()
        while self.peek() and self.peek().type in ('OPERATOR'):
            operator = self.advance()
            right = self.parse_unary()
            left = ParseTreeNode('FACTOR', (left, operator, right))  # Create a node for factor
        return left

    def parse_unary(self):
        """Handles unary operators (!, -)."""
        if self.peek() and self.peek().type == 'OPERATOR':
            operator = self.advance()
            operand = self.parse_unary()
```

```
            return ParseTreeNode('UNARY', (operator, operand))  # Create a node for unary operation
        return self.parse_primary()

    def parse_primary(self):
        """Handles primary expressions (numbers, identifiers, and parenthesized expressions)."""
        if self.peek().type == 'NUMBER':
            return ParseTreeNode('NUMBER', self.expect('NUMBER').value)
        elif self.peek().type == 'IDENTIFIER':
            return ParseTreeNode('IDENTIFIER', self.expect('IDENTIFIER').value)
        elif self.peek().type == 'LPAREN':
            self.expect('LPAREN')
            expr = self.parse_expression()
            self.expect('RPAREN')
            return expr
        else:
            raise ParserError('Expected primary expression')

    # Example of a simple statement parser
    def parse_statement_list(self):
        """Parse a list of statements."""
        while self.peek() and self.peek().type != 'RBRACE':
            self.parse_statement()

    def parse_statement(self):
        """Parse a single statement, like an assignment or function call."""
        if self.peek().type == 'IDENTIFIER':
            identifier = self.expect('IDENTIFIER')
            if self.peek() and self.peek().type == 'ASSIGN':
                # Handle assignment
                self.expect('ASSIGN')
                expr = self.parse_expression()
                return ParseTreeNode('ASSIGNMENT', (identifier.value, expr))
            elif self.peek() and self.peek().type == 'LPAREN':
                # Handle function call
                self.expect('LPAREN')
                args = self.parse_arguments()
                self.expect('RPAREN')
                return ParseTreeNode('FUNCTION_CALL', (identifier.value, args))
        elif self.peek().type == 'KEYWORD':
            if self.peek().value == 'if':
                return self.parse_if_statement()
            elif self.peek().value == 'while':
                return self.parse_while_statement()
        # Add more statement types as needed
        raise ParserError('Invalid statement')

    def parse_arguments(self):
        """Parse function call arguments."""
```

```
        args = []
        while self.peek() and self.peek().type != 'RPAREN':
            args.append(self.parse_expression())
            if self.peek() and self.peek().type == 'COMMA':
                self.expect('COMMA')
        return args

    def parse_boolean_expression(self):
        """Parse a boolean expression."""
        left = self.parse_expression()
        if self.peek() and self.peek().type == 'BOOL_COMP':
            operator = self.advance()
            right = self.parse_expression()
            return ParseTreeNode('BOOLEAN_EXPRESSION', (left, operator, right))
        return left




    def parse(self):
        while self.peek():
            self.parse_statement()

    GRAMMAR_RULES = [
        "RULE1: if_statement -> 'if' '(' condition ')' '{' statement '}'",
        "RULE2: while_statement -> 'while' '(' condition ')' '{' statement '}'",
    ]


class ParserError(Exception):
    pass


# Example Usage
if __name__ == "__main__":
    code = open('x.A', 'r', encoding="utf-8").read()

    tokens = tokenize(code)
    print("Tokens:")
    print(tokens)

    parser = ControlStructureParser(tokens)
    try:
        parser.parse_statement_list()
        print("Parsing completed successfully.")
    except ParserError as e:
        pass

```