```
import re

class Tokenizer:
    def __init__(self, program):
        self.tokens = self.tokenize(program)
        self.position = 0

    def tokenize(self, program):
        token_specification = [
            ('NUMBER', r'\d+'),  # Integer
            ('STRING', r'"([^"\\]|\\.)*"'),  # String
            ('CHARACTER', r"'([^'\\]|\\.)*'"),  # Character
            ('ID', r'[a-zA-Z_][a-zA-Z0-9_]*'),  # Identifiers
            ('FUN_KEYWORD', r'fun'),  # Function keyword
            ('IF_KEYWORD', r'if'),  # If keyword
            ('ELSE_KEYWORD', r'else'),  # Else keyword
            ('WHILE_KEYWORD', r'while'),  # While keyword
            ('FOR_KEYWORD', r'for'),  # For keyword
            ('MATCH_KEYWORD', r'match'),  # Match keyword
            ('CASE_KEYWORD', r'case'),  # Case keyword
            ('INPUT_KEYWORD', r'input'),  # Input keyword
            ('OUTPUT_KEYWORD', r'output'),  # Output keyword
            ('RETURN_KEYWORD', r'return'),  # Return keyword
            ('NOTHING_KEYWORD', r'nothing'),  # Nothing keyword
            ('TRUE_KEYWORD', r'true'),  # True keyword
            ('FALSE_KEYWORD', r'false'),  # False keyword
            ('LIST_KEYWORD', r'list'),  # List keyword
            ('DICTIONARY_KEYWORD', r'dictionary'),  # Dictionary keyword
            ('STRUCTURE_KEYWORD', r'structure'),  # Structure keyword
            ('INTEGER_KEYWORD', r'integer'),  # Integer keyword
            ('TEXT_KEYWORD', r'text'),  # Text keyword
            ('BOOLEAN_KEYWORD', r'boolean'),  # Boolean keyword
            ('CHARACTER_KEYWORD', r'character'),  # Character keyword
            ('FIXED_KEYWORD', r'fixed'),  # Fixed keyword
            ('TRY_KEYWORD', r'try'),  # Try keyword
            ('CATCH_KEYWORD', r'catch'),  # Catch keyword
            ('THROW_KEYWORD', r'throw'),  # Throw keyword
            ('GO_KEYWORD', r'go'),  # Go keyword
            ('ASSIGN', r'=|\+=|\-=|\*=|/=|\*\*=|='),  # Assignment and compound assignments
            ('COMMA', r','),  # Comma
            ('SEMI', r';'),  # Semicolon
            ('COLON', r':'),  # Colon
            ('PAREN_OPEN', r'\('),  # Opening parenthesis
            ('PAREN_CLOSE', r'\)'),  # Closing parenthesis
            ('BRACE_OPEN', r'\{'),  # Opening brace
            ('BRACE_CLOSE', r'\}'),  # Closing brace
            ('ADD', r'\+'),  # Addition
            ('SUB', r'-'),  # Subtraction
            ('MUL', r'\*'),  # Multiplication
            ('DIV', r'/'),  # Division
            ('MOD', r'%'),  # Remainder
            ('POWER', r'\*\*'),  # Exponentiation
            ('REL_OP', r'<=|>=|<|>|==|!='),  # Relational operators
            ('SKIP', r'[ \t\n\r]+'),  # Skip spaces and tabs
            ('MISMATCH', r'.'),  # Any other character
```

```
        ]
        tok_regex = '|'.join(f'(?P<{pair[0]}>{pair[1]})' for pair in token_specification)
        tokens = []
        line_num = 1
        line_start = 0
        for match in re.finditer(tok_regex, program):
            kind = match.lastgroup
            value = match.group()
            column = match.start() - line_start + 1
            if '\n' in value:
                line_num += value.count('\n')
                line_start = match.end() - value.rfind('\n')  # Start of new line
            if kind == 'SKIP':
                continue
            elif kind == 'MISMATCH':
                raise SyntaxError(f"Unexpected character: {value} at line {line_num}, column {column}")
            else:
                tokens.append((kind, value, line_num, column))
        return tokens

    def next_token(self):
        if self.position < len(self.tokens):
            token = self.tokens[self.position]
            self.position += 1
            return token
        else:
            return None


# Load the grammar from a file with error handling for empty or malformed lines
def load_grammar(filename):
    grammar = {}
    with open(filename, 'r') as file:
        for line in file:
            line = line.strip()
            if not line:  # Skip empty lines
                continue
            if ' : ' not in line:  # Skip lines that do not contain ' : '
                print(f"Warning: Skipping malformed line: {line}")
                continue
            rule_name, rule_def = line.split(' : ', 1)  # Ensure we handle spaces correctly
            grammar[rule_name.strip()] = rule_def.strip().split()
    return grammar


# Recursive descent parser
class Parser:
    def __init__(self, tokenizer, grammar):
        self.tokenizer = tokenizer
        self.grammar = grammar
        self.current_token = self.tokenizer.next_token()

    def parse(self, rule_name):
        rule = self.grammar.get(rule_name, [])
        if not rule:
```

```
            raise ValueError(f"Rule '{rule_name}' not found in grammar")

        for token in rule:
            if token.isupper():
                # If token is non-terminal, recursively parse it
                if not self.parse(token):
                    return False
            elif not self.match(token):
                return False
        return True

    def match(self, expected_token):
        if self.current_token and self.current_token[0] == expected_token:
            print(f"Matched {self.current_token[0]} ({self.current_token[1]})")
            self.current_token = self.tokenizer.next_token()
            return True
        else:
            if self.current_token:
                token_type, token_value, token_line, token_column = self.current_token
                print(f"Syntax error: Expected {expected_token}, found {token_type} ({token_value}) at line
{token_line}, column {token_column}")
            return False

    def parse_program(self):
        # Expect 'fun main' at the start of the program
        if not self.match('FUN_KEYWORD'):  # Expect FUN_KEYWORD
            print("Syntax error: Expected 'fun' at the start of the program.")
            return False
        if not self.match('ID'):  # Expect function name (i.e., main)
            print("Syntax error: Expected function name after 'fun'.")
            return False
        if not self.match('PAREN_OPEN'):  # Expect '('
            print("Syntax error: Expected '(' after function name.")
            return False
        if not self.match('PAREN_CLOSE'):  # Expect ')'
            print("Syntax error: Expected ')' after function parameters.")
            return False
        if not self.match('BRACE_OPEN'):  # Expect '{'
            print("Syntax error: Expected '{' after function declaration.")
            return False
        # Parse the body of the main function here (e.g., statements)
        if not self.parse_statements():  # Parse statements inside the function
            print("Syntax error: Expected statements inside the function body.")
            return False
        if not self.match('BRACE_CLOSE'):  # Expect '}'
            print("Syntax error: Expected '}' to close function body.")
            return False
        return True

    def parse_statements(self):
        # A simple rule to accept one statement
        if not self.match('ID'):  # Expect some form of statement (e.g., variable assignment)
            return False
        return True
```

```
if __name__ == "__main__":
    # Load grammar from file
    grammar = load_grammar('grammar.txt')

    # Sample input program
    program = """
    fun main() -> nothing {
        x = 5;
        y = 10;
        if (x < y) {
            x = x + 1;
        }
    }
    """

    # Initialize tokenizer and parser
    tokenizer = Tokenizer(program)
    parser = Parser(tokenizer, grammar)

    # Parse the input program
    if parser.parse_program():
        print("Input program is valid")
    else:
        print("Syntax error in program")
```