CSE 307: Principles of Programming Languages
Assignment I

Lexer and Parser for MiniML

Dr. Ritwik Banerjee
Computer Science, Stony Brook University

1 Overview and Learning Objectives

In this assignment, you will implement a lexical analyzer (**lexer**) and a **parser** for MiniML, a simplified subset of the OCaml programming language. This is the first step toward building a complete interpreter for a functional programming language. Your lexer will convert source code into tokens, and your parser will convert those tokens into an Abstract Syntax Tree (AST). This assignment directly applies concepts from the first six or seven lectures, including syntax specification, lexical analysis, and parsing techniques. You will use Python (3.10+) to implement your solution, but please remember that this course is about programming language principles, not Python syntax. So, as and when needed, you should conduct your own research on Python syntax.

By completing this assignment, you will:

- understand how to design and implement a lexer using regular expressions and finite automata concepts;
- gain practical experience with recursive descent parsing;
- learn to construct Abstract Syntax Trees from source code;
- apply formal grammar specifications to implement a working parser; and
- handle operator precedence and associativity in parsing.

2 The MiniML Language

MiniML is a subset of the OCaml programming language that includes the following features:

2.1 Literals

- Integers: Sequence of digits, e.g., 42, 0, 123
- Booleans: true and false

2.2 Identifiers

Identifiers start with a lowercase letter followed by any combination of letters, digits, or underscores. For example: `x`, `my_var`, `counter2`.

2.3 Operators

Arithmetic operators (in order of precedence):

- `*` (multiplication), / (division); these are left-associative
- `+` (addition), `-` (subtraction); these are left-associative

Comparison operators:

- `=` (equality), `<>` (inequality), `<`, `>`, `<=`, `>=`

Boolean operators:

- `&&` (logical AND, left-associative)
- `||` (logical OR, left-associative)

Unary operators:

- `not` (logical NOT)
- `-` (unary minus, i.e., negation)

2.4 Keywords

- `let`, `in`, `if`, `then`, `else`, `fun`, `rec`

2.5 Expressions

- Let bindings: `let x = 5 in x + 1;;`
- Conditional expressions: `if x < 0 then -x else x;;`
- Function definitions: `fun x -> x + 1`
- Recursive functions: `let rec fact n = if n = 0 then 1 else n * fact (n - 1);;`
- Function application: `f x` or `f (x + 1)`
- Parenthesized expressions: `(2 + 3) * 4`

3 Grammar Specification

Your parser should implement the following context-free grammar for MiniML. Note that this grammar is already designed to handle operator precedence correctly.

```
program → expr
expr → let_expr | if_expr | fun_expr | or_expr
let_expr → LET [REC] ID [ID]* = expr IN expr
if_expr → IF expr THEN expr ELSE expr
fun_expr → FUN ID [ID]* -> expr
or_expr → and_expr (|| and_expr)*
and_expr → comp_expr (&& comp_expr)*
comp_expr → add_expr [comp_op add_expr]
comp_op → = | <> | < | > | <= | >=
add_expr → mult_expr ((+ | -) mult_expr)*
mult_expr → unary_expr ((* | /) unary_expr)*
unary_expr → (NOT | -) unary_expr | app_expr
app_expr → primary_expr primary_expr*
primary_expr → INT | BOOL | ID | ( expr )
```

4 Token Specification

Your lexer must recognize the following tokens:

Token Type	Pattern	Examples
INT	`[0-9]+`	0, 99, 1204
BOOL	`true|false`	`true`, `false`
ID	`[a-z][a-zA-Z0-9_]*`	`x`, `my_var`, `counter2`
Keywords	Reserved words	`let`, `in`, `if`, `then`, `else`, `fun`, `rec`, `not`
Operators	Various	`+`, `-`, `*`, `/`, `=`, `<>`, `<`, `>`, `<=`, `>=`, `&&`, `||`, `->`
Delimiters	Various	`(`, `)`

Important: Keywords must be recognized *before* identifiers. For example, `let` is a keyword, not an identifier, but the token `letter` in a program is an identifier.

5 Implementation Requirements

5.1 File Structure

- `token.py` – Token class definition
- `lexer.py` – Lexical analyzer implementation
- `ast_nodes.py` – AST node class definitions
- `parser.py` – Parser implementation
- `main.py` – Main driver program

5.2 Token Class

Your Token class should minimally include:

- `type`: The token type (e.g., INT, ID, PLUS)
- `value`: The actual lexeme (e.g., '42', 'x', '+')
- `line`: Line number in the source code
- `column`: Column number in the source code

5.3 Lexer Implementation

Your lexer must:

- Accept a string of MiniML source code as input
- Produce a list of tokens
- Skip whitespace (spaces, tabs, newlines) and comments
- Handle single-line comments starting with `(*` and ending with `*)`
- Report meaningful error messages for invalid tokens, including line and column numbers
- Distinguish between keywords and identifiers correctly

Note: In OCaml, even multi-line comments have this syntax. But, for this assignment, we are restricting ourselves to this simplication. You do not need to handle multi-line comments in this assignment.

5.4 AST Node Classes

Define classes for each type of AST node. At the very least, you should have:

- `IntLiteral`: Integer literals
- `BoolLiteral`: Boolean literals
- `Variable`: Variable references
- `BinaryOp`: Binary operations (arithmetic, comparison, boolean)
- `UnaryOp`: Unary operations (`not` and unary `-`)
- `Let`: Let bindings (both simple and recursive). Should store the function name, a list of parameter names, the bound expression, and the body expression.
- `If`: Conditional expressions

- `Fun`: Function definitions. Should store a list of parameter names and the function body.
- `App`: Function applications

Each AST node class should have a `__repr__` method that provides a readable string representation of the node and its children.

5.5 Parser Implementation

Your parser must:

- Accept a list of tokens from the lexer
- Implement recursive descent parsing based on the provided grammar
- Build and return an Abstract Syntax Tree
- Handle operator precedence and associativity correctly
- Report meaningful syntax errors with line and column information
- Detect and report unexpected end-of-input errors

5.6 Main Program

Your `main.py` should:

- Read MiniML source code from a file (filename provided as command-line argument)
- Invoke the lexer to tokenize the input
- Invoke the parser to build the AST
- Print the resulting AST in a readable format
- Handle and display any lexical or syntax errors gracefully

6 Tests

Your implementation must correctly parse the following test cases. Create test files for each and verify your output.

Test 1: Simple Arithmetic

- `2 + 3 * 4`
- Expected AST: `BinaryOp('+', IntLiteral(2), BinaryOp('*', IntLiteral(3), IntLiteral(4)))`

Test 2: Let Binding

- `let x = 5 in x + 1`

Test 3: Conditional

- `if x < 0 then -x else x`

Test 4: Function Definition and Application

- `let f = fun x -> x + 1 in f 5`

Test 5: Recursive Function

- `let rec factorial n = if n = 0 then 1 else n * factorial (n - 1) in factorial 5`

Test 6: Multi-parameter Function

- `let add x y = x + y in add 3 4`

Test 7: Boolean Logic

- `(true && false) || not false`

Test 8: Complex Expression

- `let rec power x n = if n = 0 then 1 else x * power x (n - 1) in power 2 8`

7 Error Handling

Your implementation must handle errors gracefully. For each error, your program should print:

- Error type (Lexical Error or Syntax Error, as described below)
- Line and column number
- A descriptive message explaining the problem

Lexical Errors

- Unrecognized characters
- Unterminated comments

Syntax Errors

- Missing tokens (e.g., missing then in if expression)
- Unexpected tokens
- Mismatched parentheses
- Invalid operator usage

Grading Rubric

COMPONENT	POINTS
Lexer Implementation	**35**
Correct tokenization of all token types	15
Proper handling of keywords vs identifiers	5
Whitespace and comment handling	5
Line and column tracking	5
Meaningful lexical error messages	5
Parser Implementation	**45**
Correct parsing of expressions with proper precedence	15
Correct parsing of let bindings (simple and recursive)	10
Correct parsing of conditionals and functions	10
Function application parsing	5
Meaningful syntax error messages	5
AST Design and Implementation	**10**
Well-designed AST node classes	5
Proper `__repr__` methods for debugging	5
Code Quality and Style	**10**
Clean, readable, well-organized code	5
Appropriate comments and documentation	5
Total	**100**

8 Hints and Tips

- Start early! This assignment requires careful attention to detail.
- Implement and test the lexer thoroughly before starting the parser.
- Test each grammar rule individually as you implement it.
- Use a debugger (or at least use print statements) to understand how your parser is making decisions.
- Pay special attention to operator precedence: test with expressions like `2 + 3 * 4` and `(2 + 3) * 4`
- For left-associative operators, consider how `1 - 2 - 3` should be parsed as `(1 - 2) - 3`, and not `1 - (2 - 3)`
- The expression `f x y` should be parsed as `(f x) y`, not `f (x y)`. This means you apply `f` to `x` first, then apply the result to `y`.
- For multi-parameter functions like `fun x y -> x + y` or `let add x y = x + y`, store the parameters as a list in your AST nodes. You do not need to convert them to nested single-parameter functions.
- Use the Ed Discussion forum to ask questions, but don't share code!
- Make sure to handle edge cases like empty input, missing closing parentheses, etc.

Academic Integrity

This assignment must be completed individually. You may discuss general concepts and strategies with your colleagues, but you may not share code, look at another student's code, or use code found online (including from AI tools).

Remember: Generative AI tools are prohibited for producing solutions. While you may use AI for understanding concepts, submitting AI-generated code as your own work is academic dishonesty.

SEE THE NEXT PAGE FOR RULES AND SUBMISSION FORMAT REQUIREMENTS

- Your code must run on **Python 3.10 or higher**. Code that requires a different/older Python version will not be graded.
- You may have additional helper methods/functions in your classes beyond what is specified in the assignment. However, maintain good code organization and use appropriate access control (e.g., prefix private helper methods with underscore: `_helper_method`).
- Your lexer must track line and column numbers accurately. These will be tested with various input files.
- Error messages must include:
  - Error type ("Lexical Error" or "Syntax Error")
  - Line number
  - Column number
  - A descriptive message
- Your `__repr__` methods for AST nodes should produce human-readable output that clearly shows the tree structure. We will use these for grading, so make sure they are complete and informative.
- **Do not modify the provided grammar or token specifications.** Your implementation must follow the grammar exactly as specified in this document.
- Your `main.py` must accept a filename as a command-line argument and work as follows:
  - `python main.py test1.ml` should read from `test1.ml`
  - Print either the AST (on success) or error messages (on failure) to standard output
- **What to submit?** All five required Python files (`token.py`, `lexer.py`, `ast_nodes.py`, `parser.py`, `main.py`) plus a `README` file, all submitted as a single `.zip` file via Brightspace.
  Deviations from the expected submission format carry varying amounts of score penalty:
  - Code requires Python version other than 3.10+. *Penalty: not graded.*
  - Submission only consists of compiled files, but no .py program files. *Penalty: not graded.*
  - Missing one or more of the five required Python files. *Penalty: no points for any portion of the rubric affected by this absence.*
  - Code uses external libraries beyond Python standard library and its modules. *Penalty: 10 points.*
  - Submission includes compiled `.pyc` files or `__pycache__` directories. *Penalty: 3 points.*
  - Code does not run from command line as specified (wrong arguments, missing `main.py`, etc.). *Penalty: 10 points.*
  - Missing `README` file or `README` does not explain how to run the program. *Penalty: 5 points.*
  - Submission is not a `.zip` file (e.g., `.rar`, `.7z`, or individual files). *Penalty: 3 points.*
- **Code that does not compile/run will receive zero credit for all implementation components**, regardless of partial correctness. **Make absolutely sure your code runs before submitting!**

Submission Deadline: March 2 (Monday), 11:59 pm.