CMPSC461 Fall-2025
Programming Language Concepts
Instructor: Dr. Suman Saha
Project #1: Building a Parser
Due: October 31, 2025

Introduction:

Students will implement a parser for a custom programming language. The language will support basic arithmetic operations, boolean expressions, variable assignments, control flow structures like if-else and while, and function calls. The goal is to understand parsing techniques, abstract syntax trees (AST).

Project Outline:

1. **Lexer:**
    a. Write a lexer to tokenize input code into meaningful tokens (e.g., keywords, operators, identifiers, numbers).
    b. Handle basic tokens like if, else, while, +, -, *, /, =, !=, ==, <, >, (,), :, etc.
2. **Parser:**
    a. Build a parser that converts a token stream into an Abstract Syntax Tree (AST).
    b. Implement parsing logic for different constructs like assignment statements (x = expression), binary operations (expression1 + expression2), boolean expressions (x== y), if-else statements, while loops, and function calls.
    c. Use recursive descent parsing methods to generate AST nodes for these constructs.
3. **AST Representation:**
    a. AST node definitions are already defined in the project repo, please use them to create AST node instances from the parser. Please don't make any changes to the code that is related to AST node definitions.

Grammar of the language:

```
program            ::= statement*
statement          ::= assign_stmt | if_stmt | for_stmt | print_stmt
assign_stmt        ::= IDENTIFIER '=' expression
if_stmt            ::= 'if' boolean_expression ':' block ('else' ':'
block)?
for_stmt           ::= 'for' IDENTIFIER '=' expression 'to'
expression ':' block
print_stmt         ::= 'print' '(' arg_list? ')'
block              ::= statement*
arg_list           ::= expression (',' expression)*

boolean_expression ::= boolean_term ('or' boolean_term)*
boolean_term       ::= boolean_factor ('and' boolean_factor)*
boolean_factor     ::= 'not' boolean_factor | comparison
comparison         ::= expression (('==' | '!=' | '<' | '>')
expression)*

expression         ::= term (('+' | '-') term)*
term               ::= factor (('*' | '/' | '%') factor)*
factor             ::= ('+' | '-') factor | primary
primary            ::= NUMBER | IDENTIFIER | '(' expression ')'

IDENTIFIER         ::= [a-zA-Z_][a-zA-Z0-9_]*
NUMBER             ::= [0-9]+
```

Example:

Let's take the following program and understand the derivation based on the new grammar:

```
x = 1
for i = 1 to 5:
     if i % 2 == 0:
     print(i)
```

This program has two top-level statements:
1. x = 1
2. for i = 1 to 5: ...

--- Derivation of the first statement: x = 1 ---

This statement is an assignment statement.

Grammar rule: statement ::= assign_stmt

1. Derive statement (x = 1):
   statement ::= assign_stmt
   assign_stmt ::= IDENTIFIER '=' expression
2. Derive the expression part (the number 1):
   expression ::= term
   term ::= factor
   factor ::= primary
   primary ::= NUMBER
   NUMBER ::= 1
3. Full derivation of x = 1:
   statement ::= assign_stmt
   assign_stmt ::= IDENTIFIER '=' expression
   IDENTIFIER ::= x
   expression ::= term
   term ::= factor
   factor ::= primary
   primary ::= NUMBER
   NUMBER ::= 1

--- Derivation of the second statement: for i = 1 to 5: ... ---

This statement is a for-loop.
Grammar rule: statement ::= for_stmt

for_stmt ::= 'for' IDENTIFIER '=' expression 'to' expression ':' block
1. Derive the loop signature:
   IDENTIFIER ::= i
   The start expression (1) and end expression (5) are derived just like in the assignment statement above, down to NUMBER.
2. Derive the block part of the for-loop:
   The block contains a single if statement.
   block ::= statement*
   statement ::= if_stmt
   if_stmt ::= 'if' boolean_expression ':' block ('else' ':' block)?
3. Derive the boolean_expression (i % 2 == 0):
   boolean_expression ::= boolean_term
   boolean_term ::= boolean_factor
   boolean_factor ::= comparison
   comparison ::= expression '==' expression
   - Derive the left expression (i % 2):
     expression ::= term
     term ::= factor '%' factor

factor ::= primary
primary ::= IDENTIFIER
IDENTIFIER ::= i
factor ::= primary
primary ::= NUMBER
NUMBER ::= 2

- Derive the right expression (0):
  expression ::= term
  term ::= factor
  factor ::= primary
  primary ::= NUMBER
  NUMBER ::= 0

4. Derive the block part of the if-statement:
   The block contains a single print statement.
   block ::= statement*
   statement ::= print_stmt
   print_stmt ::= 'print' '(' arg_list? ')'
   arg_list ::= expression
   expression ::= term
   term ::= factor
   factor ::= primary
   primary ::= IDENTIFIER
   IDENTIFIER ::= i
5. Entire derivation of the second statement:
   statement ::= for_stmt
   for_stmt ::= 'for' IDENTIFIER '=' expression 'to' expression ':' block
   IDENTIFIER ::= i
   expression ::= ... (derives to 1)
   expression ::= ... (derives to 5)
   block ::= statement
   statement ::= if_stmt
   if_stmt ::= 'if' boolean_expression ':' block
   boolean_expression ::= comparison
   comparison ::= expression '==' expression
   expression ::= term (derives to i % 2)
   expression ::= term (derives to 0)
   block ::= statement
   statement ::= print_stmt
   print_stmt ::= 'print' '(' arg_list ')'
   arg_list ::= expression (derives to i)

Important Instructions:

The project consists of 4 Python files:

1. **ASTNodeDefs.py**: This file contains the definitions for the AST nodes. You will have to create the AST using the node definitions provided in this script. DO NOT make any changes to this file.
2. **Parser.py**: This file contains the incomplete implementations of the Lexer and Parser. You will have to complete all functions marked as "TODO". This is the only file that you have to upload to Gradescope. DO NOT change the file name or its library imports. DO NOT import any external Python packages as the autograder environment does not have any additional packages installed.
3. **checker.py**: This file contains 7 test cases with inputs and expected outputs. You should use this file as a reference for your implementation. Note that the test cases provided here are not comprehensive as they do not cover all possible cases.
4. **verify.py**: This script runs the parser on the test cases provided in checker.py. Carefully analyze this file to understand how the Lexer and Parser classes are used to run the test cases.
5. The Lexer and Parser classes contain two important high-level functions: `tokenize()` and `parse()`. tokenize() must return a list of tokens in the code (expected return type is `List[Tuple[str, Any]]`). `parse()` returns the AST after parsing the code (expected return type is `List[ASTNode]`). You may create other functions in the classes if you wish. Additionally, you do not have to use all the helper functions provided in the Parser class (i.e., `parse_if_stmt()`, `parse_for_stmt()`, etc.). These functions return individual AST nodes for different statements and are provided as they help you break down the parsing process into logical chunks. You could theoretically implement everything in the `parse()` function itself. However, creating modular functions will help with debugging. For example, if a test case with a for loop fails, you know that the error is almost certainly in the `parse_for_stmt()` function.

Test cases and their AST representations:

We have provided a few representations of AST for some test cases in the project.

Test Case 1:

```
x = 10
if x > 5:
```

```
 y = 1
else:
 y = 0
```

AST Representation:

Assignment(('IDENTIFIER', 'x'), ('NUMBER', 10))
IfStatement(BinaryOperation(('IDENTIFIER', 'x'), ('GREATER', '>'), ('NUMBER', 5)),
Block([Assignment(('IDENTIFIER', 'y'), ('NUMBER', 1))]),
Block([Assignment(('IDENTIFIER', 'y'), ('NUMBER', 0))]))

Test Case 2:

```
for i = 1 to 10:
 if i % 2 == 0:
   print(i)
```

AST Representation:

ForStatement(('IDENTIFIER', 'i'), ('NUMBER', 1), ('NUMBER', 10),
Block([IfStatement(BinaryOperation(BinaryOperation(('IDENTIFIER', 'i'), ('MODULO', '%'), ('NUMBER', 2)), ('EQ', '=='), ('NUMBER', 0)), Block([PrintStatement([('IDENTIFIER', 'i')])]), None)]))

Grading Criteria:

In addition to the 7 test cases provided in verify.py, which are worth 8 points each, there are 3 additional hidden test cases in the autograder. The first 7 test cases are the same as the test cases in checker.py. If your code passes all these 7 test cases locally, you are guaranteed to get 56/100 points. The hidden test cases may not have equal weighting. The maximum total points for this project is 100, assuming all 10 test cases are passed successfully.

Academic Integrity Course Policy:

You are prohibited from copying any content from the Internet including (discord or other group messaging apps) or discussing, sharing ideas, code, configuration, text, or anything else or getting help from anyone in or outside of the class. Consulting online sources is acceptable, but under no circumstances should anything be copied. Failure to abide by this requirement will result in penalty as described in our course syllabus.

Dishonesty includes but is not limited to cheating, plagiarizing, facilitating acts of academic dishonesty by others, having unauthorized possession of examinations, submitting work of another person, or work previously used without informing the instructor. Students who are found to be dishonest will receive academic sanctions and will be reported to the University's Office of Student Conduct for possible further disciplinary sanctions; refer to Procedure G-9 (http://undergrad.psu.edu/aappm/G-9-academic-integrity.html). Furthermore, this course will follow the academic sanctions guidelines of the Department of Computer Science and Engineering, available at http://www.eecs.psu.edu/students/resources/EECS-CSE-Academic-Integrity.aspx
For violation of AI, we will follow

- 0 for the submission that violates AI, **AND**
- a reduction of one letter grade for the final course grade

(Students with prior AI violations will receive an F as the final course grade)