CMPSC461 Fall-2024

Programming Language Concepts

Instructor: Dr. Suman Saha

Project #1: Building a Parser

Due: November 1, 2024

Intoduction:

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 | while_stmt | expr_stmt | function_call
assign_stmt ::= IDENTIFIER '=' expression
if_stmt ::= 'if' boolean_expression ':' block ('else' ':' block)?
while_stmt ::= 'while' boolean_expression ':' block
block ::= statement*
expr_stmt ::= expression
function_call ::= IDENTIFIER '(' arg_list? ')'
arg_list ::= expression (',' expression)*
```

```
boolean_expression ::= term (( '==' | '!=' | '>' | '<' ) term)*
expression ::= term (( '+' | '-' ) term)*
term ::= factor (( '*' | '/' ) factor)*
factor ::= 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:

```
x = 10
if x > 10:
    foo(x + 20)
```

This program has two statements:

1. `x = 10`
2. `if x > 10: foo(x + 20)`

The first statement is an assignment statement : x = 10

```
statement ::= assign_stmt | if_stmt | while_stmt | expr_stmt | function_call

Derive statement (x = 10):
statement ::= assign_stmt
assign_stmt ::= IDENTIFIER '=' expression

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

Derive number:
factor ::= NUMBER
NUMBER ::= 10
```

Full derivation of the statement : x = 10

```
assign_stmt ::= IDENTIFIER '=' expression
IDENTIFIER ::= x
```

```
expression ::= term
term ::= factor
factor ::= NUMBER
NUMBER ::= 10

The second statement is ( if x > 10: foo(x + 20))

statement ::= if_stmt
if_stmt ::= 'if' boolean_expression ':' block ('else' ':' block)?

First derive boolean expression:
boolean_expression ::= term (( '==' | '!=' | '>' | '<' ) term)*

Derivation for boolean expression:
boolean_expression ::= term '>' term
term ::= factor
factor ::= IDENTIFIER
IDENTIFIER ::= x
term ::= factor
factor ::= NUMBER
NUMBER ::= 10

Next we derive block part of the if statement:
block ::= statement*

statement ::= function_call
function_call ::= IDENTIFIER '(' arg_list? ')'

arg_list ::= expression (',' expression)*
expression ::= term (( '+' | '-' ) term)*

term ::= factor
factor ::= IDENTIFIER
IDENTIFIER ::= x
term ::= factor
factor ::= NUMBER
NUMBER ::= 20

Entire derivation of the second statement is:

statement ::= if_stmt
if_stmt ::= 'if' boolean_expression ':' block
boolean_expression ::= term '>' term
term ::= factor
```

```
factor ::= IDENTIFIER
IDENTIFIER ::= x
term ::= factor
factor ::= NUMBER
NUMBER ::= 10
block ::= statement
statement ::= function_call
function_call ::= IDENTIFIER '(' arg_list ')'
IDENTIFIER ::= foo
arg_list ::= expression
expression ::= term '+' term
term ::= factor
factor ::= IDENTIFIER
IDENTIFIER ::= x
term ::= factor
factor ::= NUMBER
NUMBER ::= 20
```

Test case and their AST representations:

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

Test case 1:

```
x = 5

y = y + x
```

```
AST representation:
[Assignment(
  ('IDENTIFIER', 'x'),
  ('NUMBER', 5)
), Assignment(
  ('IDENTIFIER', 'y'),
  BinaryOperation(
    ('IDENTIFIER', 'y'),
    ('PLUS', '+'),
    ('IDENTIFIER', 'x')
  )
)]
```

Test case 2:

```
x = 1
y = 2
z = 3
x = x * y + z
y = y + x / z
if x != y:
z = 100
```

AST representation:

```
[Assignment(
  ('IDENTIFIER', 'x'),
  ('NUMBER', 1)
), Assignment(
  ('IDENTIFIER', 'y'),
  ('NUMBER', 2)
), Assignment(
  ('IDENTIFIER', 'z'),
  ('NUMBER', 3)
), Assignment(
  ('IDENTIFIER', 'x'),
  BinaryOperation(
    BinaryOperation(
      ('IDENTIFIER', 'x'),
      ('MULTIPLY', '*'),
      ('IDENTIFIER', 'y')
    ),
    ('PLUS', '+'),
    ('IDENTIFIER', 'z')
  )
), Assignment(
  ('IDENTIFIER', 'y'),
  BinaryOperation(
    ('IDENTIFIER', 'y'),
    ('PLUS', '+'),
    BinaryOperation(
      ('IDENTIFIER', 'x'),
      ('DIVIDE', '/'),
      ('IDENTIFIER', 'z')
    )
  )
), IfStatement(
  BooleanExpression(
    ('IDENTIFIER', 'x'),
    ('NEQ', '!='),
    ('IDENTIFIER', 'y')
  ),
  Block(
        Assignment(
        ('IDENTIFIER', 'z'),
        ('NUMBER', 100)
      )
  ),
  None
)]
```

You must understand that you don't need to create these ASTs, you just need to complete the lexer and parser implementation, the AST creation is already taken care by the implementation we provided.

Instructions:

1. Don't modify ASTNodeDefs.py and verify.py files.
2. Please refer to checker.py to see the expected output format.
3. Please don't change the name any files in the directory.

Steps to complete the project:

1. Understand the grammar thoroughly.
2. Go through checker.py file to see the expected output requirement, please always make a copy of that file before if you are going to modify.
3. Try implementing lexer and parser for first two test-cases.
4. Verify the correctness of your implementation by running verify.py, it will output how many of your testcases got passed.
5. Iterate on step 2, to handle more test-cases.

Grading Criteria:

In addition to the 7 test cases provided in `verify.py`, which are worth 10 points each, there are 3 additional hidden test cases in the autograder. These hidden test cases do 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

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)