Project 2: Scoping Rules and Type Checking in a Custom Language Parser

1. Project Description

In this project, you will implement a parser for a custom programming language. The language has a set of rules governing variable declarations, type compatibility, and scoping, similar to many modern programming languages. The goal is to enhance your understanding of fundamental programming language concepts such as scoping rules, type checking, and parser design. You will implement various functionalities to check for errors like type mismatches, scope violations, and variable redeclarations. Your implementation will emit error messages to provide feedback on the parsing process, ensuring adherence to the language's rules. This project is an extension of Project 1, where you initially built a basic parser for a custom language. Now, you will extend that parser to include additional features related to scoping and type checking. You can use your Project 1 code to extend the functionality. If you did not complete Project 1, the instructor's solution will be provided as a starting point.

2. Learning Objectives

1. Understand the concept of lexical analysis and syntactic analysis;
2. Learn how to implement a symbol table for managing variable declarations and scoping;
3. Gain a solid understanding of type checking in programming languages;
4. Develop an understanding of error handling and diagnostic messages for syntax analysis.

3. New Language Grammar

The grammar is mostly the same with Project 1, with the following changes:

We add declaration statement in *statement*, and the new grammar for *statement* is:

```
statement ::= decl_stmt | assign_stmt | if_stmt | for_stmt | print_stmt
```

In declaration statements, types of variables are specified by 'int' (decimal number), 'hex' (hexdecimal number), 'bool' (Boolean). The following rules are new for Project 2 extended from the grammar in Project 1:

```
decl_stmt ::= type IDENTIFIER
type ::= 'int' | 'hex' | 'bool'
primary ::= NUMBER | HEXNUMBER | BOOL | IDENTIFIER | '(' boolean_expression ')'
HEXNUMBER ::= 0x[0-9a-f]+
BOOL ::= 'True' | 'False'
```

To accommodate scoping, we change if_stmt and for_stmt as the following:

```
if_stmt ::= 'if' boolean_expression ':' block ('else' ':' block)? 'endif'
for_stmt ::= 'for' IDENTIFIER '=' expression 'to' expression ':' block 'endfor'
```

All other rules are the same with Project 1. For your convenience, the complete grammar for Project 2 is shown below:

```
program        ::= statement*
statement      ::= decl_stmt | assign_stmt | if_stmt | for_stmt | print_stmt
decl_stmt      ::= type IDENTIFIER
type           ::= 'int' | 'hex' | 'bool'
assign_stmt    ::= IDENTIFIER '=' expression
if_stmt        ::= 'if' boolean_expression ':' block ('else' ':' block)? 'endif'
for_stmt       ::= 'for' IDENTIFIER '=' expression 'to' expression ':' block '
    endfor'
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 | HEXNUMBER | BOOL | IDENTIFIER | '('
    boolean_expression ')'

IDENTIFIER     ::= [a-zA-Z_][a-zA-Z0-9_]*
NUMBER         ::= [0-9]+
HEXNUMEBR      ::= 0x[0-9a-f]+
BOOL           ::= 'True' | 'False'
```

4. Rules to Implement

4.1. Type Compatibility

In this project, we will implement a small set of type compatibility checks.

1. For boolean operations (or, and, not), all operands should have type 'bool', and the output also has type 'bool';
2. For arithemtic operations, all operands should have type 'int' or 'hex', and their types should match. All arithmetic operators do not change the type of output (e.g. '+' of a hex and another hex is still a hex);
3. For comparison operations ($==, != <, >$), all operands should have the same type, and the output has type 'bool';
4. For assignment statement, both sides should have the same type;
5. For 'if_stmt', the condition should have type 'bool'.

When a type mismatch occurs, a type mismatch error should be raised by using the provided function in the starter code.

For example,

```
hex a
int b
if a > b: // Invalid: Rule 3 requires a and b have the same type
    int c
    c = a + b // Invalid: Rule 2 requires a and b have the same type
    int d
    d = a // Invalid: Rule 4 requires d and a have the same type
    print(a,b) // Allowed for different types in print_stmt
    for d = 1 to 0x10: // Allowed for different types in for conditions
        print(d)
    endfor
endif
bool e
if a and e: // Invalid: Rule 1 requires a and e both have type 'bool'
    if b: // Invalid: Rule 5 requires type 'bool' in condition
        print(a)
    endif
endif
```

4.2. Variable Declaration Before Use

A variable must be declared before it is used in an expression or statement. You should track variable declarations using a symbol table and check each variable usage to ensure it has already been declared.

For example,

```
int a
a = 5 // Valid: a is declared before use
b = 10 // Invalid: b is not declared in the current or any enclosing scopes
```

4.3. No Redeclaration in the Same Scope

Variables cannot be declared multiple times in the same scope. Attempting to declare a variable with the same name in the same scope should raise a redeclaration error. However, redeclaration is allowed in different (nested) scopes.

For example,

```
int a
int a // Invalid: a has already been declared in the current scope
if a > 5:
    int a // Valid: this is allowed as it is in a new (nested) scope
endif
```

4.4. Scoping Rules

Implement block scoping for variables. A variable declared inside a block should not be accessible outside of that block, but it should be accessible within the block and its nested blocks.

For example,

```
if 1 > 2:
    int a
    a = 10
    if 3 > 4:
        int b
        b = 20
        a = a + b // Valid: b is accessible within the nested block
    endif
    b = 5 // Invalid: b is not declared in the current or any enclosing scopes
endif
a = 15 // Invalid: a is not declared in the current or any enclosing scopes
```

5. Project Workflow

5.1. Step 1: Lexer Implementation

Implement the lexer to tokenize the input string. Keep in mind that we added new keywords ('int', 'hex', 'endif', 'endfor') compared to Project 1. The lexer should also distinguish literals in int type and hex type.

5.2. Step 2: Parser Implementation

Write functions to parse each type of statement according to the grammar. Implement scope management using helper functions **enter_scope()** and **exit_scope()**

5.3. Step 3: Error Handling and Type Checking

Add type checking logic to ensure type compatibility in expressions and assignments. Use the symbol table to track variable declarations and detect errors related to scoping.

5.4. Step 4: Testing

Implement a set of test cases to validate your parser. Test cases should cover the pass and fail cases for all rules which are required to implement. We will provide 10 test cases for you, and you are encouraged to write more test cases. In addition, there will be 5 hidden test cases when grading.

6. Expected Output

Your parser should generate an AST for correct input and emit descriptive error messages for invalid code. The format of the error message is provided in the released code. The error messages should be emitted in the order of occurrence in the input program. Ensure that you test your implementation thoroughly for all rules. For valid input programs, we test on the generated AST.

For invalid input programs, we test on the error message (AST does not matter when an error happens). For simplicity, when there are multiple type checking errors in a single statement (except 'if_stmt' and 'for_stmt'), we only expect the error message of the first type error found during parsing. You can stop type inference within a statement when a type error occurs.

7. Submission

You are expected to modify **Parser.py** file only, so please submit your **Parser.py** on Gradescope. All others files are not accepted.

8. Additional Notes

1. This project is meant to challenge your understanding of compiler design fundamentals. Don't hesitate to reach out if you have questions.
2. You are encouraged to write clean and modular code. Break down the parsing logic into smaller and reusable functions where possible.

9. 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

1. 0 for the submission that violates AI, AND
2. 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)