CS 343S: Domain Specific Language Design Studio **Spring 2025**

External Lab: MatLang

Due: Thursday, April 17rd, 11:59pm (`make submission.zip` and submit it via Gradescope)

Bugs: We make mistakes! If it looks like there might be a mistake in the statement of a problem, please ask a clarifying question on Ed.

In this assignment, you'll implement a small language for matrix arithmetic, which we will call MatLang. First, you will write a PEG grammar for MatLang, and translate the parse tree into an AST. Then, you will implement an tree-walking interpreter for the AST. Lastly, you will implement a simple type checker. As a bonus problem, you can add polymorphic functions and type inferrence to MatLang.

Setup

1. Install **parsimonious**. It is a python-based PEG parser.
2. Optional but recommended: Install **numpy**. It is a library which supports matrix operations in Python, which you can use to implement the interpreter.

1 MatLang Specification

A MatLang program consists of statements and function definitions. Each statement is ended with a semi-colon. Here is a small snippit of MatLang code:

```
def foo(x: Mat(2, 2), y: Mat(2, 2)) -> Mat(2, 2) {
    let z = x + y;
    return x * z;
}

# this is a comment
let a = [[1, 2], [3, 4]];
let b = [[1, 2], [3, 4]];
let c = foo(a, b);
print(c);
```

As you can see, MatLang should look familiar to other programming languages you have used. The following sub-sections describe each construct MatLang supports.

1.1 Comments

Text following a `#` on the same line is a comment, and should be ignored.

1.2 Expressions

1.2.1 Literals

MatLang supports integer, vector, and matrix literals. For example:

```
5
[1, 2, 3]
[[1, 2], [3, 4]]
```

Note: Expressions inside matrix literals are not supported. For example:

```
[1 + 2, 3] # Not supported
```

1.2.2 Arithmetic

MatLang supports the operators $+$, $-$, and $*$, where $*$ denotes matrix multiplication. For example:

```
[1, 2] + [3, 4] # result: [4, 6]
[1, 2] - [3, 4] # result: [-2, -2]
[[1, 2], [3, 4]] * [[1, 2], [3, 4]] # result: [[7, 10], [15, 20]]
```

Arithmetic should be *left-associative*, meaning $5-4-3$ is evaluated as $((5-4)-3)$. Multiplication should have higher *precedence* then addition or subtraction, meaning $5 + 4 * 3$ is evaluated as $5 + (4 * 3)$. Note: MatLang does not support negation (e.g., unary $-$) or parentheses.

1.2.3 Function Calls

MatLang supports function definitions (see 1.4) and function calls, which resemble those in other programming languages. For example:

```
foo(1, 2, [1, 2])
```

Here, `foo` is a function which is being called with three arguments: 1, 2, and $[1, 2]$.

1.3 Statements

Statements in MatLang end with a semicolon. Whitespace (e.g., spaces, indentation) is ignored.

1.3.1 Expression Statements

An expression followed by a semicolon becomes a statement. For example:

```
[1, 2];
foo(1, 2);
print(x, y);
```

1.3.2 Variable Bindings

MatLang supports storing the result of expressions as variables. For example:

```
let x = 5;
let y = x + 2;
let z = foo(x, y);
```

If a program references a variable that is undefined, MatLang should raise an error.

1.4 Function Definitions

Functions in MatLang must include type annotations for both arguments and return values. For example: For example:

```
def foo(x: Mat(1, 2), y: Mat(2, 3)) -> Mat(1, 1) {
    print(x * y);
    return 5;
}
```

See Section 1.6 for more details about function types. Function arguments are only in scope during the execution of the function. For example, the following program should raise an error: For example, the following program should raise an error:

```
def foo(x: Mat(1, 1)) {
    print(x * x);
}
foo(5);
print(x); # Error! Undefined variable x
```

1.5 Built-in Functions

MatLang has a single built-in function, `print`, which prints its arguments to `stdout`, and returns the empty matrix `Mat(0, 0)`. To implement this, you can just pass the values of its arguments to Python's print function. For example:

```
print([1, 2]) # output: [1, 2]
print(5, 6) # output: 5 6
print(print(5)) # output: 5
                #         []
```

We won't be picky about output formatting as long as it is clear what shape the values are. For example, `print(5, 6)` could output `5 6`, `[5] [6]`, or `[[5]] [[6]]`.

1.6 Types

All values in MatLang are typed as integer matricies (i.e. the language does not support floats, strings, etc.). We use the notation `Mat(r, c)` to denote a matrix with r rows and c columns. For example, `[[1, 2], [3, 4], [5, 6]]` has type `Mat(3, 2)`.

Integers are typed as 1 by 1 matricies (`Mat(1, 1)`), and vectors are typed as 1 by n matrices (`Mat(1, n)`). MatLang also allows empty matricies with type `Mat(0, 0)`.

Functions in MatLang must be annotated with types for the arguments and return values. For example:

```
def foo(x: Mat(1, 2), y: Mat(2, 3)) -> Mat(1, 1) {
    print(x * y);
    return 5;
}
```

Here, `foo` is a function which takes two arguments: `x` of type `Mat(1, 2)`, and `y` of type `Mat(2, 3)`. The return type of the function is indicated after the arrow. In the example, `foo` returns a value of type `Mat(1, 1)`. Functions without an explicit return statement return `Mat(0, 0)`. For example, y has the type `Mat(0, 0)` in the following code:

```
def foo(x: Mat(1, 1)) {
    print(x);
}
let y = foo(5);
```

Calling functions with arguments whose types do not match those given in the definition should raise a TypeError. For example, the following code should raise a TypeError, because `y` is of type `Mat(1, 1)`, but `print_2` requires its argument has type `Mat(1, 2)`.

```
def print_2(x: Mat(1, 2)) {
    print(x);
}
let y = 5;
print_2(y);
```

Although you will not need to implement type checking until Part 3 of the assignment, we recommend you implement the parsing of the function type annotations in Part 1.

Functions definitions whose types are malformed should raise some sort of error. For example, the following function defitions should all cause errors:

```
def bad1(x: Matt(1, 1)) {
    return x;
}

def bad2(x: Mat(print(1), 1)) {
    return x;
}

def bad3(x: 1) {
    return x;
}
```

We won't be picky about what exact errors are thrown (e.g. parsing error vs TypeError), as long as there is one.

2 Parsing

First, implement the PEG grammar for MatLang in the `grammar.peg` file. Then, implement the `MatrixVisitor` class methods in the `parse.py` file to convert the parse tree into an Abstract Syntax Tree. Remember that each method should be called `visit_<RULE_NAME>(self, node, visited_children)`, where `<RULE_NAME>` is the exact name of the rule in `grammar.peg`. Also, note that by default, parsimonious will throw an error if there are trailing characters in the text which do not parse. You can run `python3 parse.py [input_file]` to see what parsimonious outputs for you grammer on an input file.

We provide an AST you can use in `AST.py`, though feel free to modify it as you see fit. The root node of the AST should be a `Block` node, which contains all statements in the program. If you use our AST definitions, we also provide a few test cases which you can run using `python3 test_ast.py`.

Written After implementing the parser, describe your design process in `written.md`.

3 Interpretation

Next, implement the `interpret_expr`, `interpret_stmt`, and `interpret_block` functions in `AST.py`. These methods should evaluate the expressions and statements in the AST.

We provide a utility class `ScopedDict` in `utils.py` that you can use to help scope variable bindings. This can be useful to remove the bindings for function arguments after returning.

Written After implementing the interpreter, mention if you noticed any ambiguities in the language specification in `written.md`. If so, explain how your interpreter handled those ambiguities in `written.md`.

4 Typing

Lastly, implement the `type_expr` and `type_stmt` functions in `typ.py` (we have implemented `type_block` for you). These functions should check that a given MatLang AST is well typed. Make sure you check both that the arithmetic operators and function calls are well typed. If it encounters a node which has a type error, then it should raise the `TypeError` exception provided in `typ.py`. The `msg` field of the `TypeError` can be whatever message you want.

5 Bonus: Polymorphic Functions (Not Required)

Allow function parameters to be *polymorphic* in their shape. In particular, their types may contain variables for their dimension. For example:

```
def bar(x: Mat(a, b), y: Mat(a, b)) {
    print(x + y);
}
```

Here, `a` is a variable representing a dimension of the Matrix. When calling a function whose parameters are polymorphic, the variables with the same name must be equal. For example:

```
bar([1, 2], [3, 4]) # correct
bar(1, [1, 2]) # incorrect: 1 has type Mat(1, 1), and [1, 2] has type Mat(1, 2)
```

Function signatures may also mix concrete and polymorphic dimensions:

```
def mixed(x: Mat(a, 1)) {
    ...
}
```

Here is a more complex example:

```
def baz(x: Mat(a, b), y: Mat(b, c)) -> Mat(a, c) {
    return x * y;
}

baz(1, 2) # correct
baz([1, 2], [[3], [4]]) # correct
baz([1, 2], [3, 4]) # incorrect: [1, 2] has type Mat(1, 2) and
                                 [3, 4] has type Mat(1, 2): 1 != 2
```

6 Bonus: Type Inferrence (Not Required)

For a bonus problem, add type inference to MatLang. Because the language is so simple, we can actually remove the type annotations from function defintions, and infer them from the body of the function.

First, allow the programmer to give `_` as a type annotation for function parameters. Any parameters with type `_` should have their type inferred. For example:

```
def foo(x: _, y: _) -> _ {
    return x + y;
}
```

The type inference should determine that `x`, `y`, and the return value have the type `Mat(a, b)`. Be sure not to break type checking implementing this!

7 Written

Answer the `Feedback` questions in `written.md`.

Submission

Build your submission zip with `make submission.zip` and submit it to Gradescope. If you added any additional files, be sure to add them to the Makefile target!