Lab 4
Operational Semantics for Mu language

Objective

- Write semantic and typing rules (for the Mu language).
- Implement them as visitors.

Student files are in the GIT repository. You may have to install the pytest module:

```
pip3 install pytest --user
```

You won't have to send your final work but only your test files. Type `make tar` to make an archive of all your test files. This archive is due on Wednesday, October 11th, 6pm to your teaching assistants. We will merge all your examples for the upcoming other labs.

Tools for the lab

Syntax for the language We give you the syntax of the Mu language, as a full grammar depicted in Figure 4.1.

Exceptions in PYTHON Recall that in PYTHON errors can be declared, thrown and caught as depicts Figure 4.2

4.1 Semantics for expressions

The (natural) semantics for expressions is given by the following rules seen in the course: (We do not recall all rules and notations)

$$\mathscr{A}[n]\sigma = \mathscr{N}(n)$$
$$\mathscr{A}[x]\sigma = \sigma(x)$$
$$\mathscr{A}[e_1 + e_2]\sigma = \mathscr{A}(e_1) +_I \mathscr{A}(e_2)$$

EXERCISE #1 ► **Expressions**
We give you a visitor that computes this semantics. Compare the implementation and the semantics.

4.2 Semantics for Mu-language

EXERCISE #2 ► **Mu semantics**
Write on paper the (big steps) operational semantics as already seen in the course. You can forget the `log` construction, that basically prints the expression given in argument.

EXERCISE #3 ► **Interpret Mu!**
Write the evaluator/interpretor for this mini-language. We give you the structure of the code and the evaluator for numerical expressions and boolean expressions. Type:

```
make run FOO='ex/testxx.mu'
```

and the evaluator will be run on `ex/testxx.mu` (or on `ex/test00.mu` if you do not precise variable `FOO`).

You still have to implement (in `MyMuVisitor.py`):

1. Variable declarations (`varDecl`) and variable use (`idAtom`): your evaluator should use a table (*dict* in PYTHON) to store variable definitions and check if variables are correctly defined and initialized. Refer to the three test files `ex/bad_defxx.mu` for the expected error messages. **For the moment, only consider well-typed expressions.**

```
grammar Mu;

prog: vardecl_l block EOF #progRule;

vardecl_l: vardecl* #varDeclList;

vardecl: VAR id_l COL typee SCOL #varDecl;

id_l
    : ID #idListBase
    | ID COM id_l #idList
    ;

block: stat* #statList;

stat
    : assignment
    | if_stat
    | while_stat
    | log
    | OTHER {print("unknown␣char:␣{}".format($OTHER.text))}
    ;

assignment: ID ASSIGN expr SCOL #assignStat;

if_stat: IF condition_block (ELSE IF condition_block)* (ELSE stat_block)? #ifStat;

condition_block: expr stat_block #condBlock;

stat_block
    : OBRACE block CBRACE
    | stat
    ;

while_stat: WHILE expr stat_block #whileStat;

log: LOG expr SCOL #logStat;
```

Figure 4.1: MU syntax (excerpt). We omitted here the subgrammar for expressions

```
# declare !
class MyError(Exception):
    pass

# catch!
        try:
            ...
        except MyError:
            ...

# launch !
        raise MyError("Error␣Message")
```

Figure 4.2: Exceptions in PYTHON

2. Statements: assignments, conditional blocks, tests, loops.

3. Test with `make tests` **and appropriate benchmarks**. You must provide your own tests. The only outputs are the one from the `log` function or the following error messages: "`Undefined variable m`", "`m has no value yet!`", "`Warning, variable n has already been declared`". You can choose the tested files by modifying the `ALL_FILES` variable in `test_evaluator.py`).

Tests work mostly as in the previous lab. For instance, if you fail `test00.mu` because you printed `42` instead of `99.0`, you will get this error:

```
_____________ TestCodeGen.test_expect[../ex/test00.mu] _____________

self = <test_evaluator.TestCodeGen object at 0x7f0e0aa369b0>
filename = '../ex/test00.mu'

    @pytest.mark.parametrize('filename', ALL_FILES)
    def test_expect(self, filename):
        expect = self.extract_expect(filename)
        eval = self.evaluate(filename)
        if expect:
>           assert(expect == eval)
E           assert '99.0\n1\n' == '42\n1\n'
E             - 99.0
E             + 42
E               1

test_evaluator.py:59: AssertionError
```

And if you did not print anything at all when `99.0` was expected, the last lines would be this instead:

```
        if expect:
>           assert(expect == eval)
E           assert '99.0\n1\n' == '1\n'
E             - 99.0
E               1

test_evaluator.py:59: AssertionError
```

Explain the test failure for `bad_type00.mu`. We will work on fixing it in the next exercise.

EXERCISE #4 ► **Typing**

Write typing rules for expressions (on paper). Then, implement a type checker for the Mu language[1] (as a standalone visitor `MyMuTypingVisitor`) [2]. We provide you with a (basic) class for basic types and the environment initialization with the declared types. The method `_raise` allows you to add informative exception handlers. The provided test files must guide you when the implementation cannot be directly derived from the typing rules. Do not forget to intensively use new test files.

EXERCISE #5 ► **Bonus (on paper)**

We want to extend our mini language with imperative arrays. The syntax is augmented with the three following constructions:

- `Alloc(e)` allocates a new array of size equal to the value of e, with undefined values
- `Read(e1,e2)` reads the e_2^{th} value of array e_1.
- `Write(e1,e2,e3)` modifies the e_2^{th} value of array e_1 with the value of expression e_3.

Complete the typing rules and semantics of expressions, then give new rules for array modification.

[1] We do not ask for a decorated AST, only type checking.
[2] Do not forget to enable the call to this visitor in the main file.