

Student Name: Your name, your.email@ntut.edu.tw

Student ID: Your student ID number,

Instructor: Jyun-Ao Lin, jalin@ntut.edu.tw

The goal of this lab is to build an interpreter for a simple fragment of Python, called **mini-Python**. You don't have to know Python. A **mini-Python** file has the following structure:

```
# zero, one or several function definitions at the beginning of the file
def fibaux(a, b, k):
    if k == 0:
        return a
    else:
        return fibaux(b, a+b, k-1)

def fib(n):
    return fibaux(0, 1, n)

# then one or several statements at the end of the file
print("a few values of the Fibonacci sequence:")
for n in [0, 1, 11, 42]:
    print(fib(n))
```

More generally, a **mini-Python** file is composed of an optional list of function definitions, followed by a list of statements.

Caveat: the last statement must be followed by a newline.

Statements are: assignment, conditional, loop (**for**), output with **print**, returning a value with **return**, and evaluation an expression.

Expressions are: a constant (Boolean, integer, or string), access to a variable, building a list (with syntax `[e1, e2, ..., en]`), access to a list element (with syntax notation `e[i]`), function call, or one of the operations `+`, `-`, `*`, `//`, `==`, `<>`, `<`, `<=`, `>`, `>=`, **and**, **or** and **not**.

We also consider three built-in functions: **list(range(n))** builds the list `[0, 1, 2, ..., n-1]` and **len(l)** returns the length of list `l`. (We only use **list** and **range** jointly in this way.)

Code supplied . To help you building this interpreter, we provide the basic structure (as a set of Java files together with a Makefile).

Download this archive: [mini-python-java.tar.gz](#) on Teams.

Once uncompressed (with `tar zxvf mini-python.tar.gz`), you get a directory `mini-python-java` /. You can run Eclipse in this directory with the command `eclipse -data .`, then create a project `mini-python-java` (disabling option Use default location). This project contains a package `mini_python` with the following files:

Syntax.java	abstract syntax of mini-Python (complete)
Lexer.java, MyLexer.java	lexical analysis (complete)
parser.java, sym.java	parsing (complete)
Interp.java	Interpreter (to be completed)
Main.java	main file (complete)
Makefile	to automate the build, if you don't use Eclipse (complete)

The library `java-cup-11a-runtime.jar`, contained in sub-directory `lib/`, must be added to the project (with Java Build Path -> Libraries -> Add JARs). The code compiles but is incomplete. Places to be completed are indicated with `throw new Todo()`.

The executable takes a mini-Python file on the command line, with suffix `.py`. When it is absent, file `test.py` is used.

Exercise 1: Arithmetic Expressions

For the moment, we only consider arithmetic expressions without variables. Complete the interpreter to evaluate such expressions. Test on the following file

```
print(1 + 2*3)
print((3*3 + 4*4) // 5)
print(10-3-4)
```

whose output must be

```
7
5
3
```

Division and modulus operations must signal a division by zero, by raising exception `Error`. To test, simply edit file `test.py` and run `Main`.

Exercise 2: Boolean Expressions and Conditionals

Complete methods `isTrue` and `isFalse` in class `Value`, which respectively decide whether a value is true or false. In Python, the value `None`, the Boolean `False`, the integer `0`, the empty string `""` and the empty list `[]` are all considered false, and any other value is considered true. Then complete the interpreter with Boolean constants, arithmetic comparison and operations `and`, `or` and `not`. In Python, comparison is structural. You can use the method `compareTo` that is provided in class `Value`.

Finally, complete the interpreter to handle the conditional (construction `Sif`).

Test on the following file

```
print(not True and 1//0==0)
print(1<2)
if False or True:
    print("ok")
else:
    print("oups")
```

whose output must be

```
False
True
```

ok

Exercise 3: Variables

To handle variables (global variables but also local variables and parameters) we are going to use an environment, which maps each known variable to its value. This table is the attribute `vars` of the class `Interp`.

Complete the interpreter so that we can access variables (case `Eident`). Accessing a variable that is not in the map must raise an error.

Similarly, complete the interpreter so that we can assign a value to a variable (method `visit (Sassign s)`). This time, the variable may or may not be in the table. In the latter case, its value is modified.

Finally, complete the interpreter so that we can concatenate two strings with operation `+`. Test on the following file

```
x = 41
x = x+1
print(x)
b = True and False
print(b)
s = "hello" + " world!"
print(s)
```

whose output must be

```
42
False
hello world!
```

Exercise 4: Functions

We now consider function definitions and calls. Functions are stored in the global hash table `functions`. Each function name is mapped to a pair consisting of the list of its parameters and its body (a statement). The code supplied is already filling this table with function definitions.

Then complete the interpreter to handle function calls. For a call `f(e1, ..., en)` to a function `f` defined as `def f(x1, ..., xn): s`, we have to build a new environment that maps each formal parameter `xi` to the value of `ei`. This means building a new object of type `Interp`. Then we interpret the statement `s` (the body of the function) in this new environment. The statement `return` is interpreted using the Java exception `Return` (already defined). If the execution terminates without an explicit `return`, the value `None` must be returned.

Test on the following file

```
def fact(n):
    if n <= 1: return 1
    return n * fact(n-1)

print(fact(10))
```

whose output must be

3628800

Exercise 5: Lists

Add support for lists. To do so, complete function `expr` so that we can concatenate two lists with operation `+`, to interpret calls to `len` (length of a list) and `list(range(n))` (the list `[0, 1, 2, ..., n-1]`), and to interpret expressions `[e1, e2, ..., en]` and `e1[e2]`.

Complete function `stmt` to interpret the assignment of a list element (class `Sset`).

Finally, complete function `stmt` to interpret the `for` loop. The statement `for x in e: s` successively assigns to the variable `x` the values of the list `e`, and executes the statement `s` for each of them. The expression `e` must be evaluated only once.

Test using the program at the beginning of this assignment. The output must be:

```
0
1
89
267914296
```

Exercise 6: Other tests

Positive and negative tests are provided. To run your interpreter on these tests, launch `make tests`.