# Group 4: Sprint 1 Report (Parser)

## Lexer

In order to parse Python's grammar, we had to add two major changes to the original lexer flow:

- Comment states
- Post processing

Aside from these changes, the rest of the lexer consists of the TOKENS and regex rules defined in the grammar to parse it into a list of LexTokens.

### Comment States

In order to parse docstrings, we had to create exclusive states that would read through any string within two pairs of ‘”, and disregard it from creating a token. This was done because we didn't want to clutter the lex token stream with unnecessary tokens. Once the lexer reaches the closing triplet of ‘”, it exits out of the exclusive state and returns to its normal behavior.

### Post Processing

Python's grammar isn't context free, this means that it requires knowledge of previous lines to be able to successfully parse the correct tokens. For Python, this means keeping track of which lines need to be indented and which lines don't. We follow similarly to how Python's grammar works, in the sense that, we wrap any lines that need to be indented (if statements, loops, functions) with a pair of INDENT DEDENT tokens. This allows the parser to be able to interpret groups of tokens as segmented code (similar to how parentheses works for the calculator parser). However, this implementation posed several challenges, one being the need for a post processing step.

In our lexer, we first determine all the valid tokens. Afterwards, we loop through all these tokens to determine when to insert an indent/dedent. If the token is a COLON, or multiple WS (at the start of the line) then we can assume it requires an indent. If the lines no longer correspond to the right indentation, a dedent is inserted. Likewise, WS is filtered out of the tokens, as it's only needed to calculate indentation.

Having the lexer do more work allows the parser to focus on grouping and providing the correct semantics for the program.

---

## AST

### Description

Our input language is a subset of Python, so to make our parsed tree easily verifiable, we used Python's AST module for our nodes. Using these nodes made it easier to unparse our tree and verify that our tree was convertible to Python again. Having an easy way to verify like this will also help us in future sprints as we convert our input into different Python code (one line lambda function).

In our AST, some of the highlevel nodes we have include: a statement in Python (has to end with a newline in our case, some languages use semicolons), a list of such statements, function definitions (function name, arguments, body – which is a list of statements). The statements we support include for loops, while loops, if-statements, assignment statements, and general expressions in Python. The lower-level (base) nodes include different expressions such as numbers, strings, floats, function calls, ids, binary operations, and boolean operations. Expression nodes for base type constants such as numbers, strings and floats have a kind (type) attribute to make it easier for us to work with the parsed tree. Everything is wrapped in a Python module node (comparable to a main class node in languages like Java) at the end.

### Language structures not parsed

We did not manage to support lists and dictionaries. We will be adding lists as a part of sprint 3, but we aren't planning to proceed with dictionaries.

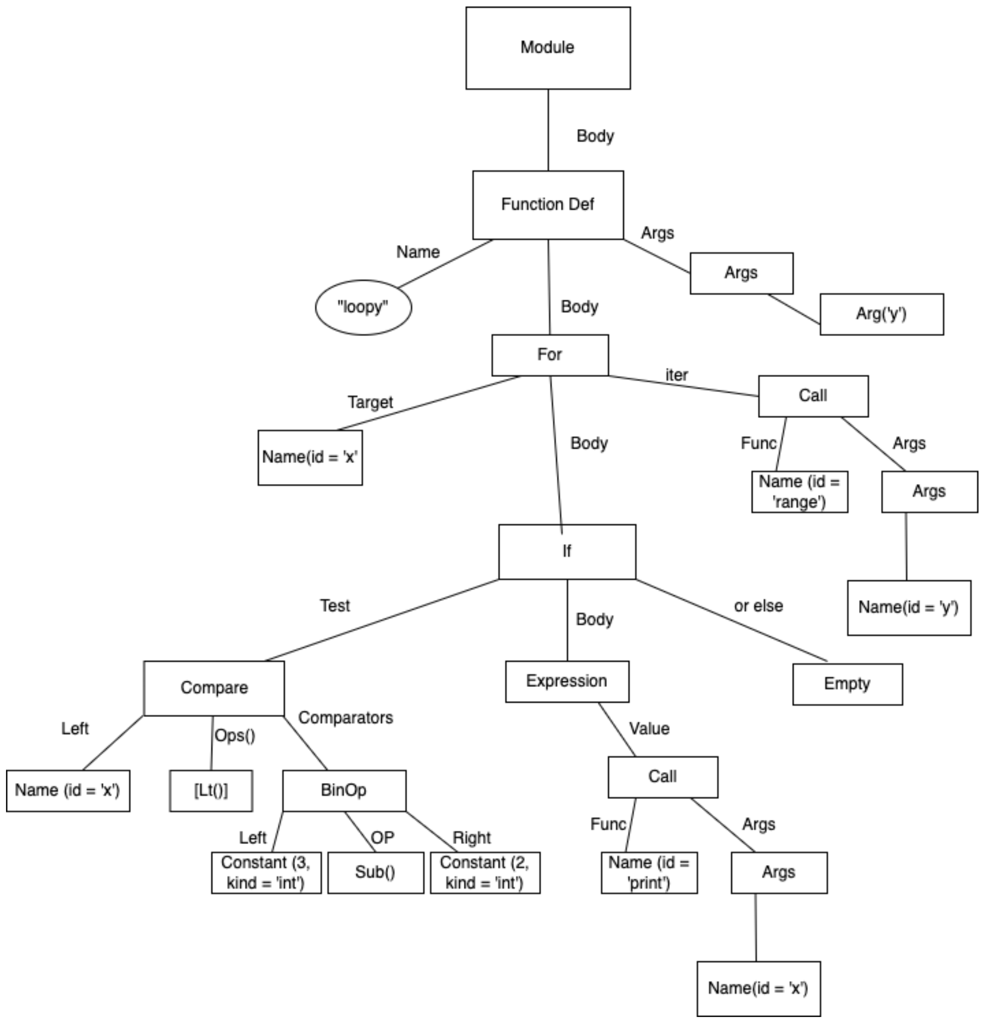
### Example inputs explanation

We have quite a few inputs that aim to test that our parser can parse the full subset of python that we have committed to supporting. These inputs are found in `/test_inputs` and `/code_examples`. Inside `/test_inputs`, we have

**syntactically** (not semantically) correct pieces of python code that aim to test our parser capabilities, these inputs test a good subset of our input language. Inside `/code_examples`, there are **syntactically and semantically** correct pieces of Python code to test our parser. These inputs are not as comprehensive as our test inputs, but they contain valid pieces of code that can be run. More information regarding our testing process can be found in the verification scripts and examples section below.

## Example Visual

Here is what a tree we generate would look like for example



The code this represents:

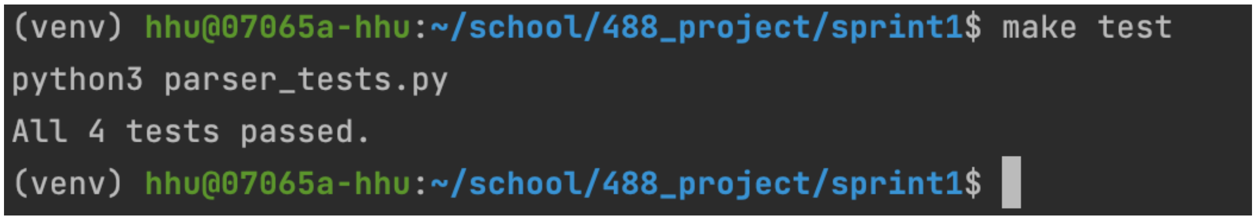
```
def loopy(y):
   for x in range(y):
       if x < 3 - 2:
           print(x)
```

## Verification Script and Examples

### **Requirements to run: Python 3.9.x and ply==3.11**

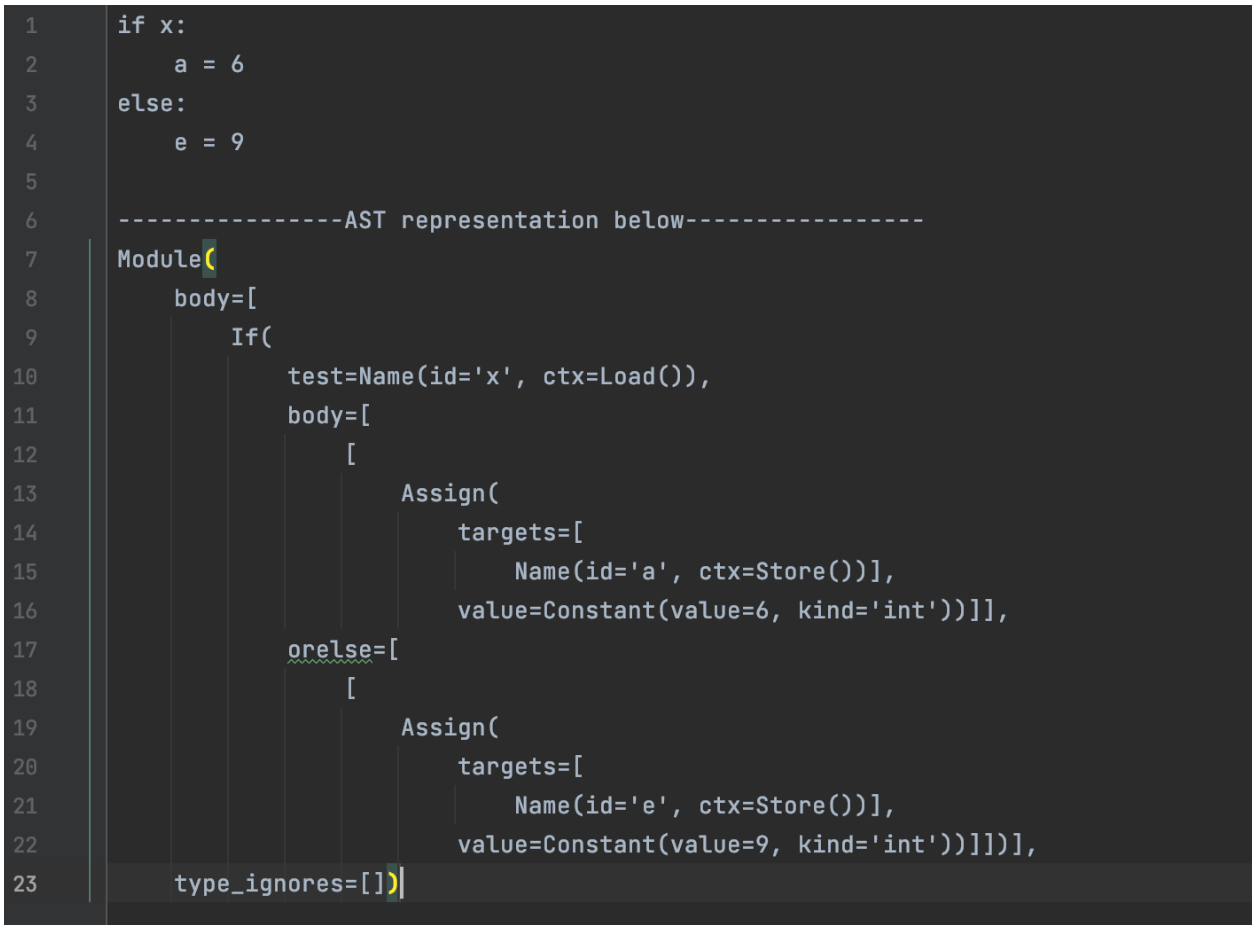
### How to run scripts and verify output files

When you cd into our sprint 1 folder, you can run `make test` to run the testing script like so:



```
(venv) hhu@07065a-hhu:~/school/488_project/sprint1$ make test
python3 parser_tests.py
All 4 tests passed.
(venv) hhu@07065a-hhu:~/school/488_project/sprint1$
```

This runs the tests and writes the text representation of the ASTs generated to output files in the **/out** directory.



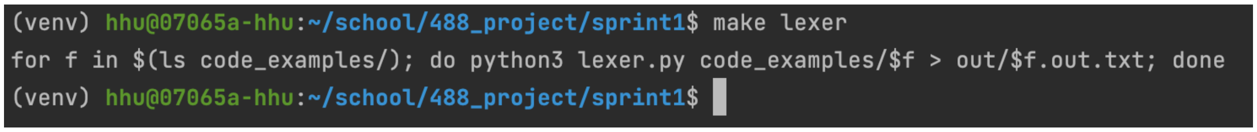
```
if x:
    a = 6
else:
    e = 9

----------------AST representation below----------------
Module(
    body=[
        If(
            test=Name(id='x', ctx=Load()),
            body=[
                [
                    Assign(
                        targets=[
                            Name(id='a', ctx=Store())],
                        value=Constant(value=6, kind='int'))]],
            orelse=[
                [
                    Assign(
                        targets=[
                            Name(id='e', ctx=Store())],
                        value=Constant(value=9, kind='int'))]])],
    type_ignores=[])
```

The output files will look something like this, with the top part of the file indicating the code parsed, and the bottom part of the file displaying the text representation of the AST that it has been parsed into. Note that during the test, the output is **verified** by converting the AST back into normal source code and ensuring that it is the same as the original input file.

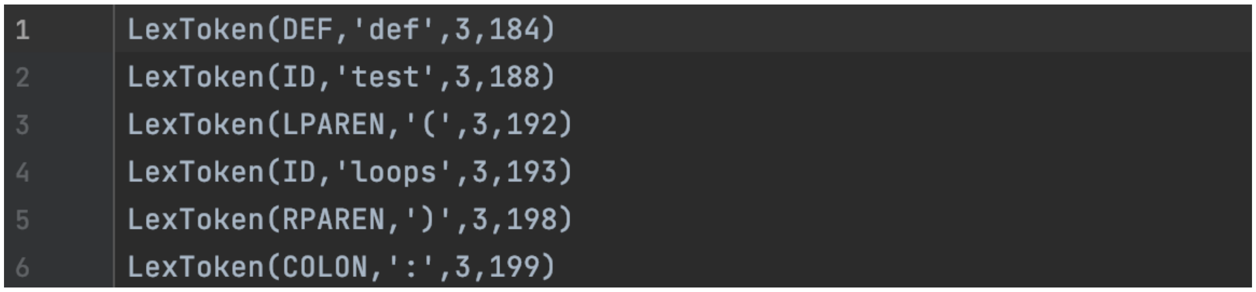
You might notice that these tests are only syntactically correct python, but most of it is not semantically correct, this is intentional as we are testing the parsing capabilities of our parser. (this means that you might see tests like “a = b” when b is not defined, etc)

We also have the ability to check our lexer by running “make lexer” and then looking at the **/out** directory again:



```
(venv) hhu@07065a-hhu:~/school/488_project/sprint1$ make lexer
for f in $(ls code_examples/); do python3 lexer.py code_examples/$f > out/$f.out.txt; done
(venv) hhu@07065a-hhu:~/school/488_project/sprint1$
```

The directory will contain the tokens produced by our test files in **/code_examples**, and will look something like this:



```
LexToken(DEF,'def',3,184)
LexToken(ID,'test',3,188)
LexToken(LPAREN,'(',3,192)
LexToken(ID,'loops',3,193)
LexToken(RPAREN,')',3,198)
LexToken(COLON,':',3,199)
```

These test files are then verified by manual inspection (although they are verified automatically as part of the parser tests).

## BNF Grammar

```
letter = "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" |
"j" | "k"
| "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" |
"v" | "w"
| "x" | "y" | "z" | "A" | "B" | "C" | "D" | "E" | "F" | "G" |
"H" | "I"
| "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" |
"T" | "U"
| "V" | "W" | "X" | "Y" | "Z"

number = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
"9"

float = {number} number "." {number}

boolean = "True" | "False"

type = letter | number | float | list | boolean

list = "[" [type {"," type}] "]"

expr = type
    | "range"(number)
    | expr binary_op expr
    | unaryOp expr

binary_op = "+" | "-" | "*" | "/" | "<" | "<=" | ">=" | ">" |
"=="  |
"and" | "or" | "+=" | "-=" | "*=" | "/=" | "%"

unaryOp = "-", "not"

compound_stmt = function_def
    | if_stmt
    | for_stmt
    | while_stmt

assignment = ID "=" expr
```

```
statement = compound_stmt | assignment

ID = letter | ID letter  | ID number | ID "_"

function_def = "def" ID "(" [params] "):" NEWLINE
               INDENT {statement} NEWLINE
                "return" expr DEDENT

base_type = "int" | "float" | "bool" | "str" | "list"

params = ID [: base_type]

if_stmt = "if" expr ":" NEWLINE
            INDENT {statement}  DEDENT
    | "if" expr ": NEWLINE
    INDENT {statement} DEDENT
      elif_stmt
    | "if" expr ":"  NEWLINE
    INDENT {statement} DEDENT elif_stmt
    else_block
    | "if" expr ":" NEWLINE
    INDENT{statement} DEDENT
    else_block

elif_stmt = "elif" expr ":" NEWLINE INDENT{statement} DEDENT
    | "elif" expr ":" NEWLINE INDENT{statement} DEDENT elif_stmt
    | "elif" expr":" NEWLINE INDENT {statement} DENDENT
else_block

else_block = "else" ":" NEWLINE INDENT block DEDENT

while_stmt = "while" expr ":"  NEWLINE
                     INDENT {statement} DEDENT

for_stmt = "for" target "in" expr ":" NEWLINE
        INDENT {statement} DEDENT
```