Group 4: Sprint 2 Report (Backend)

Group Goals for Sprint:

- **Primary:** Defining our Intermediate Representation and converting an AST to our IR.
- **Secondary:**
  - AST to target
  - Intermediate Representation back to AST

Intermediate Representation (IR)

To get our IR, we traversed the AST generated by our Parser using a *post-order* traversal. So for each node, we checked its children (body, conditions, depending on the node) before checking the root. Since we used the built-in Python `ast` module in our parser to build our `ast`, it was easier for us to extend the ast module's `NodeVisitor` class to implement our traversal.

We used the following process when converting our AST to IR:

- Single memory addresses such as constants (ints, strings, bool, floats, ids) were kept as they are, so a constant `x` was represented as `x`.
- For more complex statements such as binary operations, assignments statements - we used **temporary variables** prefixed by **`t_`**. This allowed converting operations which used multiple variables to just use 1 variable recursively. For our implementation, we decided to use a global variable (in the ASTVisitor class) to keep a count for how many temporary variables we created, and used this to generate a new variable each time. We understand that this can

be optimized further to reset variables, but did not prioritize doing so in this sprint. This is something we will consider looking into in the next sprint.

- We represented **blocks of code** such as blocks in the whole module (globals), inside if-statements, loops and functions as *keys* in a dictionary. We used the 'main' key to store all outer code, and prefixed blocks in if-statements, loops and functions with **'_L'** (local). This allowed using statements such as GOTO {block} when exiting another block or using TAC to describe which if-block to go to (described more in examples).

To explain our conversion better, we'll use some examples:

- Example 1: simple expressions

```
x = 1 + 2 / 3
y = x * 5
```

TAC conversion:

```
{'main': [ ('/', 2, 3, 't_0'),
           ('+', 1, 't_0', 't_1'),
           ('=', 't_1', None, 'x'),
           # space for clarity
           ('*', 'x', 5, 't_2'),
           ('=', 't_2', None, 'y')]}
```

Explanation: the keys in the dictionary are the different scopes or blocks of code (in this case we just have the main scope). Each associated value is a list of statements in that scope.

We parse the first statement `x = 1 + 2 / 3` by creating a temporary variable `t_0` to represent `2 / 3` since this itself has 2 memory addresses. We then assign the temporary variable `t_1` to represent the operation `1 + t_0`. We then represent the whole assignment statement as `a = t_1` which in our tuple representation is `('=', 't_1', None, 'x')`. We use the same process for the next statement. Note that for this we just needed 1 temporary variable (`t_2` referring to `x * 5`) to represent the statement with 3 addresses.

- Example 2: if statements

```
x = True
z = False
if x:
   y = 2
elif z:
   y = 5
else:
   y = 7
a = y * 2
```

TAC Conversion:

```
{'main': [('=', True, None, 'x'),
          ('=', False, None, 'z'),
          ('IF', 'x', None, '_L1'),
          ('IF', 'z', None, '_L2'),
          ('=', 7, None, 'y'),
          ('GOTO', None, None, '_L0')],
'_L1': [('=', 2, None, 'y'),
        ('GOTO', None, None, '_L0')],
'_L2': [('=', 5, None, 'y'),
        ('GOTO', None, None, '_L0')],
'_L0': [('*', 'y', 2, 't_0'),
        ('=', 't_0', None, 'a')]}
```

Explanation: Just like the last example, the first two assignment statements here are in the main scope. We then have `('IF', 'x', None, '_L1')` representing that we go to the `'_L1'` block if `'x'` is True. The next statement (representing the elif) similarly says we go to `'_L2'` if z is True. Since need to execute the code after the if, elif and else block but aren't in main if the statements are True – both `'_L1'` and `'_L2'` point to the `'_L0'` block at the end representing that we move to `'_L0'` (the remaining code in main) after the blocks are executed. The `('=', 7, None, 'y')` in the main block represents the statement in the else block, and just like `'_L1'` and `'_L2'` points to go to `'_L0'` after the else block is executed.

- Example 3: Loops

```
x = 0
for i in range(5):
   x = x + i
```

Conversion:

```
{'main': [('=', 0, None, 'x'),
          ('PUSH-PARAM', None, None, 5),
          ('CALL', None, None, 'range'),
          ('FOR', 'i', 'ret', '_L1')],
'_L1': [('+', 'x', 'i', 't_0'),
        ('=', 't_0', None, 'x'),
        ('GOTO', None, None, '_L0')]}
```

Explanation:
We deal with main components here: function calls and loops. We first pust the parameter 5 so we can use it as a parameter to the function call made next. We use this for as many parameters as needed, and then reset after the call is made. We then use the return value as the iterator in our FOR loops TAC which has 4 components: FOR, loop variable, iterator, and loop block/scope. In this case we say the variable i loops over an iterator (which happens to be the last return value – like a return register), and the scope for the loop is in '_L1'. Just like if-statements described above, the loop block ends with a GOTO so we can go back to any remaining main-function code if needed. Since we don't have any in this case (no code after the loop), this '_L0' doesn't exist in the dictionary.

```
i = 0
while i < 5:
   i = i + 1
```

Conversion:

```
{'main': [('=', 0, None, 'i'),
          ('<', 'i', 5, 't_0'),
          ('WHILE', 't_0', None, '_L1'),
          ('GOTO', None, None, '_L0')],
'_L1': [('+', 'i', 1, 't_1'),
```

```
        ('=', 't_1', None, 'i'),
        ('GOTO', None, None, '_L0')]}
```

Explanation:
We represent while loops with (WHILE, cond, _, Block). In this case, ('WHILE', 't_0', None, '_L1') represents that we execute the block L1 as long as condition t_0 is True. Just like if-statements (and for loops) - we go back to remaining code in the main function (represented by _L0) after this block is executed. In this case there is no remaining code, so _L0 is not a key in the dictionary.

Example 4: Function definitions

```
def foo(x):
   y = x + 5
   return y / 2
z = foo(5)
```

Conversion:

```
{'main': [('DEFN', None, None, 'foo'),
          ('PUSH-PARAM', None, None, 5),
          ('CALL', None, None, 'foo'),
          ('=', 'ret', None, 'z')],
'foo': [('ADD-PARAM', None, None, 'x'),
        ('+', 'x', 5, 't_0'),
        ('=', 't_0', None, 'y'),
        ('/', 'y', 2, 't_1'),
        ('RETURN', None, None, 't_1')]}
```

Explanation:
We represent a function definition in the main function with (‘DEFN’, _, _, function_name). In this case, we’ve defined the function foo. We add the function name as a key, and map it to the code in its body/scope. We add each required parameter for the function as ('ADD-PARAM', None, None, parameter).

Our script 3ac_to_code converts the TAC generated back into an AST. We use the AST’s unparse function to confirm that our results match the original code (since no optimizations have been made yet).

IR to Target

Our target generator takes as input our AST (which has gone to three address code and back) which it then converts to one line of python source code.

The goal of our project is quite simple: take the AST representation of some python source code and produce an equivalent **line** of python code that does the same thing. Implementing this was quite a different story, and we ran into multiple challenges which we had to find creative ways to solve along the way and it turned out to be a lot harder than we initially estimated. (but we prevailed!)

The basis of our target generator is very straightforward, traverse through the AST and recursively translate each bit of functionality into a single line. If we assume that each node can be translated to a single line of code that can be inserted into an existing line of code, then we can recursively "translate" all of the nodes and wrap them around each other. Sort of like a huge code onion, albeit an ugly onion at that:

```
(lambda _Y: (lambda test: (lambda c: print(c))(test(5, [1, 2, 3, 4, 5])))(lambda a, b: (lambda new: (
    lambda _term1, _items1: (lambda i: (lambda i: (lambda _loop1: _loop1(i))(_Y(lambda _loop1: (lambda i: (
        [new.append(i), (lambda i: _loop1(i))(next(_items1, _term1))][-1] if (i % 2) == 0 else
        [new.append((i + a)), (lambda i: _loop1(i))(next(_items1, _term1))][-1]) if i is not _term1 else new))))(
        i if "i" in dir() else None))(next(_items1, _term1)))([], iter(b)))([])))(
    (lambda f: (lambda x: x(x))(lambda y: f(lambda *args: y(y)(*args)))))
```

Fig 1. An example of some code that has been "one-lined".

We are proud to say that **our target generator supports the full input specification we set out to achieve** in our proposal (see our sprint 0 document), and we are working on even more support for the upcoming sprints!

The test inputs provided in **/code_examples** aim to test the full subset of inputs that our target generator can support, we do this by setting a flag for each handler when that handler is invoked, and we check that all flags have been set after testing to ensure that we have hit every handler. More information on how to run our scripts are provided in the section below.

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 2 folder, you can run `make test` to run the testing script like so:

```
(venv) hhu@07065a-hhu:~/school/488_project/sprint2$ make test
python3 test_compiler.py
test_assignments ------------------- Passed
test_loops ------------------------- Passed
test_ifs --------------------------- Passed
test_comprehensive ----------------- Passed
test_real_files -------------------- Passed
test_unparse_files_multiple -------- Passed
All 6 tests passed.

Please see the "/out" directory for produced artifacts.
```

This runs the tests and writes the IRs and artifacts generated to output files to files in the **/out** directory.

As a quick recap, our project aims to convert regular python files into still runnable python files that are only 1 line long but preserve the functionality of the original file. This means that whatever the original file was doing before, our “one-lined” file should continue to do the same thing, even though the source code is only 1 line long.

Our final output artifacts (**which are written to /out after running our tests and have filenames that end with ".flattened.py"**) look like this:

```
# below is the 1 line version of the file "./code_examples/function_defs_and_calls.py"
(lambda test: print(test(1, 2)))(lambda a, b: (a + b))


# if you execute this file, the output will be the same as the original file, which is as follows:
"""
3
"""

```

The first line contains a comment indicating the original file that this output was generated from. The second line contains the one lined output that our compiler generated. Note that we aren't cheating here by using exec(), semicolons, or any newline characters, it's just good ole' lambdas 🙂

The following lines contain the expected output of the one line of code, which is the same as the original file, and can be verified by running the file. We also verify this when running the tests. We've shown a relatively short example in this document, but there are more interesting examples in our test suite which you can look at after running the tests. As part of our testing, we've also made sure that every handler of our target generator is tested at least once by our test suite, so our test coverage should be quite good.