Sprint 4 - Optimization and Final Submission

What we did for the sprint:

1. Optimization:
    a. Unused Variables
    b. Shortened Names
2. Parser Fixes
3. Additional Target Language Support

Optimization

We worked on adding 2 features to optimize our code: removing unused variables and shortening all variable names. Both features were added post converting our AST to our IR (three address code).

Unused Variables

To remove unused variables, we went through our Three Address Code and mapped each declared variable to where it is used next in the same block. So for example, with

```
x = 3
y = x + 4

z = 5
print(z)
```

We generate the TAC:

```
{'main': [('=', 3, None, 'x'), ('+', 'x', 4, 't_0'), ('=', 't_0', None, 'y'), ('=', 5, None, 'z'), ('PUSH-PARAM', None, None, 'z'), ('CALL', None, 1, 'print'), ('=', None, 'ret', 't_1')]}
```

And the unused variable optimizer generates the mapping:

```
{'x': ['t_0'], 't_0': ['y'], 'y': [], 'z': ['call'], 't_1': []}
```

Here we know that x is used in the definition of the variable t_0, t_0 is used in the definition of y, and y *isn't used anywhere*. However, 'z' is used in a function call – so we keep it, but remove both x and y. The output generated after converting this optimized TAC to Python code (for easier viewing) would then be:

```
z = 5
print(z)
```

(Note: we are showing this instead of the TAC for easier understanding, but actually the TAC is converted to an AST that we use in the target language).

In this case, we did not remove z because it was used in a function call. We also look for variables used inside lists or indexes and avoid removing them. For example:

```
x = 1 + 2
y = x + 3
z = 1
a = z - 4
b = z + 4
lst = [a, b]
print(lst[a])
```

Is optimized to become

```
z = 1
a = z - 4
b = z + 4
lst = [a, b]
print(lst[a])
```

We also do this for other blocks, such as IF blocks, loops, and functions to remove nested variables that are unused.

Issues

Scope: it's inefficient to check if a variable is used in a nested block given our current design. It's one of those things that we *know* how to add support for, but given the time, we prioritized fixing things such as our parser and cleaning things up for the final submission instead of working on additional features. So, unfortunately, for something like:

```
x = True
y = False
```

```
if y:
   if x:
       print("Hello")
```

The optimizer *currently* would remove the x. Now, this is something we *know* how to fix - but given our TAC definition, and the way we loop over nested blocks plus issues we faced in sprint 3, we chose to hold back on this for now, and focus on cleaning code.

Name remapping "optimizer"

This is a unique optimization that remaps all variables into a smaller length variable name while keeping the same logic.

We first create a generator function that returns the next shortest length string given an alphabet. It exhausts all possibilities for single length strings, and then moves to 2 length and so on. Thanks to this generator, we're able to assign new
This was proven to be easier than expected with three address code, because we were able to isolate which tuples could contain variable names, and mapped it accordingly.

This optimizer can be found in **`tac_shorten.py`** . There are corresponding tests in **test_compiler**, test_tac_shortener, which uses our original code examples and runs it to see that there's no real difference in output; these can be run using **make test** in the sprint4 directory.

The optimizer can also be run separately, by running **`python3 tac_shorten.py`** which uses the code in **test_input.py** and runs it through the shortener.

Parser Fixes

We had 32 shift/reduce conflicts in our last sprint.

We realized we got some of them because our pattern for method calls allowed {expr} DOT function call – but floats also have a dot, and could be parsed as {number} DOT … – which caused an ambiguity. To avoid this, we only allow method calls for lists, and ids.

Our other conflicts were caused by precedence issues. After a lot of thinking, we realized that the reason we got shift/reduce conflicts was that there was ambiguity in what to do after seeing

an expression - i.e. precedence issues. The reason was that we hadn’t added all the tokens that can occur after an expression in ply’s precedence list (like LBRACE and IN). After adding them in, this was resolved.

We removed unused tokens from our lexer, except the WS token. The reason for keeping it was that even though our parser grammar doesn’t need this token, we need it in our post-processing step to make indentation tokens which we need in our grammar.

Changes to the target generator

This sprint, we added even more support for other features of the language to the target generator. We wanted this project to be as complete as possible, so we endeavored to support as many features as we possibly could. With this final sprint, we are happy to say that we've added support for **default arguments**, **recursive class initializers**, **subclasses and inheritance**, **ImportFrom statements** (`from x import y as z`), **multiple and simultaneous assignment** (`a, b = b, a`), **for loop target unpacking** (`for x, y in z:`), **chained comparisons**, and **function decorators**. Again, our website (https://flatliner.herokuapp.com/) that we built in sprint 2 to demonstrate our compiler continues to be updated as we add more features, so feel free to use it to test things out. We have a cool example pre-loaded on the website that you can run to see our compiled code in action, all you need is a python shell, which we've linked to on the website as well.

Example inputs and testing script

Our examples and testing procedure continues to be the same as previous sprints (with added tests and test cases).

To run our testing script, cd into our `/sprint4` directory and run "`make test`". This runs the tests and writes any produced output files into the "`/out`" directory, where they can be viewed and run (for those that produce an executable python file). A more in depth explanation of our testing script and procedures can be found in our sprint 2 document.

Likewise, running the same command produces a *.tac.py file that holds the TAC representation of each of the examples in `/tac_ast_examples` . All test files are converted to TAC, and then reconstructed and verified to ensure they produce the right ASTs. For this sprint, we also output the optimized TAC and code, so that they can be compared.

Concluding remarks

While we initially set out to only convert a subset of python that only included functions and a few built in data types, we've consistently pushed ourselves every sprint to include more functionality than we planned. We can now convert rather complicated programs, and are quite proud of the milestones that we've accomplished along the way. We hope you have enjoyed this journey with us, and perhaps even learned something as well!