Sprint 3 - Type extension and more supported features

What we did for the sprint:

1. Scope with TAC – improvements and fixes
2. Supporting non-primitive types: Lists, list methods, slicing, indexing
3. Added support for more features in the target generator

Scope

When we tested our three-address code scripts with bigger test files, we figured that we weren't handling scopes correctly in some cases. The biggest problem for us was figuring out what code to execute after we'd entered a block of code in a while loop, for loop or if-statement. As we mentioned in our design last time, the way we planned it out was to include a GOTO statement at the end of a block (while/for/if) that would indicate which block of code to execute next. This means that any separate segment of code is split into a block of code, which we will refer to as a bucket (L{number}), for easier understanding. However, we found out that in some cases this block was inaccurate, or wasn't placed.

Fixing this issue required more insight into how we wanted to structure out TAC. We decided that we wanted to explicitly tell which hops to go to, in comparison to the standard TAC model that assumes the next label is L{number + 1}. This brought many challenges such as figuring out the next hop to go to; and how nesting control flow statements would pan out.

We support control flow and long jumps for the following operators: Ifs, Whiles, Fors, Function definitions. The general flow for all these control flows is we provide a continuation (which we will refer to as an end segment throughout this document) for which part of the code the statement should go to next after handling it's internal states.

These control flow statements are handled in different ways:

Ifs

If statements have the ability to be chained, using elif statements, so the way we solved this is to allocate a separate bucket prior to processing each bucket, this allows us to ensure each if statement is read in a sequential order, and allows for the else statement to be apart of the original bucket, which follows the same TAC design show in class.

After allocating a bucket, we process the body of each if statement independently. We also pass in the end segment, so that each body knows exactly where it needs to go after it finishes processing its code.

The else statement is appended to the original block of code, as per the standard TAC pattern, this allows for easier parsing since there isn't a separate bucket and goto statement to consider.

Whiles/Fors

Whiles and for loops have the same control flow behavior. We append a GOTO at the end of the control flow's bucket, to indicate that after this loop ends it should go to this bucket for the next continuation.

Parsing the body requires more work. The behavior changes depending on if it's a control flow statement or not.

If the next statement isn't a control statement, then we can parse it normally and add it to the same bucket.

If the next statement is a control flow statement, this indicates that we split the code off into different flows, so we need to allocate a new end segment that the rest of the code can send to continue the flow.

However, there is one edge case to consider, which is that if this statement is the last available statement then it doesn't have a new segment to continue to, rather it's continuing code would be the original end segment that was designated (so that all flows converge back to the next statement line)

Function Definitions

In our previous implementation, we only included a DEFN statement and parsed the function independently.

However, with the introduction of control flow statements, we needed to ensure that the statements after the function definition ended up being added to main's bucket, which was done by setting the key back to its previous.

With these implementation details in mind, we were able to incorporate nested control flow statements, with our original codebase.

Nesting flow control elements

One major concern we encountered was the idea of nesting control flow statements. Our TAC to code converter reads the GOTO that shows up at the end of the list as it's end segment. This means that all sibling nodes are connected together by a chain of GOTO's in a linked list manner. Figuring out when to add a GOTO, and to which end segment, proved to be an interest challenge. We solved this by telling each recursive call whether it should include a GOTO or not.

Only the first level control flow statements, that are not the last statement, should include a GOTO since it requires a split from the same branch. However, any nested condition/the last statement in the branch should not include a GOTO, because the flow ends there.

Supporting Lists

To support lists, we first had to go back to our parser to recognize lists as an expression. We also had to add support for indexing and slicing. While indexing behaves close to how Python behaves, supporting all expressions inside, we kept our slicing limited and aren't supporting "step" arguments as of now for parser simplicity. We will consider adding this support in our last sprint, but didn't prioritize it during this sprint.

The trickier part was adding support for lists to our three address code. For this, we used a similar approach as what we did for function parameters. We use a token - START-LIST _ _ {temp_variable_name} to indicate that a list is beginning. The next lines either put an element on to the list, or do some prep-work (like binary operations, calls) to push an expression on to the list. Lastly, we terminate the list-block with a END-LIST to indicate that we are done pushing elements to it.

An example of this is:

```
lst = [1, 1 + 2, 3]
```

This gets translated to:

```
{'main': [
('START-LIST', None, None, 't_0'),
('PUSH-ELMT', None, None, 1),
('+', 1, 2, 't_1'),
('PUSH-ELMT', None, None, 't_1'),
('PUSH-ELMT', None, None, 3),
 ('END-LIST', None, None, 't_0'),
('=', 't_0', None, 'lst')]}
```

In the example above, we push 1, the temporary variable t_1 which represents a binary operation (addition), and 3 on to the list.

The biggest challenge for us here was figuring out how to make nested lists work, since that could start another 'START-LIST' block inside. To recognize when the *outer* list was ending, and differentiate that from the *inner* list - we used temporary variables like *t_0* in the example above. Using temporary variables also makes it easy to assign the list to a variable for assignment statements.

For slices and indexes, we use similar blocks which start a slice with ('SLICE', list_variable, None, temp_variable for slice). It is followed by a line which has ('SLICE', start, stop, end). For example:

```
lst[1:2]
```

Gets translated to:

```
('SLICE', 'lst', None, 't_1'), ('SLICE', 1, 2, None)]
```

Here the second variable represents what we are slicing, and the temporary variable makes it easy to assign the slice to another variable.

Similarly:

```
lst[1]
```

Gets translated to:

```
('INDEX', 'lst', None, 't_1'), ('INDEX', 'lst', 1, None)
```

This is just like the slice format - but the difference is that instead of the start, stop and step - the following line has the index, and the list being indexed.

Changes to the target generator

Since the target generator already supported lists in the last sprint, this sprint we added even more support for some other features. Among them, we now support **keyword arguments**, **raise and assert** statements, along with **break and continue** keywords. The website (https://flatliner.herokuapp.com/) that we built last sprint to demonstrate our compiler continues to be updated as we add more features, so feel free to use it to test things out.

Among the new features added, break and continue were the most challenging, and they were implemented by treating them as continuations of the program, where break jumps to code that would be executed after the loop, and continue jumps to the beginning of the loop. This gets a bit more complicated when you consider nested loops with different breaks and continues at different levels of nesting, but the core idea remains the same.

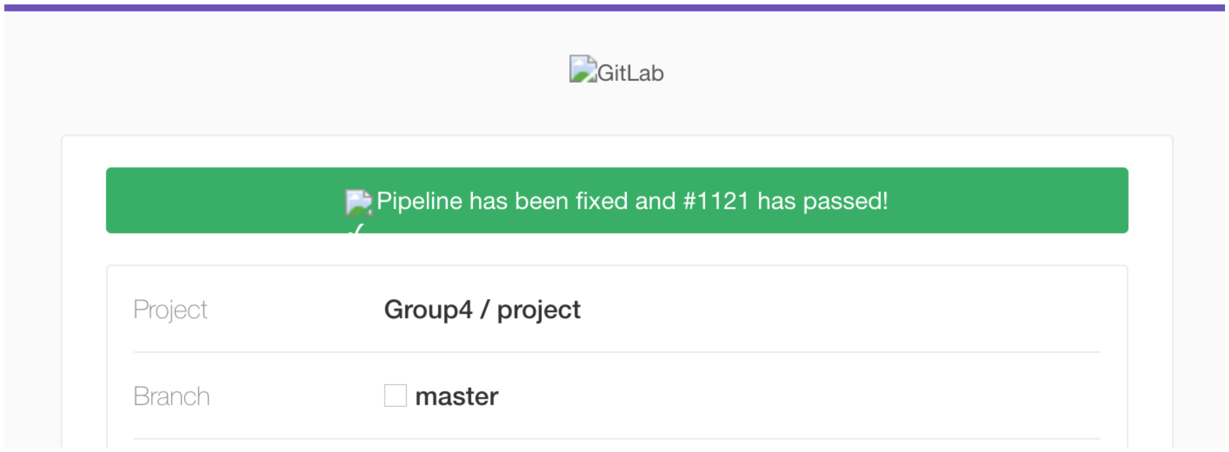
Example inputs and testing script

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 `/sprint3` 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’ve also set up CI to run our tests on every push just in case.



GitLab

Pipeline has been fixed and #1121 has passed!

Project	**Group4 / project**
Branch	**master**