This documentation can be found in the group's wiki: https://mcsscm.utm.utoronto.ca/csc488_20221/group9/project/-/wikis/Sprint-4-Documentation

List Generation

We used C++ vectors to generate our Python lists. We chose vectors because vectors are a dynamic structure in C++, meaning internally the vector is reallocated automatically if more space is needed. Therefore, we figured vectors would work most similarly to Python lists and would let the user keep most Python list functionality when compiling to C++.

```
lst = [1, 2, 3]
lst.append(4)
lst.pop()
x = len(lst) + lst[0]
```

converted to:

```
vector<int> lst{ 1, 2, 3, };
lst.push_back(4);
lst.back();
lst.pop_back();
x = lst.size() + lst[0]
```

where append is converted to push_back, and pop() is converted to back() and pop_back(). pop() is converted into two functions because pop_back which does the popping does not return the element that is popped, so we call back() to handle the returning. We also support Python's len() function where it is converted to the size() function, and list indexing.

For Loops Functionality Expanded

This sprint we also expanded on our current for-loops functionality. Before, for-loops could only handle hard-coded numbers, but now it can handle variables and expressions as well. For example, the following python code:

```
l = [1,2,4,5]
for i in range(len(l)):
        print(l[i])
```

Will result in the equivalent C++ code:

```
#include <stdio.h>
#include <iostream>
#include <string>
#include <vector>
using namespace std;

int main() {
        vector<int> l{ 1, 2, 4, 5, };
        for (int i = 0; i < l.size(); i += 1) {
                cout << l[i] << endl;
        }
        return 0;
}
```

Optimizations

We implemented the Constant Folding optimization. We use this optimization to reduce complex arithmetic expressions. For example, this expression:

```
x = 5 * 4 + 2 - 6 / 2
```

will get reduced to the following in C++: ```C++ int x = 19;

```


Note that we if there is a variable in the expression, we won't apply constant folding. An example of this would be:
```python
x = y * 4
```
```

In our presentation we stated that we had implemented Constant Propagation but we ended up having to remove it, due to a bug we found with it. Originally, our Constant Propagation substituted variables with their value if the value was a constant. However, we didn't account for the case when that variable is a parameter given in a function. When function parameter variables were used in an expression, our entire expression wouldn't get generated properly (it would show up as `None`). We didn't have a way to get around this because we weren't able to detect when a variable is a function parameter or not. Therefore, we opted to just remove Constant Propagation altogether.

Furthermore, we only apply Constant Folding to arithmetic expressions which are stored in variables. If the expression is just the return value of a function, then it won't get folded.

Finally, our Constant Folding optimization doesn't modify our IR that much. Only the final variable which contains the arithmetic expression will get reduced. The temporary variables themselves will stay the same (and won't get removed either).

Testing

To test our compiler, there is a test_suite_script.py which will run a test suite on multiple test cases. To run it, use the command

```
python3 test_suite_script.py
```

You will find the the input for these test cases in the `tests` directory and the output in the `tests/output` directory. The input test files will be named in the format *test_case_TYPE* where TYPE will be PASS or FAIL. The PASS tests are for cases that will successfully generate an IR and the FAIL tests are for the cases that will fail at some step in our compiler.

The output files are named in the following format: "*test_case_name*_output.txt". In each output file, you will see each step of our compiler. To verify the target code generation, the output section will be highlighted with text similar to: "==== TARGET CODE ====". All other output sections will appear within the output file labeled similarly to the target code section.

To verify the target code, compare the target code to the source code line by line and verify that each line of python code has been correctly translated into C++ code. If the compiler raises an error, e.g. a variable being used before declaration, verify that the error is valid using the error message.

To test our Constant Folding, there are many optimizable statements in the full_test_PASS test case such as:

```
x = 1 + 2 * 3
y = 3 + 2 + 3 + 5
z = 3 - 2 / 4
```

To test our list to vector functionality, see test cases list_PASS and list_incorrect_type_FAIL. list_PASS tests a variation of different list operations, and list_incorrect_type_FAIL attempts to initialize a list with multiple types.

Limitations

- list functions are not validated, meaning you could append a data type to a list that does not conform to that list's type
- optimization limitations mentioned above in the optimization section