This documentation can be found in the group's wiki: https://mcsscm.utm.utoronto.ca/csc488_20221/group9/project/-/wikis/Sprint-3-Documentation

Target code generation

We decided to generate target code directly from our TAC IR of the source code. We decided to employ this method because the TAC format is similar to the structure of C++ code, since it is a lower level language. We used a strategy that involves only one pass over the IR to produce the target code.

The strategy goes as follows: we take advantage of the simplicity of the TAC representation of our source code to use regex to parse the different kinds of expressions. For example, these expressions can take the form of variable assignments which includes assignments to temporary registers and variables, or if statements with the cond and labels indicating the branching. We then go line for line matching the different expressions with our regex and translating each TAC into its corresponding C++ code. We use two pointers, where each pointer dictates where the next line of code goes. One is a global pointer and the other being a main function pointer. As the names suggest, the global pointer keeps track of where the next line of code goes in the global scope and the main pointer does the same with the main function.

An example of our end-to-end compiler is where our compiler takes an input of this python code:

```
 x = 2
y = x + 2 + 3 + 5
i = 0

def do_something(x: int, y: bool) -> int:
        l = [1,2,3,4]
        if x == 1000:
                if x == 100:
                        if x < 102:
                                if y:
                                        print(x)
                while y:
                        if x == 0:
                                x -=1
                                y = False
                        print(x)
        elif x < 0:
                return x + 100


        return x * 99

def print_sum(x: int, y: int, t: int):
        print(x+y+t)


x = do_something(x, False)
print_sum(x, 4, 100)
```

And translates it into the following C++ compilable code:

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

int do_something(int x, bool y) {
        vector<int> l{ 1, 2, 3, 4, };
        if ((x == 1000)) {
                if ((x == 100)) {
                        if ((x < 102)) {
                                if (y) {
                                        cout << x;
                                }
                        }
                }
                while (y) {
                        if ((x == 0)) {
                                x = (x - 1);
                                y = false;
                        }
                        cout << x;
                }
        }
        else if ((x < 0)) {
                return (x + 100);
        }
        return (x * 99);
}
void print_sum(int x, int y, int t) {
        cout << ((((x + y)) + t));
}
int main() {
        int x = 2;
        int y = (((x + 2) + 3) + 5);
        int i = 0;
        x = do_something(x,false);
        print_sum(x,4,100);
        return 0;
}
```

This is a shorter version of the test under: `tests/full_test_PASS`, to test this you can do: `python pythonCompiler.py tests/full_test_PASS`.

Data structure implementation - Lists

We implemented the lists data structure during this sprint along with target code generation. We convert the following python list:

```
lst = [2, 3, 4]
```

into this

```
vector<int> lst{ 2, 3, 4}
```

Firstly, we made minor changes to our parser to make list handling easier. Since we planned to support lists from the beginning of the project, our grammar and parser already supported lists making this part trivial. Next, we must support adding lists to our symbol table. We deduce what type the list variable entry should be by checking the type of the first element. In the above example, lst would have an int entry. We then move on to type checking. In Python, lists consisting of elements of different types is allowed, however this is not the case in C++. Therefore, we restrict all list usages to be of a single type. We check each element of the list ensuring that it matches the type of the first element. If not, we raise an exception with an appropriate error message. We then generate the following string for its IR:

```
lst := [2, 3, 4]
```

Since there is no official TAC for a list declaration, we used this format to make this representation easy to write regex rules for. From this point, it is converted into a C++ vector of the appropriate type.

Testing

To test our compiler thus far, 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 either at the symbol table or type-checking level.

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.

Limitations

- methods will not have access to global variables because all global code will end up in the the C++ `main` function
- stray expressions won't be supported, ie. the `1 + 1` won't get translated into our target code

```
 1 + 1
x = 5
```

will get generated as

```
int x = 5;
```

- list elements can only be literal values or variables. expressions are not supported as elements and the list will not be parsed. For example, `5+2` here will cause the list to not be parsed

```
l = [s, 3, 5+2]
```

- long comparison expressions don't get parsed correctly in the generated C++ code, for example

```
x = True or False or 5 > 10
```

will get generated as

```
bool x = (((true || false) || 5) > 10);
```

- Function calls in arithmetic expressions don't work:

```
x = 3 + foo()
```

- for loops with a negative "step" parameter in the range function aren't supported:

```
for i in range(10, 4, -1): ...
```

Bugs

We have found multiple issues/bugs. These bugs, along with some of our limitations, are being recorded in the Issues section of our repo.