Sprint 2

IR Description

In our IR generator, TAC is the format we choose to follow, and data class in python gives better readability on top of TAC. Our generator imports the simple register and label operations from week 7 tutorial, such as `inc_register`, `inc_label`, `mark_label`, etc. After parsing and type-checking, we simply assume all codes arrived at the IR generator are syntactically and semantically correct.

In our implementation of AST, `if`, `elif`, and `else` statements are treated as individual statements, so those nodes do not track their subsequent statements. In this scenario, our IR uses two stacks, `cond_label_stack` to record the true label of `if statement` for jump instruction, and `cond_label_idx_stack` to record the true label's index in the list of IR. This index is later used for `elif` and `else` IR insertion. The true label, which is declared at the end `if` statement(head), will be used by all subsequent `elif` and `else`. To ensure the correct label wrapping, subsequent statements are inserted before the true label, and use the true label at the top of the `cond_label_stack` for jump instruction. So the true label is always at the end of a series of conditional statements. The choice of stack data structure supports nested `if` statements as well. Except for the above approach, our conditional statements work the same way as regular `if-else` statements.

Our compiler has three loop statements, `while` ,`for range` and `for list`, and all of these representations use the same logic as following, (1) store loop variant, (2)declare loop label, (3)check variant condition, (4) body, (5)update variant (while loop skip this), (6)go back to loop label. In our IR, while loop strictly follows these steps, whereas `for range` and `for list` has slightly more complicated loop variant initialization. In `for range`, we use three registers to store the start, step and stop value respectively, then update the variant using start and step value. Our `for list` supports both list (it is actually an array in our design) and tuple. The idea is to obtain the pointer of the list head, which has the length of list stored. Then use the length of the array and head pointer to iterate over the list. We used a simplified approach to represent the increment in list indexing. Increment in pointer respects to an increase of the size of the corresponding type in address.

For lists and tuples, a list/tuple "Head" is created with its own register as well as the length of the list/tuple. Then the individual elements are added with their own register's and values. The IR can be used to easily traverse a list since the length of a list is stored in the data class and each list element is stored in its own data class, "IR_List_Val"

Strings are considered a primitive type in our AST, similar to int or float. However in our IR it is treated differently, being stored in its own IR data class. It is treated similarly to lists/tuples, where the head of a string is stored first with its own register and string length. Each character in the string is stored in its own data class with its own register. This will be useful when translating into C since in C a string is a char array (with a null terminator).

Function declaration creates a new label. A parameter list is also created that operates similarly to how lists are stored in the IR. The parameter list has a head that stores its

register and the number of parameters the function declaration has. Then each parameter is stored in its own data class with its own register and parameter name. Return statements are stored separately from the function declaration.

Function calls store its argument list similarly to how function declarations store its parameter list. The argument list starts with a head that has its own register and the number of arguments in the function call. Each argument has its own data class that stores its register, which is generated by traversing the individual argument. Function calls have their own registers, and it creates another register for the function return as well.

Testing

We use pytest to run our test cases. In addition to the previous sprint, we added test cases for the IR generator.

To run all tests:

1. Ensure you have pytest installed: `pip install pytest`
2. `cd sprint2/`
3. `pytest`

To run only IR generator tests: run `pytest ir_gen_test.py`

We have 10 test cases for the IR generator. Since the input is usually multiline, we create one input file for each test case. Alongside with input, we also have an expected output file that is being checked against during testing. We created a script that can automatically discover these tests.

The test cases cover most features we have in our language. There are simple test cases that only test a few language features at a time. And there are closer to real world tests that test lots of language features at the same time.

Specifically, the test cases for the parser is organised as follows:

- `tests/<test_id>_input.py`: parser input.
- `tests/<test_id>_output.txt`: expected parser output
- `tests/<test_id>_received.txt`: actual parser output. This file is not saved in version control and is automatically cleaned up between each run.
- `tests/<test_id>_received.diff`: a diff between the expected output and actual output. This file only exists if they are different.