Sprint 3

1. **Code Generation**
2. **Type Extension (Tuple, Array)**

Code Generation

Our strategy for generating target code is reverse-engineering. We convert our IR to a C_AST, a AST designated for C language, then we translate the C_AST to our target code with some helper function written in C.

Primitive type translations are straightforward. We use AST to map `int` to `long long`, `float` to `double` and `boolean` to `int`.

From our IR to C_AST, we use the symbol table from the type checker, as well as a temporary symbol table in C_AST. The main idea of the temporary symbol table is for hashed function names. Our compiler allows function declarations with the same name, but different parameter names and types, which is prohibited in C. We give each function declaration a unique id and then map the original function name to the hashed name, in order to achieve this functionality. Due to the nature of TAC, our compiler lost much information about the data types during the translation from AST to IR. The type checker symbol table is used for retrieving data types, such as elements' types in list and parameter types in function.

In many of our translations from IR to C_AST, we used look-ahead to wrap the whole block in one place. For instance in `if-elif-else`, all subsequent `elif` and `else` must have the same true label jump as the `if`. We then use this label as an index to manage the number of `elif` in this block.

Our implementation of while loop in IR uses labels and `if-statement`. So the actual translation is similar to the `if` statement, except that we use designated label names to distinguish different blocks, such as `while`, `for` and `function`.
######
ADD SOMETHING ABOUT FORLOOP####

The code generation from C_AST is mainly done by the string formatter. We implement the same node visitor pattern as in type checker to process the generation. In this process, we split `function declaration`, `function definition`, `main code` and `clean up` into different sections so that we always have functions declared (and not necessarily defined). The translation in this step is to dissemble the C_AST then fill it in string format written in C syntax. We have a `starter.c` file that is `include`-ed in the first line. It defines all the helper functions, `typedef`, and various other code that make the generated code work.

Type Extension

The types we chose to extend are `tuple` and `array`. So far, our compiler only compiles statements and individual expressions, so we believe it is nice to have some types which can store a collection of values.

In our compiler, `tuple` must store the data of the same type as well. Sothe difference between `tuple` and `array` is immutability. This immutable property and strict same data type in one collection is checked in the type-checking stage. For checking immutability, we distinguish `tuple` and `array` in the `name` attribute of `NonPrimitiveLiteral` AST node. For checking data type, we iterate over the `children` attribute of the AST node to make sure all of the child elements have the same type or same as the LHS type in `assignment`. We then use this attribute to check if all modification operations are made on `array`. After the type checking, we are safe to compile `tuple` and `array` in the exact same way in later layers.

We have two IR objects used for these two extensions, `IR_List` and `IR_List_Val`. The first object notifies our compiler that we are about to process a non-primitive and the length of the non-primitive. Then `IR_List_Val` will appear `length` times after the head with the value of the element stored in this object.

In the translation from IR to C_AST, we used the symbol table from type checker to get the types of empty non-primitives, as the IR itself does not record the types of its children. According to our typechecker structure, the empty non-primitives only have type when they are used in some expression that specifies the type, such as assignment, function declaration, etc. Therefore, an empty non-primitive itself is currently not allowed in our compiler due to the lack of type information.

Besides target code generation files, we also wrote helper functions in C for these non-primitive values. Both `tuple` and `list` are represented using `struct list`. This struct has three properties, `data_t data`, `int_t length` and `int_t uninitialized_length`. The first property stores the value in the non-primitives and its type `data_t` is a union of all types our compiler supports. The `length` and `uninitialized_length` are used in initialization and array operations. Current helper functions support non-primitive initialization, get and add. Initialization initializes the values on the heap and makes sure the `uninitialized_length` is set to zero after this process. Get operation uses the address indexing in C. Add operation realloc space for the non-primitives then add the additional value to the end of the pointer.

In the next sprint, we will allow nested arrays and more operations on the array, such as index slicing, count, etc. Due to the time constraint, we did not complete the other type extension `string` in the target code conversion stage. We will complete `string` extension and corresponding operations in the next sprint as well.

Testing

We use pytest to run our test cases. We have the following test suites and in total 87 test cases.

- `test_lex.py`
- `test_yacc.py` along with files in `tests_yacc/`
- `test_ir_gen.py` along with files in `tests_ir_gen/`
- `test_C_AST.py`
- `test_c_gen.py` along with files in `tests_c_gen/`

To run a single test suite: `pytest test_XXX.py`
To run a single test: `pytest test_XXX.py::[test_name]`

Since this sprint we created the `test_c_gen` test cases, we explain the setup here. Four files may exist for any test case:

- `<test_name>_input.py` is the input from our input language
- `<test_name>_ir_received.txt` is the IR generated (if successful)
- `<test_name>_ir_received.c` is the final C file generated (if successful)
- `<test_name>_ir_received` is the gcc-compiled executable (if successful)

Note: there is a bug with the tests where objects are shared between tests. The test must be run individually at the moment.