Changes in project scope from sprint 1

This documentation can be found at the repo's wiki page: https://mcsscm.utm.utoronto.ca/csc488_20221/group9/project/-/wikis/sprint-2-wiki

We made several changes in the scope of our project from sprint 1. They include changes in supported python function definitions, function parameters, python lazy class evaluation.

supported python function definitions and function parameters

We were originally planning on supporting all forms of python function definitions like:

```
 def pick(l, index):
def pick(l: list, index: int) -> int:
def pick(self, l: list, index: int) -> int:
```

Where each function may or may not have a return type, or may or may not have parameter types. However looking at our target language of C++, we figured this would be too time consuming and pivoted to having all functions having a return type otherwise they would be considered void functions, also every function parameter must have a type.

```
 def pick(): # valid but, is a void function
def pick(x, y): # is invalid because x and y don't have a type
def pick(self, x: int, y: bool) -> float: # valid
def pick(self, x: int, y: bool): # valid, void function
```

python lazy class evaluation

Another thing we are changing in the scope of our project is handling of python lazy evaluation in classes. Take this python code:

```
 class testClass:
        t = 2
        t_b = True
        def __init__(self):
                testfunc5()
                t = 3
                z = self.doSomething()
        def doSomething(self) -> int:
                return 3

def testfunc5():
        pass
```

This is a valid python program, even though testfunc5() is called in `__init__(self)`, because python uses lazy evaluation, by the time a `testClass` object is created after testfunc5, `testfunc5` has been defined. However if a `testClass` object is created before `testfunc5` definition, it gives an error. This is difficult to capture fully in C++ because the exact equivalent C++ program would be this:

```
 class testClass {
   public:
     int t = 2;
     bool t_b = true;
     testClass();
     int doSomething();
};


testClass::testClass(){
  testfunc5();
  t = 3;
  int z = doSomething();
}
int testClass::doSomething() {
  return 3;
}

void testfunc5(){
}
```

However this gives an error because `testfunc5` is defined after its usage in a class function. Thus in order to get around the reordering of the target code, we are specifying for the developer to define everything he uses before he uses it. For example the correct version of the python code would be:

```
def testfunc5():
        pass
class testClass:
        t = 2
        t_b = True
        def __init__(self):
                testfunc5()
                t = 3
                z = self.doSomething()
        def doSomething(self) -> int:
                return 3
```

Following, the correct C++ output program would be:

```
void testfunc5(){
}

class testClass {
   public:
     int t = 2;
     bool t_b = true;
     testClass();
     int doSomething();
};



testClass::testClass(){
  testfunc5();
  t = 3;
  int z = doSomething();
}
int testClass::doSomething() {
  return 3;
}
```

Running the program

To run the program, run `pythonCompiler.py` with a file. This runs the parser, then the symbol-table generator, then the typechecker, then the IR generator and prints the IR. For example using a file from `tests`, we can run this. Details for the flags used are in `pythonCompiler.py`.

```
python pythonCompiler.py -a tests/class_test
```

Changes in AST from sprint 1

We made multiple optimizations to our AST structure, as per the feedback we received from sprint 1. Our original AST from sprint 1 was overly verbose, incorrectly indented, and our grammar also caused some classifications of expressions. Consider this snippet of python code below:

```
if x:
    return 2
```

This yielded the following AST in sprint 1:

```
 ======= AST START =======
statements:
        if_stmnt:
                comparison_expressions_unary:
                        id_: x
                statements:
                        return_statement:
                                comparison_expressions_unary:
                                        integer_number:
                                                int: 2
======= AST END =======
```

After correcting our grammar, correcting indentation levels to properly match the code, and adjusting our grammar we are able to generate the following AST:

```
 ======= AST START =======
program: program
        if_stmnt:
                id_: x
                return_statement:
                        int: 2
======= AST END =======
```

However, note that for if/elif statements with further elif/else statements, we keep the subsequent conditional block as the **third child** of its preceding conditional statement. This means that for if/elif statements like this:

```
 if True:
    pass
else:
    pass
```

The returning AST will be (note how the `else_stmnt` is the third child of the `if_stmnt`):

```
 ======= AST WITH TYPES =======
program: program
    if_stmnt:
        bool: True
        reserved_words_statement: pass
        else_stmnt:
            reserved_words_statement: pass
======= AST WITH TYPES =======
```

The reason we did this is because we needed this structure to correctly generate our IR for conditional statements.

Symbol table implementation

Symbol-table generation is currently completed in two passes through our AST. In our first pass, we only add functions and classes into our symbol-table, and enforce error checking on all non-class function calls. This means, if a non-class function is called before it is defined, we throw an error and stop compilation. We do not enforce this on class methods because Python allows method calls within a class before defining the method itself. In the second pass, we then check variable assignment expressions. We deduce the type of the variable depending on the expression. For arithmetic expressions, we check the types of the terms within the expressions. For function call expressions, we appeal to the symbol-table to find the type the function returns. We also check class methods within the second pass. We know by the second pass, that if a class method is defined then it should exist in the symbol-table of its classes scope. Therefore, we can now check that method calls are properly defined.

Typed-ast implementation

Uses the symbol table to attach types to the ast. This python code gives the following tree.

```
 x = True
z = 2
y = z + z
```

```
program: program
        variable_assignment:
        Type :bool
                id_: x
                bool: True
        variable_assignment:
        Type :int
                id_: z
                int: 2
        variable_assignment:
        Type :int
                id_: y
                arithmetic_expression_plus:
                        id_: z
                        id_: z
```

TypeChecking

We check every expression in the tree and make sure it is the correct type in its context. For example we check if the return types for a method match its return statement, this fails the type checker.

```
def x_function()->int:
        return 5.5
```

We also make sure every expression's type properly matches what it's supposed to be. For example this passes the type checker

```
def doSomething() -> int:
        return 2

z2 = doSomething() + 2
t = 4
z = False
r = 99.9*(t + z2*t - (t+2/(3*(z2+100-2))) + (doSomething()/100 - 2))
```

but changing `r` to use `z` instead of `z2` fails it. The type checker does this recursively for every node in the tree. Also assigning something of a different type to a variable also fails, this is handled in the symbol table step.

```
r = 99.9*(t + z2*t - (t+2/(3*(z+100-2))) + (doSomething()/100 - 2))
```

Another thing to note is when running the type checker, if it catches an error it will raise an exception and halt execution. Otherwise, if there is no error it will just appear like:

```
======= TypeChecking =======
======= TypeChecking =======
```

IR Generation

Our IR generation converts our AST into 3AC format. We used our own 3AC syntax when describing for loops and while loops:

```
while i < 5:
    some_body
    i = i + 1
```

converts to:

```
 _L1:
while !(i<5) goto _L2
some_body_in_3AC
_t1 := i + 1
i := _t1
goto _L1
_L2:
```

where we specifically use the "while" key word instead of the "if" key word to describe the while condition. This is done to know that this was originally a while loop in the source code, so we should translate it into a while loop in the target code. We do a similar conversion with for loops:

```
for a in range(3, 4, 1):
        some_body
```

converts to

```
 L1:
for !(a < 4) goto _L2
some_body_in_3AC
start, step: 3, 1
goto _L1
```

where we use the "for" key word instead of the "if" keyword to describe a condition branch, and at the end of the for loop body we have "start, step" keywords to define the start of the for loop variable (a in this case) and the step after each loop.

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 type checking and IR generation, their output section will be highlighted with text similar to: "==== TYPE CHECKING ====" and "==== IR GENERATION ====" respectively.

To verify type checking, see if there is any output in the type checking section. If there is no output, it means there were no errors in the input file. If there is an exception raised, it means that a type error was found. Verify that the error it raises is valid.

To verify IR generation, view the IR generation section and compare the input code with the corresponding 3AC output. Check with the IR Generation section in this document when verifying the output of while loops and for loops, which we use our own representation of.

Changes in grammar

Limitations

The following expressions are currently not supported: - Function calls in arithmetic expressions

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

- Arithmetic expressions involving strings. These will be handled by returning an error, since C++ does not support such operations.

```
x = "hello"*3
```

- for loops with a negative "step" parameter in the range function.

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

- We currently do not have comparison expression precedence working within IR generation.

```
x = a and (b or c)
```

converts to

```
x = a and b or c
```

Bugs

We have found multiple issues/bugs. These are being recorded in the Issues section of our repo.