Phase 3

In the last part of our project, after implementing the scanner and parser of a Cool compiler, we are going to implement the code generator (maybe the most challenging part!) regarding this language.

All in all, we will attach all parts that we have implemented in this term to create a complete Cool compiler. To be more Precise, you need to code a program that takes a file containing a Cool program (with ".cool" extension) as input, and produces an output file in the MIPS assembly language (a three-addressed instruction set similar to the one discussed in class) then You'll be able to run that file with SPIM. (a MIPS processor simulator).

Structure of a program

A complete explanation about the **Cool** structures was given in previous phases, so your program must be able to generate MIPS programs for all of those structures.

Please notice that your project will be evaluated by some sample input codes, which attaches great importance to the implementation of the mentioned.

Scoping

As you have seen in the previous phase, this language supports scoping, meaning that the declaration of variables with the same name in different scopes will mask each other. To go further in detail, every class has its own scope and the inner scopes are methods or Statement Section that was mentioned in phase 2.

So, for instance if you have a declaration like "let a: int" outside of your main method, you are allowed to use that variable in your main unless you declare a new "let a: int" in your main method. In that case, the declaration of "a" in the inner scope will mask the declaration of "a" in the outer scope.

Arrays and Strings

- After creating an array, the number of elements can no longer be altered.
- Indices which are out of range are distinguished at run time.
- The indices can only be integers, starting at 0.
- If you assign an array to another array, only the reference will be copied. it’s the same for Strings.
- In case of comparing two arrays, their references will be used.
- String values can be compared with == and != .

- The range creates an immutable array of integers. A complete range is in form of range(START,STOP ,STEP). STOP and STEP is optional. Range arguments can be expression.

Semantic Errors

In phase 2, your program should have reported syntax errors. In this phase, you should report semantic errors that consists of:

1. Checking the type of variables in assignments and expressions (it consists of both default type and user declared types).

```
let int a;
a = 2.5; // Semantic error
let string d;
d = “salam”;
print(d + a); // Semantic error
```

2. Type checking in input and return of methods.

```
func getInt() int { return 2.5; // Semantic error}
func doNothing(int [] arr) void {} func b() void { doNothing(2); // Semantic error }
```

3. Using a variable or calling a method that could not be accessed in the current scope.

```
class MyClass {
    let int myInt;
    func a() void {
        let int localIntOfA;
        myLocalInt = myInt;
    }
    func b() void {
        let int localIntOfB;
        localIntOfB = localIntOfA; // Semantic error
    }
}
```

```
class A {
        void incr() { a = a + 1; // Semantic error }
}
```

4. Errors about arrays:
   - Index of arrays should be integer, greater than zero and be in the range of array size
   - the length of array should be greater than zero and integer
   - the type in`new type[]`should be the same as the type in declarations of it.

Some notes

- Note that all programs must contain a “Main” class that contain “main” method to start running.
- Implicit casts comprise of int to double, double to int, int to boolean and boolean to int.
- The code of casting the value of the Expr should be automatically generated when the code of the assignment is being generated.
- Method calls are treated as call by value which means that any change to a method parameter won’t affect its value outside of that method unless its type is array or string or be an instance of a class which case its value will be affected.
- Bitwise operators are used only with integer operands.
- Recursive functions are supported.

Grading Policy

We will provide some Cool codes which will try to cover all aspects of the language. Initially there would be some test cases which comprise of errors, and your compiler should output the message ''Not Compiled!''.

Otherwise (if the program doesn't have any errors) your compiler must pass the tests by creating a file containing the appropriate MIPS instructions. In order to get the complete score of this phase, your compiler must be able to handle all of the following parts. (The red ones will count as bonus points.)

Feature	Relative points
Reading from the console or an input file (Reading integers and strings)	**6**
Writing in the console or an output file	**6**
Integer assignment and computations	**8**
Real assignment and computations	**7**
Boolean assignment and computations	**5**
Type casting (just explicit casts of real to int and int to real)	**2**
Handling other type of expressions (for example bitwise expressions)	**5**
Handling Strings (printing and proper storing of strings)	**6**
Checking the type of variables in assignments and expressions (Semantic Error)	**3**
Type checking in input and return of methods (Semantic Error)	**2**
Using a variable or calling a method that could not be accessed in the current scope (Semantic Error)	**2**
Errors about arrays (Semantic Error)	**3**

1D Arrays	**8**
if-else	**8**
Loops	**9**
Methods	**15**
Classes	**10**
Len() method (for strings and arrays)	**2**
String '+' operator	**3**
Warning for unreachable code	**6**
Cascade assignment (like a = b = 2)	**4**
Proper Error declaration (indicating the line of error and a proper message)	**5**
++ and -- after Expression (++ and -- before expression is not a bonus)	**2**
optimization (basic block)	**4**
optimization (constant folding and constant propagation)	**5**
optimization (loop unrolling)	**3**
***Any other innovations**	**Depends!**

Some notes about grading policy

- consider that only the implicit casts count as bonus not the explicit ones.

  Example:

  ```
  let int a;
  let real b;
  b = a + 2;    // implicit cast
  b = (real) a;  // explicit cast
  ```

- By default, you should declare the variables before using them but in a class like below we can use it before declaration, handling this condition also counts as a bonus point.

Example:

```
class MyClass {
        func incr() void { count = count + 1; }
        let int count;
}
```

General Procedure

In order for you to have an overall insight, the following steps are advised to you for implementing your compiler:

1. Adding semantic tokens to edges of your diagrams in PGen (your diagrams may need some modification).
2. Obtaining the parse table from PGen.
3. Creating a program that connects your scanner and parse table (based on the discussions in the class) and indicates if a program has errors or can be compiled.
4. Writing the code for each semantic token and making the assembly file (with ".s" extension) and add commands to it.
5. Testing your assembly file with SPIM.

How to connect phases

After generating full parser with "Export Full Parser" option in PGen, the generated files contain Lexical.java, Parser.java and CodeGenerator.java.

Your Scanner in phase 1 must implements generated Lexical.java interface and your CodeGenerator in phase 3 must implements generated CodeGenerator.java interface.

Finally, you should call "parse" method of Parser.

SPIM

As mentioned before, your output files should be in the form of a **MIPS program**. Then your file can be executed using SPIM, which is a MIPS processor simulator. By installing it on a linux distribution and running the following command, you can execute your file:

```
spim -a -f b.s <c.in> d.out
```

To be more precise, a **Cool** program like code.cool will be given as input to your compiler, expecting it to raise an error or producing a MIPS file like b.s. In case of the latter, you can execute b.s with inputs in a file like c.in and save the results in a file like d.out.

Note that SPIM utilizes an instruction set which may Facilitate writing MIPS programs, so be sure to use the instruction sets mentioned in the SPIM documentations, and not raw MIPS instructions. Also, you can use QTSpim and load your assembly file to QTSpim and run it.

An example of assembly SPIM code:
https://gist.github.com/hamidhandid/fe9c8bb20a48794312a47d00ddf59cbd

How to test

If your program has any errors (syntax error or semantic error), "Not Compiled!" should be printed in the console, else you should generate code in your assembly file. Your generated assembly file will be tested through the script that will be given to you. Make sure that script can run with your program.

Final Notes

- The due date is Tir 7th.
- Your generated assembly file should run properly for tests that will be given to you.
- You should upload your project and a report that explains how to run your project.
- This part of the project can be done in groups of two.