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 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 recent 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. Conspicuously your failure in their implementation results in a significant loss of points.

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.

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 != (considering case sensitivity).

Some notes

- Note that all programs must contain a “Main” class that contain “main” method. Infringement of this law must lead to a compile error.
- 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 and double operators.
- Recursive functions are supported.
- A static variable can be access without creating any instance of a class. (Note that implementation of static variable has a bonus point)

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	6	in_int in_string
Writing in the console or an output file	6	out_int out_string
Integer, boolean and double assignment and computations (based on the expressions mentioned in the previous phase)	20	
Type casting	5	int-real
Handling other type of expressions	7	
Handling Strings	5	

1D Arrays	6	let a: real[];
		arr <- new Array(real, 5);
		len()

if-else	8	

Loops	7	for
		while

Methods	12	
Classes	8	
Static Variable for Classes (with static prefix)	3	
*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)	4	
++ and -- after Expression (++ and -- before expression is not a bonus)	3	
*Any other innovations	Depends!	

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 procedures to 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 rule and making the assembly file and add commands to it.
5. Testing your output files 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 Lexical.java and your CodeGenerator in phase 3 must implements CodeGenerator.java.

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

An example of connecting phases:

https://github.com/hamidhandid/compiler_connecting_phases_example

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 a.d 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.

The documents for spim will be attached to this one.

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

Notes

- The due date is Tir 10th.
- You should upload your project and explain how to run it. (Report is not necessary. Just an explanation of how to run the written compiler is enough.)
- This part of the project can be done in groups of two.