Assessed Exercise 2

Introduction. Here is the second (of two) assessed exercises. It consists of three tasks.

- **Task 1** (worth 40%) is here.
- **Task 2** (worth 45%) is here.
- **Task 3** (worth 15%) is here.

When and how to submit. The first assessed exercise is due Friday 15 December 2017, at 18:00. To assure anonymity, please submit via the course's Study Direct page, where I will set up a submission link. For submission guidelines see here.

Your task is to implement, in Java, code generators. As source language we take subsets of (a minor variant of) the simple language that we generated code for in the lectures (see the lecture notes, Page 4). As usual, the language is represented by ASTs. The target language is MIPS machine code.

Background. It might be interesting to see the language you will have to compile in it's conventional (non-AST) form. Here it is as a context free grammar.

```
PROG → DEC | DEC; PROG
DEC → def ID (VARDEC) = E
VARDEC →  ε | VARDECNE
VARDECNE → ID | VARDECNE, ID
ID → ... (identifiers)
INT → ... (Integers)
E →  INT
   | ID
   | if E COMP E then E else E endif
   | (E BINOP E)
   | (E)
   | skip
   | (E; E)
   | while E COMP E do E endwhile
   | repeat E until E COMP E endrepeat
   | ID := E
   | ID(ARGS)
   | break
   | continue
ARGS → ε | ARGSNE
ARGSNE → E | ARGSNE, E
COMP → == | < | > | <= | >=
BINOP → + | - | * | /
```


The ASTs for this language can be found here. Do note change these definitions at all! (except by adding new methods). If you change them in a way that my test suite will not compile, you will get 0 points.

An important difference from the pseudo-code used in the lectures is that in declarations we use numbers as variables (instead of strings). Variable 1 refers to the first (leftmost) variable in the declaration, variable 2 to the second from the left and so on. For example the procedure

```
def f(x,y,z) = { if x == y then z else 0 }
```

becomes

```
Declaration ( "f",
              3,
              new If ( new Variable ( 1 ),
                       new Equals (),
                       new Variable( 2 ),
                       new Variable( 3 ),
```

```
                                    new IntLiteral ( 0 ) ) );
```

The meaning of `break` and `continue` will be explained in Task 3. Note that you don't have to write a lexer or parser for this language, only a code generator. All your code generators will use the following given classes.

Notes on the implementations. It is your task to implement the code generators in Java. You will have to use the following interface and exception to implement the code generators.

```
class CodegenException extends Exception {
    public String msg;
    public CodegenException ( String _msg ) { msg = _msg; } }

interface Codegen {
    public String codegen ( Program p ) throws CodegenException;
}
```

The MIPS code to be produced is returned by all these methods as a string. Don't print out the generated code. Just return it as a string.

You can download these here. Do note change these definitions at all! If you change them, my test suite will not compile and you will get 0 points. I will instantiate your code generators with the following classes, one for each task.

```
  class Task1 {
     public static Codegen create () throws CodegenException
{ ... } }
  class Task2 {
     public static Codegen create () throws CodegenException
{ ... } }

  class Task3 {
     public static Codegen create () throws CodegenException
{ ... } }
```


(You find these three code fragments here, here and here.) To implement e.g. Task 1, replace the `...` with appropriate code that instantiates the `Codegen` interface. If you don't want to do a task, just throw a `CodegenException` exceptions in `create` instead.

The `codegen` method is intended to generate code for a whole `p` of type `Program`. The code generated for a whole program p includes preamble, data layout directives etc. The generated code should be valid and executable MIPS code. You need to add alignment instructions and ensure that upon start of your program control is handed to the initial program, and upon termination control is handed back.

The code generators for Task 1 and Task 2 work only on subsets of the language. The test suite I will use for Task 1 will only use ASTs that are valid programs in the subset of Task 1. The test suite I will use for Task 2 will only use ASTs that are valid programs in the subset of Task 2. So it doesn't matter what your code generators for these tasks do when they encounter ASTs that are not valid in the relevant subsets. For example if you do Task 1 and 2, it makes sense to write a code generator for Task 2, and then let `Task1.create` as well as `Task2.create` return the same code generator.

The generated code must do the following.

- Be syntactically correct, meaning is must compiler.
- Generate syntactically correct code, meaning generated code must run in the MARS emulator.
- Implement the accumulator machine described in the lectures. You should use `$a0` as accumulator. The final result of the program should also be in `$a0`. If you don't want to implement an accumulator machine, then please contact me beforehand and discuss this matter with me.
- Implement the meaning of the program correctly, i.e. 4+4 must evaluate to 8 and so on.
- Should not lead to an error when being executed, e.g. due to memory misalignment.
- Integers should be mapped to MIPS 32 bit integers, as described in the lectures. Note that

you don't have to care about integer overflow. All integers occurring in test-data will be sufficiently small to avoid overflow.

You can assume that each program that I use to test your submission definition defines at least one procedure. You can also assume that all declarations in a program have distinct names. All programs are valid, e.g. contain no type errors. Moreover, the first declaration in the declaration list of a program takes 0 arguments. This first declaration is to be executed first, i.e. it is the procedure that should be executed when control is handed over to the code your code generator produces. Note that the first procedure does **not** have to be called `main`.

Note that the test suite will call the `codegen` multiple times on the same object of type `Codegen`.

Use the provided `CodegenException` to signal errors or problems in your code generator. Do not use other means of signalling errors. In particular, please don't abort the program if you encounter a problem.

How to avoid common mistakes. Here is a list of suggestions that help you avoid common mistakes, all of which I've seen in plenty of submission last year:

- You'll need a function that generates fresh labels. Don't hardcode labels, ensure that the function generates a fresh label every time it is called. Don't use a random generator for this, use a global/static variable for this purpose. This is one of the few legitimate uses of global/static variables.
- If your sumission for the first assessed CW suffered from problems related to using instance variables where you should have used method-specific variables, ensure that you don't make this mistake again.
- The MIPS assumbly your submission needs to generate a string, which each new assembly command being on a new line. It's easy to forget a "\n" here or there, if you add them 'by hand'. Hint: don't put them in by hand. Have your code generator generate an auxiliary list (or similar) of MIPS commands and then transform this into a string where the "\n" are automatically added.
- Check that your submission is not miscompiling conditionals like `if/then/else` or loops in the sense that if the condition evaluates to true then the else-branch is executed and vice versa.
- Repeated implementations of the translation of conditionals: The source language has several places where conditionals need to be translated: `if`, `while`, and `repeat`. I recommend not to repeat the code generator for conditional three times, but have an auxiliary function generating the code, and calling that auxiliary method for `if`, `while` and `repeat`. Note that the conditional for repeat is the 'other way round' in comparison with `while` so a joint translation method would a way of handling the two different ways of translating conditionals.
- I suggest that you split your code generator into at least the following three methods: `codegenProg`, `codegenDecl` and `codegenExp` that translagte programs, declarations and expressions, respectively. You are welcome to have more (e.g. for comparison operations and arithmetic expressions).
- I suggest to ensure that your submission isn't 'noisy', meaning what you submit does not emit debug output: the problem with debug output is that it might be interpreted as MIPS assembly which leads to errors.
- I strongly recommend testing your code before submission. Note that cutting/pasting code into MARS is insufficient as test, because that doesn't deal with the termination.

A small test suite that you can use will be made available here a little later. Since I will evaluate your submissions using a testing framework like the above, it is vital that you keep the relevant class signatures intact.