COMP 321: Design of programming languages
Homework 7: Compliation for a fragment of C

---

1. Written problems

There are no written problems for this assignment.

2. Coding problems (20 points)

For this assignment you will write a compiler for the fragment of C for which you have already written a type-checker and interpreter. The language is as described in Homework 6 except as noted below. There are a number of things to keep in mind.

- I suggest that when you first write the compiler, hard-code the stack and local storage space for each function to 1000. Once you've completed everything, go back and see if you can optimize this. To do so, you may have to first write to a temporary file, then read back in, and then write the final result to `outsream` argument of `compile`.
- You may make use of `ldc` etc., instead of optimizing these instructions.
- Do not make the environment a global mutable value as suggsted by the text; instead, your functions should return an updated environment.
- I will only test your code on programs that can be interpreted without errors by the interpreter supplied with the testing framework. This is the same language as in HW 6, except as noted in the next point.
- There is one change to the language in this assignment, which fixes a bug in previous versions. Previously, the only valid form of `return` statements was `return` e, where e is an expression. The language for this assignment permits `return` statements without any expression, for use in returning from `void` functions. Furthermore, every `void` function must execute a `return` statement, although (as for non-`void` functions), failure to do so will not be caught by the type-checker.
- The class to which you compile your code must be named `C`. All functions must be static functions of class `C`.
- We don't know how to handle strings, so don't worry about them.
- Floating-point computation for Java and SML is different, presumably because Java is 64-bit and SML is 32-bit. That can cause annoyances when using the testing framework to check your files because, e.g., you might be expecting 3.2, but when the JVM executes your compiled code, it produces 3.200000047683716. Some numbers I have found to be "safe" are integral numbers (1.0, 3.0, etc.); 1.5; 3.5; and 8.5.

The compiler. You must implement a structure named `Compile` that implements a function `compile` : `AnnAst.program*TextIO.outstream -> unit`. `compile`(p, $outs$) will compile the program p, writing its output to the output stream $outs$.

The main functions that you will likely use to write lines to the output stream is `TextIO.output` : `TextIO.outstream*string -> unit`; `TextIO.output`($outs$, s) writes the string s to $outs$. Remember that `TextIO.output` does *not* add newlines! You will probably want to invoke `TextIO.flushOut` on the output stream every time you call `TextIO.output` so as to ensure that the line is immediately written to disk. If you do not do so and your program crashes, then it is likely that not all text that you have invoked `output` on is actually written to disk. That can make it very difficult to debug your code, because it will look like you have called `output` fewer times than you actually did. Most likely you will want to write a number of short functions for printing lines to the output stream.

The test programs invoke functions such as `readXXX` and `printXXX` where `XXX` is `Int`, `Double`, or `Bool`. These functions are defined in in the `CSupport` Java class, for which the source code and bytecode are provided in the testing framework. You will need a bit of special-purpose code in your compilation scheme for `ECall` expressions, because these I/O functions are in the `CSupport` class, whereas all other functions should be assumed to be in the `C` class.

3. Testing

The driver program has been extended to include a `compile` command which will invoke `Compile.compile` to compile the C source code in the named file to Jasmin assembly. The assembly code will be in a file of the same name as the C source code file, but with `.cc` replaced with `.j`. This is a plain text file, and so you can examine it with any text editor.

You may compile the Jasmin assembly to Java bytecode, suitable for interpretation by the `java` interpreter, by running Jasmin. Jasmin is available from `http://jasmin.sourceforge.net`, and there is also a copy of it in the `testfiles/good` subdirectory of the testing framework. To compile `f.j` to a class file `C.class` (where `C` is the class that is defined in `f.j`[1]), execute

```
$ java -jar jasmin.jar f.j
```

You can then execute the compiled program with

```
$ java C
```

Note that if the original source code invoked `readInt`, etc., then `CSupport.class` must be in your classpath; the easiest way to arrange this is to ensure it is in the same directory as `C.class`.

As usual, I have provided a testing framework for you with some sample files. All files parse, type-check, and can be successfully interpreted by the interpreter supplied with the testing framework (as noted above, the language, and hence the type-checker and interpreter, are slightly different than those for HW 6). The testing framework will compile each `.cc` file in `testfiles/good` using `Compile.compile`, invoke Jasmin to assemble the result, and then execute the resulting class file with `java`. You must have `jasmin.jar` in the `code` directory and `CSupport.class` in `testfiles/good`. You must also have `java` installed on your system and in your `PATH` in order to execute the testing framework (this is the case for the VirtualBox virtual machine distributed for this course). Just as with HW 6, input to each program (for `readXXX`) is supplied by a corresponding `.input` file, output (from `printXXX`) is written to the corresponding `.results` file, and the `.results` file is compared to the `.output` file to check for correct output. The testing framework can report three kinds of errors for each file: `compiles` errors are caused by exceptions being raised by `Compile.compile`; `assembles` errors are errors that occur during the execution of Jasmin (the latter would typically be caused by syntax errors in the assembly your compiler emits); and `executes correctly` errors are errors in the output of a program that compiles and assembles.

4. Submission

You will submit only `compile.sml` (your interpreter implementation).

[1] Unlike compiling Java source code with `javac`, there is no requirement that the filename match the class that is defined in the file.