PROGRAMMING LANGUAGES AND COMPUTATION

Problem Sheet 8: Verifying and Compiling While Programs

We first reason about given While programs, and prove some simple properties.
We then compile While programs down to the Abstract Machine, and beyond.

Reasoning about While Programs

Consider the While program example from Week 1.

```
x := x * x - 1
while !(x = 0) {
  x := x - 2
  y := 2 * y - 1
}
```

* 1.

(a) Specialize the semantics of While to consider only configurations reachable from initial configurations whose statement component is the While program example.

(b) Compare the resulting set of transitions to the transition system described in the original example. (In such a situation, we say that the transition systems *simulate* each other.)

*** 2. (Optional.) TBD

Compiling While Programs

** 3. Consider the While programs from the Week 5 worksheet (also used in Week 6). Manually compile them to the Abstract Machine language.
You can test them with the `bam` program provided in the labcode archive. Don't forget to initialize the variables meant to serve as inputs before running tests.

At the moment, `bam` reads its input from `stdin` and prints any output to `stdout`. In order to invoke it, run:

```
> cat <filepath> | cabal run bam
```

or

```
> echo "program" | cabal run bam
```

You can combine `echo` and `cat` to prepend any state initialisation required to pass arguments. For example

```
> echo "PUSH 12; STORE n;" | cat - bam/test/factn.am | cabal run bam
```

will run the factorial program in a state with variable `n` initialised to 12.

Making Haskell Compile While Programs

In this section, we compile While programs down to the abstract machine's language. (Or rather, we make Haskell do it for us.) Our compiler will translate While ASTs to AM ASTs in a syntax-directed way.

We do so in file while/src/WhileC.hs.

We first write a function to compile arithmetic expressions (type `AExpr`) down to Abstract Machine code (type `Code`).

An Abstract Machine program is just a list of instructions (although some of them currently have more structure), so we can construct them mostly using operations over type `List`. However, those are most efficient when constructing lists from right to left. (Another story of left recursion vs right recursion.)

To allow us to keep thinking about the compilation problem from left to right without losing the (very minor) performance gains of constructing the list from right to left, we write our translator using an accumulator. (In essence, we construct the AM program on our way back up from the recursive calls instead of constructing its beginning, then recursing down to construct the end.)

As an example, you may have already seen how the following naive implementation of list reversal

```
rev :: [a] → [a]
rev []     = []
rev (x : xs) = (rev xs) ++ x
```

can be implemented much more efficiently as

```
rev :: [a] → [a]
rev = aux []
 where
   aux acc []     = acc
   aux acc (x : xs) = aux (x : acc) xs
```

** 4.

(a) Write function `arithToInstrs :: AExpr → Code → Code`.
Its behaviour when its second argument is `[]` should be as specified in the lecture (and the reference material).
We actually want to keep this as its externally-visible type because we'll also accumulate

(instead of concatenating) when we call it in the midst of compiling boolean expressions.

(b) Write function `boolToInstrs :: BExpr → Code → Code`.
Its behaviour when its second argument is `[]` should be as specified in the lecture (and the reference material).

(c) Write function `stmtToInstrs :: Stmt → Code → Code`.
Its behaviour when its second argument is `[]` should be as specified in the lecture (and the reference material).