Assignment 5
Translation
Due: 11:59pm, Wednesday October 24

E-mailed questions about this assignment should be sent to cs132help@cs.hmc.edu.

1. Make a copy of your Assignment 4 solution and `svn cp` the files from `src/a5`.
   Please do not change the new files without consulting with the professor first.

2. Read the following four signatures:
   - `typeutil.sig`: the interface for the structure `TypeUtil` in `typeutil.sml`. A few new functions have been added, and the types of others have been changed (see below).
   - `temp.sig`: the interface for the structure `Temp` in `temp.sml`. This module implements a representation of temporaries (like general purpose registers in a CPU, but there are as many as we need) and labels (as in assembly code).
   - `target.sig`: the interface for the structure `Target` in `target.sml`. This is an abstract, machine-independent low-level representation.
   - `machine.sig`: the interface for the structure `X86` in `x86.sml`. This is an abstract interface to a module with provides all the machine-dependent information. You will not need all this code yet. Focus on the `access` and `procinfo` types and to fragments.

   You can ignore the rest of the signature for now, and the other files.

3. Mondify your `typecheck.sml` file to define
   
   ```
   xprog : abstract_syntax_for_a_program -> Tree.fragment list
   ```
   
   Although this looks like you might recursively walk over the code and return lists of fragments to be consed/appended, you should probably create fragments imperatively (storing them in a global table by calling `Target.addFragment`), and then at the very end have `xprog` call `getFragments` to get them all back.

   This may seem a little round-about, but it's convenient when you're in the middle of a piece of code and, for example, discover that you need to create a fragment to hold a string constant.
There are two interface changes in TypeUtil that may break your typechecking code. You may want to start by getting your typechecker to compile again.

(a) The translation context now stores a value of type X86.access for each local variable. You have to supply an access whenever you add a variable to the context. These can be obtained either from X86.formals applied to the function’s procinfo (in the case of formal parameters) or from X86.allocLocal (in the case of local variables). To generate code to access a variable, just call X86.accessToExp on the associated access value.

(b) getConstructor now returns a constructorInfo record containing the formal parameter types and the constructor’s procinfo, rather than just returning the list of formal parameter types. Similarly, getMethod now additionally returns the procinfo for that method body.

• Each method body and constructor body now has a value of type X86.procinfo (created by X86.newprocinfo). The TypeUtil code generally takes care of creating these.

• Your generated code may need lots of new “variables”. You can create new temporaries as needed by Places for storing temporary values can be be stored in a temporary created by calling Temp.newtemp().

• The translation context also stores the current procinfo. You can get the current frame by TypeUtil.currentProcInfo, but you will have to add calls to TypeUtil.enterCode in your type checker right before you start type checking the body of a method or a constructor.

• The type checking code is complicated enough that you may want to do the translation for each case using a helper function.

• You are likely to want to make calls to helper functions written in C (e.g., to do string append, convert integers to strings, allocate and initialize memory on the heap, or print strings). Create a file runtime.h that has declarations for any such functions; be sure to add comments to explain what these functions are supposed to do. Don’t forget to add this header file to the repository.

• TypeUtil uses plausible label names for code fragments (e.g., Class__methodname for the code of a method body and Class_vtbl for the virtual method table). Of course some fragments (such as those storing string constants) have no natural name and hence should use “fresh” labels generated at compile-time.

• Do not try to pack strings or character arrays; just use arrays of integers.

• The translation of a return is to jump to the current procinfo’s return label. (If you want to return an expression, assign the value to the RV register before the jump.)
• We discussed translating booleans as control-flow (code that jumps to one of two labels) vs. booleans as values (e.g. 0 vs. 1).

If you choose to use the control-flow approach, you may want to write a separate function

\[
xbool : TU.translationCtx -> Absyn.expr -> label * label -> Target.stm
\]

to handle only boolean expressions, and then call this when translating *if* and *for* statements.