Important: You may not work on this assignment (or even seriously think about it) between 8am on Sunday, March 13 and 8am on Sunday, March 20.

1. Make a copy of your Assignment 4 solution and `svn cp` the files in from `src/a5`.

2. Look through `Target.hs` and `X86.hs`.

3. Add

   ```
   import qualified Target as T
   import qualified X86
   ```

   with the other imports in your `Translate.hs` file.

4. There are a few interface changes in `Ctx` that will break your type checking code. Start work by getting your type checker to compile again (without doing much real translation yet).

   Here are some changes that you should be aware of:

   (a) `Ctx.hs` now assumes your type checker/translator is specifically using the `IO` monad. If you uncommented any type specifications that claimed your type checker works with `any` `Monad ((Monad m) => . . .)`, you'll need to delete or change these.

   (b) The translation context now stores a value of type `X86.Access` for each local variable, which is used to find where that memory lives (register or memory). You have to supply this information whenever you add a local variable to the context.

      • For variables declared within a function, you can create a new `Target`-level variable by running the (monadic) operation `C.freshTemp`, and then use `X86.tempToAccess`.

      • For variables that are formal parameters to a function, `X86.formalParameterLocations` tells you where you can find each parameter.

   Because of this change, the types of `addLocalVar` and `findLocalVar` have changed.

   (c) `Ctx.makeCtx` is now monadic, i.e., must be invoked within a `do` block with `<-`. Assuming your code to add all the classes and methods to the context (`BuildContext`) is not written in a monadic style, you may need to refactor your code so that `Ctx.makeCtx` is called from `translate`, and then this mostly-empty context gets passed to `BuildContext`.

5. The next step is to actually produce code. In the end, your `checkProg` should return a list of type `[T.Fragment]`, representing the various pieces of data and code in your program (e.g., one `T.FragCode` for each method, one `T.FragLabels` for each virtual-method table, and one `T.FragGlobal` for each static field.)
Nevertheless, try to **work incrementally**. For example, you might start by having ` synthExp ` return a pair containing both a type and a translation ` Target.Exp `, but then modify the rest of your code to ignore this extra return value and make sure the code still compiles/runs. You can even do this even more incrementally, by actually doing the translation for just a few cases, and returning the value ` undefined ` in all other cases.

Here are some more things to be aware of:

- To actually use a local variable or function parameter, get its ` X86.Access ` value (e.g., from the typing context) and then pass it to ` X86.accessToExp `.
- Your generated code may need lots of new ` Target ` level temporary variables. In monadic code you can create new temporaries (of type ` Target.Temp `) as needed via
  ```
  tmp <- C.freshTemp ctx
  ```
  where ` ctx ` is the typing context.
  Similarly you can get a fresh label via
  ```
  lab <- C.freshLabel ctx
  ```
  where ` ctx ` is the typing context.
- You are likely to want to generate 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). Generate ` T.CALL ` operations to functions with reasonable names, and then create a file ` runtime.h ` declares these functions. In your header file, 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.
- You will need to have your own name-mangling convention for code fragments (e.g., ` Class__methodname `).
- The translation of a ` return `; is to jump to the current function's return label. (Return values are put into the ` X86.rv ` return-value register before the jump.) The function ` Target.makeReturnLabel ` applied to the (mangled) label for the code returns a plausible label. However, you should not actually put this label at the end of the ` Target ` code you generate; it will be added later.
- One minor optimization (that can make parts of the translation easier) is a specialize translation of boolean expressions for tests in conditionals and loops. The “naive” translation will lead to code that does conditional branches to construct a zero/one value, and then immediately does more conditional branches to test whether we just constructed a zero or one.
  A simple alternative is a separate function
  ```
  ```
  that type checks and translate expressions that we know are supposed to return booleans. Given an expression and two labels, it should generate a statement that jumps to the first or second label, depending on whether the expression is true or false. (Or report an error, if the code doesn’t type check)