CS 132: Compiler Design, Spring 2011

Assignment 4: Type Checking BC#
Due: Wednesday, March 2, 11:59pm

1. Run `svn update` in `cs132s11` to make sure you have the latest files.

2. Make a `svn copy` (not just a `cp` copy!) of your `a3` directory to make `a4`.

3. Use `svn copy` to add the additional files in `src/a4`. The new files include
   - `Ctx.hs`, containing a bunch of helper code related to types and type checking, especially tracking the types of free variables and the information about classes.
   - `Translate.hs`, containing a rather bare skeleton of code to type check the BC# abstract syntax and translate it into a lower-level representation.
   - `BCSC.hs`, containing a small extension of ParseTest.

4. Extend the code in `Translate.hs` to implement a working type checker. (Later we will extend your code to additionally translate to a lower-level representation, but you do not need to worry about this now.)

The type checker will have two parts:

   - First, you need to create a typing context with information about all the classes. This will proceed by starting with `Ctx.makeCtx`, and then walking over each class in the program, calling `Ctx.recordSuperclass` for the class and then the other `record` functions for the class components. In order to do so, you will need to write a function that sanity-checks the abstract representation of types, and translates them into the `Ctx.type` “internal” representation used by the typing context.

   - Once you have the typing context all set up, the second step is to go through all the code (fields, methods, expressions, statements, etc.) and check that the types are all consistent. To keep Haskell’s laziness from throwing away checking operations, this part of the code should be written in monadic style, with do blocks. (You shouldn’t rely on using any particular monad. However, you can temporarily assume the monad is Haskell’s IO monad, allowing you to add calls to `putStrLn` in your code while debugging.)
5. For each individual student (even if you’re working in a pair), add at least three non-trivial test inputs in cs132s11/tests/goodtype that should parse and typecheck, and at least three test inputs in cs132s11/tests/badtype that should parse but not typecheck. Name these files using your HMC CS username and a strictly positive integer, e.g., stone1.bcs, stone2.bcs, .... (Please start counting from 1 in each directory.)

(Don’t forget the svn add and svn commit commands here.)

Changes to BC#

- The Object class seemed pretty boring. So, let’s say that there’s a virtual method toString() that always returns the string "object" (but which can be overridden with a more interesting string representation in any user-defined class).

- After thinking about the discussion of null in the last class, I decided that null can be used in a context expecting a string or array or object.

Hints

- You do not need to study the implementations of functions in Ctx.hs, but you will want to use most of them to implement your type checker.

- The code in Ctx.hs makes use of records. These are analogous to C structs, or tuples with named components. Two tricky bits:
  
  - Instead of saying myRecord.fooComponent to access the fooComponent component of myRecord, Haskell makes you use prefix representation

  fooComponent myRecord

  That is, the names of the components act like extract-the-particular-component functions.

  - As for all functions defined in the Ctx module, when you are outside the module the names of components will be prefixed by the module name. For example, if the variable finfo in your type checker has the record type Ctx.FieldInfo, then to access the type-of-this-field component you would say Ctx.fieldTy finfo rather than simply fieldTy finfo (or finfo.fieldTy as one would say in C/C++/Java).