Discussion questions for *Template Meta-programming for Haskell*
Scribes: Jayme Woogerud, Sophie Panuthos, Matt Ahrens

1. Compile-time meta-programming is programming in code that gets executed at compile time (versus run-time) and produces code (in abstract syntax form), which is then spliced into the abstract syntax tree just as if it were typed by the programmer.

2. Compile-time meta-programming can be used for:
   a. Variatic functions
   b. Arbitrary arity functions (e.g. `zipN`)
   c. Custom “deriving” clauses
   d. Conditional compilation
   e. Program-specific optimizations

3. A splice is notated by a ‘$‘ or the keyword `splice`.
   a. At compile time when the $ or splice is encountered, the code to be spliced in is evaluated to produce a haskell AST which is to be combined with the AST of the existing compiled haskell (host language)’s AST as if the code was written completely initially.

4. Code enclosed in `[ | … | ]` is quasi-quoted Haskell code, which is executed at compile time and generates Template Haskell code (`Expr` and `Patt`), which represent Haskell abstract syntax. In many cases, it is easier to write in Haskell and let the compiler generate the abstract syntax if possible, rather than write the abstract syntax directly.

5. The `gen` function recursed through a list of formats and accumulates a formatted string. To extend the function to support floats, we would add a new constructor to the `Format` type and and a new case for `F (float)`, which would be identical to the case for `D`.

6. In the `sel` function, the pattern for case expression depends on the value of `n`, which we cannot write in haskell notation, since haskell does not support case statements of unknown / arbitrary size.

7. “Reification” is a method for querying and examining the compiler representation of Haskell data structures directly. This is useful for optimization, conditional compilation, and allowing the programmer to code against partially compiled code.

8. The function `cross2b` may inadvertently capture the variables ‘x’ and ‘y’, which never get renamed with fresh type variables. This doesn't happen in `cross2a` because the Haskell compiler takes care of the variable renaming in the quoted code.

9. The quotation monad provides:
   a. The usual `return` and `>>=`
   b. `gensym :: String -> Q String` (generating fresh names for variables)
   c. Failure messaging
   d. `IO -> Q`
   e. `reify`

10. Template Haskell data types to represent `apply (f, x) = f x`:
    ```haskell
    lam [ptup [pvar “f”, pvar “x”]] (app (var “f”) (var “x”))
    ```
11. The language designers chose to not permit splices and quotations in generated code so that the generated code can always be statically type-checked at compile time. Generated code is executed at run-time -- therefore, code generated by these splices and quotations could only be type-checked at run-time.

12. The code $f \ x = \$ (\texttt{zipN} \ x)$ is illegal at top-level because the value of $x$ is not known at compile time, therefore $\texttt{zipN} \ x$ cannot be evaluated.

13. In the state diagram, ‘C’ represents the normal Haskell compilation stage, ‘B’ represents compilation of quasi-quoted code, and ‘S’ represents splice compilation. The levels represent how deeply nested compilation is into brackets and splices.

14. Splices are evaluated top to bottom so that scoping is intuitive and consistent. Otherwise, it would be impossible to know whether or not declarations are in scope or not.

15. Declaration splices are restricted to top-level because they may bring any number of names into scope -- if these declarations were not at top-level it would very difficult for the programmer to see where variables are bound.

16. It runs arbitrary code at compile time that has access to the full set of IO operations.

17. It allows us to refer to declarations that may be out of scope, i.e. in other modules.

18. Pro/con to having language execute at compile time
   a. Pro
      i. Don’t have to learn a new language
      ii. Haskell is well-suited for writing compilers
      iii. The code is all in one place
   b. Con
      i. Difficult to see boundaries
      ii. Compile time language is too complex. It doesn’t need to be Turing complete

19. 3 Layers of TH
   a. Quasi-quoting
      i. Easiest to read/write
   b. Q monad
   c. Data structures
      i. Just for lifting to Q monad (don’t use generally)
      ii. Most powerful