Homework 3: An Interpreter for Mini-OCaml (MOCaml)

Due Monday, May 8, at 11:30pm

Turn in your homework via the course web page as an updated version of the `interp.ml` text file that I have provided in the attached zip archive. Do not rename the file, and do not turn in (or modify) any other files.

Make sure your code can be successfully loaded into the OCaml interpreter; if not you get an automatic 0 for the homework!

Recall the CS131 Academic Honesty Policy! You must say whom you discussed the assignment with at the top of your assignment, and also what other resources you used.

For this assignment, you will implement an interpreter for a subset of OCaml in OCaml. This is a fun way to get more practice with OCaml and also to really learn how the various language constructs behave. You should obey our usual style rules and tips from past homeworks, of course (e.g., no use of any imperative features of OCaml).

Now on to the assignment! First `unzip` the `hw3` directory from the zip archive I've provided. The directory contains an incomplete interpreter for a subset of OCaml, which we'll call Mini-OCaml, or MOCaml. Your job is to complete the interpreter by implementing the three functions declared in the file `interp.ml`. You may also declare any helper functions that you find useful in that file as well, and you may use functions from the OCaml standard library. Your MOCaml interpreter should behave just like the regular `ocaml` interpreter for the subset of OCaml that we are handling. So if you ever want to figure out what should happen when a particular expression is evaluated, just try it in `ocaml` and find out! However, there are a few important differences:

- MOCaml is *dynamically typed*: all typechecking is performed at run time. So your interpreter will happily execute any syntactically legal program (but may signal errors during execution), while OCaml will reject some programs during static typechecking. For this reason, MOCaml allows some programs that OCaml rejects, for example `if true then 0 else false`. Like OCaml, MOCaml supports user-defined data structures like `Node(Leaf, 1, Leaf)`, but unlike in OCaml, these data structures can be built without having to first declare an associated `type` (e.g., a `tree` type).

- MOCaml does not have built-in lists, but it still supports the list syntax. However, now that syntax just serves as a shorthand for building a data structure using the constructors `Cons` and `Nil`. For example, the expressions `1::2::[]` and `[1;2]` are both just shorthands for the value `Cons(1, Cons(2, Nil))`.

- You can assume that no variable name is bound more than once in a pattern (while OCaml checks for this property at compile time).

- MOCaml's binary operators > and = apply only to integers (you should raise a dynamic type error if the operands are not both integers), while they are more general in OCaml.

- The error messages displayed by your interpreter will be different than in OCaml.

- You can exit your interpreter by typing a single period followed by a return.

Here is a description of the various files I've provided and what you must do. **Read the rest of this document carefully and completely before getting started!**

- `lexer.ml` and `parser.ml`: The first step in an interpreter is to *lex* and *parse* a given program. The lexer breaks up a given program (represented as a string) into a sequence of *tokens* (e.g., the keyword "then" will be treated as a single token). The parser checks this sequence of tokens for syntactic correctness (signaling a syntax error otherwise) and produces an *abstract syntax tree* (AST) that unambiguously represents the program. This is a fascinating topic on its own, but is not the subject of this class (you should take CS132!), so I've implemented the lexer and parser for you.

  The two files `lexer.ml` and `parser.ml` contain a lexer and parser for MOCaml; the files were automatically generated from a high-level description of the MOCaml grammar that I wrote using the `ocamllex` and `ocamlyacc` parser generator tools. If you're curious about how these tools work, you can look at the files `lexer.mll` and `parser.mly`, which respectively describe the *tokens* in the MOCaml langauge and the *grammar* of the MOCaml language. *You do not need to understand any of these files in order to complete the assignment.*

- `ast.ml`: This file defines the abstract syntax tree representation for MOCaml programs, which is produced by the parser. An abstract syntax tree (AST) is just a data structure, conceptually similar to the data structure for calculator expressions on Homework 2. In this case, a MOCaml program includes syntax and associated ASTs for patterns, expressions, and declarations. This file also defines the syntax and representation of the *values* that result from executing programs. *Read this file carefully in order to familiarize yourself with our representation of MOCaml programs, which you will need to manipulate in your code.*

  During this assignment it will often be useful to understand how a particular declaration is parsed into our AST representation. For this purpose, my code includes a `toAST` function (declared in `repl.ml`) that provides this information. For example, `(toAST "1+2")` returns `Expr (BinOp (IntConst 1, Plus, IntConst 2))`.

- `env.ml`: Your interpreter needs a notion of *environment* to keep track of the variables in scope, as we've discussed. This file contains a module `Env` that defines a (polymorphic) type `env` for environments along with associated operations. The type `env` represents a mapping from strings (i.e., variable names) to values of some type `'a`. In `ast.ml` the type `moenv` is defined as a version of `Env.env` with `'a` instantiated with our type for MOCaml values, `movalue`. *Read the comments in the file `env.ml` as well as the "module type" definition, which gives the interface to the environment module, in order to familiarize yourself with the type for environments and the associated operations.*

- `interp.ml` declares the three functions that you need to implement. Specifically:
  1. The `evalDecl` function evaluates a given declaration in the context of a given environment. I've implemented the case when the declaration is simply an expression: in that case we simply invoke `evalExpr` (see below) to evaluate the expression. Your job is to implement the cases for "let" and "let rec". Note that these constructs are more restricted syntactically than their counterparts in OCaml: MOCaml syntax for "let" is `let x = e` and its syntax for "let rec" is `let rec f x = e`, where `f` and `x` are both identifiers. The latter syntax ensures that "let rec" can only used to declare functions.

     My code is maintaining the top-level environment for you, which is passed as an argument to `evalDecl`. The `evalDecl` function produces a `moresult` which is a pair of an optional name declared by the declaration along with its associated value. Whenever a `moresult` has a name associated with it, that (name,value) pair is added to the top-level environment by my code, for use by subsequent declarations.

     Suggestion: Implement the simple "let" case, which is useful for testing purposes, and then move on to `evalExpr` (see below) before coming back to "let rec" later.

2. The `evalExpr` function evaluates a given expression to a value, in the context of a given environment. I've implemented the simplest case for you (evaluating an integer constant) but you have to do the rest. Just like OCaml does, your function should signal a match failure exception (by raising the `MatchFailure` exception defined in `interp.ml`) whenever pattern matching fails (see the next item below). Additionally, your interpreter should perform *dynamic typechecking*, raising a `DynamicTypeError` exception whenever a type error is encountered (for example, when an integer is about to be added to a boolean).

   The most interesting part of expression evaluation pertains to functions and function calls. A function evaluates to itself, except that a function value also stores the current environment. *This environment should be used whenever the function body is evaluated during a function call*. Doing so will ensure that you implement OCaml's *static scoping* behavior properly.

   Implementing recursion is tricky because it requires a function value's environment to refer to the function value itself, thereby requiring the creation of a cyclic data structure (which is not possible in the purely functional subset of OCaml). Fortunately there is an easy workaround for the kind of recursion we support: a recursive function's name should not get added to its own environment, but instead the function value should separately store its own name. For that purpose, a function value includes a component of type `string option`, which can be used to store a recursive function's name in the `evalDecl` function above. Then when the function's body is evaluated during a function call, the mapping from the stored name to the function value should first be added to the environment, ensuring that references to the function's name in its own body will be handled properly.

3. MOCaml includes the `match` expression along with several different kinds of patterns. You will therefore need to implement a function `patMatch` that matches a value against a pattern. I have declared this function for you and implemented the simplest case (matching against a pattern that is just an integer constant), but you have to do the rest. A successful match should produce an environment, which maps the variables declared in the pattern to their values. For example, in OCaml the result of matching the value `(1,2)` against the pattern `(x,y)` is the environment mapping `x` to `1` and mapping `y` to `2`. This environment is then used to allow the variables `x` and `y` to be referenced in any expression where they are in scope. An unsuccessful match should raise the `MatchFailure` exception.

- `repl.ml`: This file contains code that implements the MOCaml *read-eval-print loop* or REPL, which continually reads a declaration, parses it, evaluates it by calling your `evalDecl` function, and prints the result. The function `mocaml()` can be invoked to start the MOCaml interpreter. *You do not need to look at this file*.

- `test.ml`: Interpreters are complex and error-prone, so you will want to test yours extensively -- we'll be doing the same as part of grading. To facilitate testing, I've provided a simple test harness in this file -- read the comments to see how to use this test harness. The function `runtests()` will run all the tests for your interpreter. The list `tests` at the top of the file currently has a *very* small number of simple tests for your interpreter. You should add a lot more tests, for common cases, corner cases, error cases, etc. I encourage you to share your tests with others (e.g., on Piazza), to have everyone collaborate to build a comprehensive test suite. You may also find it useful to define a bunch of tests before starting your implementation, to act as a kind of "spec" for what your code should do.

- `hw3.ml`: This file simply loads all of the other files in order of dependencies. Therefore, to load your MOCaml interpreter, type the directive `#use "hw3.ml";;` in the OCaml interpreter. Now type `mocaml();;` to fire up your interpreter or `runtests();;` to run your test suite on your code. *You do not need to understand the code in this file*.