CSE305, Spring 2023 Assignment 7 Due Fri. May 12, 11:59pm

Mini-Project: Interpreting Lines of C Code
(for teams of two, or individual)

Brief Task Statement: Write an OCaml program that completes the following workflow, given one or more lines of simple C assignment statements (in a file):

1. Lex and parse each statement and render the resulting expression tree into an OCaml expression datatype of the kind already seen in lectures and assignments.
2. Then translate the tree into a sequence of postfix commands in our "rudimentary stack-based language"—this is already accomplished on Assignment 5.
3. Design OCaml objects (inside modules) to represent the system stack and external storage of an interpreter.
4. Execute the commands, periodically showing updates to the stack and storage, the latter when stored value(s) change.
5. At the end, output the compiled stack commands and a history of the changes to the variables. Tracing each step of the evaluation stack is highly recommended for debugging but can be commented out for the final product—and you may also comment out the parser trace print code in the supplied `Parser.ml`. You may assign each variable an initial random value.

You are given code `Swi.ml` and `Parser.ml` for the first task. The answer key for Assignment 5 has been modified into a file `Compile.ml` for the second task. One more supplied code file, `PolyOps.ml`, helps solve the issue of emulating arithmetic in C that mixes integers and floats. These files are mirrored in the `OCaml` section of the course webpage (four uppermost files here) and on `timberlake` in the folder (not a URL) `~regan/cse305/LANGUAGES/OCAML`.

The code files for you to write must have names `EvalStackMMM.ml`, `StorageNNN.ml`, and `DriverMMMNNN.ml`, where `MMM` and `NNN` stand for the initials of the team members. Each of the former two must comprise a mutable OCaml `object` that can be manipulated by the driver; it can reside inside a `module` or `class`. The driver file may define its own (`class` or `module` and) `object` with these objects as fields to represent the whole system, or may simply consist of top-level OCaml functions and other code. Your code can be developed on your own machines but must be tested on `timberlake` and submitted via `submit_cse305` there.

The *larger theme* of the project is that we are providing a unique *semantics* for the C statements. It is unique even when the statements are "weird code" that causes differences among prominent compilers. Essay questions and possible short code additions ("virtual" or real) explore this and object-oriented aspects of the code design.

1 Notes and Rules and Points

The project has been simplified and scaled down in the following respects, compared to possibilities from previous assignments and coverage:

(a) The `exp` data type is no longer polymorphic (in the sense of having a generic type variable `'a`); instead, numeric data is stored in string form. Similarly, there is now just a `token` datatype for the command elements.

(b) The `exp` and `token` datatypes are left in extensible form but are defined all-together now. There is no duplication of `pcompile` and other functions, nor are there curried functions like `atostr` since the representation has been fixed as `string`.

(c) Code for arithmetical operations on strings that can hold either `int` or `float` data is given in the file `PolyOps.ml`, rather than trying to grapple with concretely polymorphic functions in OCaml.

(d) Variable names are left as strings and their values can be looked up that way, rather than make allocation with a binding address a design feature. Hence also, the pointer address-of `&`, dereference `*`, and `ArrayEntry` features are not implemented.

(e) Tasks have been structured to minimize the need for IDE-level help and steer clear of incompatible library versions (4.02 on `timberlake` versus 4.06 versus 4.12 versus the new OCaml 5, oh my). See tips below for averting some common "gotchas" of OCaml code.

The project has 45 *individual points* and 105 *joint points*. Followups on the next and last assignment (which will have some short Prolog questions) may bring the total to the vicinity of 180. Formation of teams of 2 should be communicated by email to `regan@buffalo.edu` by **1pm Friday, May 5**. It is also possible to work on all the code individually. The division of labor is:

- One member is responsible for coding `EvalStackMMM.ml`. This object can be an enriched version of the kind of stack shown in lecture, with extra handy stack-manipulation operations besides `push` and `pop`. It does not need to know about the existence of the stack-language module `SL` nor the `token` type—processing that can be left to the shared driver program.

- The other is responsible for coding `StorageNNN.ml`. This can be mostly a "wrapper" for the OCaml library's `Hashtbl` module, which has several handy features and is being described in recitations and lectures. The OCaml Hashtbl reference is well-written. To even-up the individual coding work, this file is required also to provide a method for printing the history of changes to a variable. The changes are automatically preserved by `Hashtbl` (it tacks on new values for keys rather than replace them); the method may make a clone using OCaml's native `{< >}` syntax (exemplified in the supplied `Swi.ml` file) and then call `Hashtbl.remove` until the key is empty to record them all.

- Both may collaborate on `DriverMMMNNN.ml`. Of course, some collaboration will also be needed on deciding the interface of the individual parts and how they work—including whether to return `option` elements ("the safe way") or cut out this extra layer but have to raise exceptions in cases of "stack underflow" or nonexistent values in stored memory. As stated above, use of a `module` or `class` and/or `object` in this file is optional (it may lessen the headache of implementing accumulation parameters and other "recursive juggling").

OCaml desires files to have the same name as a capital-letter module in the file. If there is none, then OCaml capitalizes the name of your code file and pretends under-the-hood that your file is a module by that name. To avoid potential issues, we require that `EvalStackMMM.ml` and `StorageNNN.ml` have modules whose names include the initials. There are `45 individual points` for the individual part and **90 code points** to each team member for the driver, plus **15 points** to each for a short essay answer, which should be in a `(* ...  comment ...  *)` at the bottom of the driver file. Some of the points are for code "quality" and format, including having a header comment with your name(s) and the purpose and usage of the file at the top. *This assignment will not use CSE Autograder.* Doing the above individually does not by-itself carry extra credit. Possible extra-credit options can be enquired after in the second week.

2 Essay Question

Describe the architecture of your system and how the use of objects saved work both in the initial coding and in debugging. Do the eval-stack and storage modules/object need to know any details of the stack language commands, and could they adapt seamlessly to possible extensions in the range of C-language commands emulated? On this latter question, you may imagine what-if the actual code used binding addresses (say, six-digit integers) as keys in the storage lookup, so that potentially an address could be a fetched value of a pointer variable, rather than use variable names as keys directly? (Joint answer, 15 pts. to each team member.)

3 Coding Tips

This list will probably be added to...

1. OCaml has a "Dangling Nested Match" issue that is even thornier than its "Dangling Else" problem. If you use a nested match, e.g. to unpack `Some x` versus `None` when a function returns an `option` value, be sure to enclose the whole nested match expression inside parentheses as `(match ... with ...` and `)` after the last case-alternative; I line up the closing parenthesis with the vertical bars before each case.

2. Remember that writing `foo(x)` does not make `x` associate to `foo` any tighter than `foo x` does. The precedence of the "individual application operator" often makes it advisable to put parentheses *outside* instead, i.e., `(foo x)`. To compose functions `foo` and `bar`, you can't write `foo bar x` because you will get `foo` applied to `bar` instead. You can write `foo (bar x)`, which is equivalent to the usual `foo(bar(x))`, but note that `(foo (bar x))` provides more protection against `foo` being "grabbed" by code that may come before it.

3. OCaml modules use dot `.` but Ocaml objects use `#` not dot. That does bind tighter than the application operator, so you (usually) need not enclose object field access within parentheses.

4. OCaml is like Python in needing `self#` to access methods of the same object. Here `self` is not a reserved word; you can use whatever word you put in place of `self` when the object is declared as `object(self)`.

5. Especially when you use library module functions, take care of when they use currying versus tuple arguments.

6. In a `let`-binding of the form `let x = ... and y = ...`, you get an error if the body of the `y` part uses `x`. Unless the parts separated by `and` are truly parallel, you have to make them sequential via `let x = ... in let y = ... in ...` (yes, this is a "Dangling Let").

7. IMPHO, it is often neater to introduce a side effect `bar(...)` by the code idiom `let _ = bar(...) in ...` rather than use OCaml's semicolon sequencing feature. Using underscore here rather than a "throwaway" variable name avoids an annoying warning about unused variable names.

8. The hardest-to-spot bug in the answer-key code was calling the eval-stack object constructor each time in a recursion, rather than once-only before it. This effectively zapped whatever the previous iteration had added to the stack.

9. When in doubt, use fewer parentheses.

10. When in doubt, use more parentheses.