CSE305, Spring 2023 Assignment 8 Due Fri. May 12, 11:59pm

This assignment is due *by handwritten submission* **to CSE Autograder**. Submission of Prolog code via `submit_cse305` on Timberlake is optional. The questions following up Assignment 7 are all "virtual"—coding the extra features in (b) is not required. *There is no TopHat portion.*

A reminder that the **Final Exam** is on **Tueaday, May 16**, in the lecture room **NSC 216** in the middle **11:45–2:45** time period. There will be an in-person Review Session on the Monday afternoon, time and place TBA. We will prioritize grading this assignment over the project code as its themes (even relating to the project) will matter more to the exam.

Reading for next week is the text chapter 16, especially sections 16.6 and 16.7 with details and criticism of Prolog. Tuesday's lecture will cover what's needed for problem (2) as well as (1). Thursday's lecture will allocate time for project Q & A and summarize course-wide themes, some of which are exemplified in problem (3) below.

(1) Suppose predicates `female(X)`, `male(X)`, `mother(X,Y)`, and `father(X,Y)` (the last means that X is the father of Y) have already been defined, in place of the "one-shot facts" in section 16.6 of the Sebesta text. (Namely, Section 16.6 has a list beginning `female(shelley).` `male(bill).`—ignore that, and note other differences between the following and the text's example.) Use these predicates to define the following as Prolog predicates. Use additional variables `Z,W,...` as needed. Also say which relations are symmetric, and clarify the meaning of any relation that is not symmetric—like I did for `father` above.

```
(a) fullSibling(X,Y)   (defined by sharing /both/ parents)
(b) nephew(X,Y)
(c) granddaughter(X,Y)
(d) firstCousin(X,Y)
(e) descendant(X,Y)  (saying "... :- ancestor(Y,X)." is not allowed)
```

(6+3+3+6+6 = 24 pts. For 9 pts. extra credit, use negation to code `halfSibling(X,Y)` meaning that `X` and `Y` share one parent but are not full siblings.)

(2) Write in Prolog a function `ascenders` that weeds out elements one-at-a-time from a non-negative integer list if they are not greater than all previous elements. E.g. on input `[2,6,3,4,5,8,3]`, your program should output `[2,6,8]` even though `[2,3,4,5,8]` would be a longer ascending chain. Your code should have the form `ascenders(L,Z)` where `Z` is the result variable and is marked for output only—but you may use a "helper predicate" in rules before having `ascenders(L,Z)` "call" the helper. Do you need to use the assumption that the list entries are not negative? Show the result of `ascenders([1,4,4,5,1,8,8,8,3,9],Z)`.

Note that Prolog literal lists have square brackets but comma not semicolon between elements. When matching head and tail, the Prolog symbol to use is not `::` but (confusingly with BNF) a vertical bar, as with `foo([H | Tail])` on the left side of `:-` (and/or on the other side, unlike an OCaml `match`). Note that unlike OCaml, Prolog allows you to use a variable `X` twice in a pattern to match cases where the two occurrences are the same element—in case this helps you. To get started, the fact `ascenders ([],[]).` means that the value given `[]` as input is `[]` as output. (12 pts.)

(3) (3 × 15 = 45 pts. total) *Three followup project questions.* Note especially that the second one describes an idea that does *not* work and asks for your understanding of why-not. This understanding can cite themes from the course, even ones from before spring break. For the first one, it doesn't matter whether you treat throwing an error as a required feature or not in your actual code, so long as you investigate the question with your actual code. It is OK to collaborate with your partner if you have one, but please include at the beginning or end of each one some words like, "Answered jointly with NNN." Each is 15 pts., individually.

(a) First, describe what your code does when simulating a line of C that attempts to read from a variable that doesn't have a value yet, such as `int y = x;` when you haven't initialized `x`. Do you get a runtime error/exception from OCaml itself or an error that you yourself/ves detect and raise an exception for? (Either way, you need not catch the exception via an OCaml `try .. with` clause.)

Then answer, what does your code do on an input like `y = (x = 3) + 5 + x = 4;`—? It is like an example shown in class where in a C code file it gave the error stating that the destination of the rightmost `=` is not an *lvalue*.

If your code throws an error, again say whether it is an exception that you yourself defined (and maybe left uncaught, which again is AOK) or a runtime error from OCaml itself. Say where and why it occurs.

If your code does not throw an error, trace your output and scrutinize it to find a step that seems wrong in terms of the C code—which does not compile. Then sketch how you could detect such an error in either the `Storage` or `EvalStack` object itself (or in your client).

(b) Suppose we wanted to implement the pointer address-of operator `&` and the pointer dereference operator `*`. As these are unary operators that in fact have higher precedence than all the binary operators we are considering, we do not need to modify any of the code in `Parser.ml` to handle them. The object of the game is that we also wish to avoid having to edit any code in `Compile.ml`. Its `CE` and `SL` modules have extensible datatypes, so let's try to take advantage of that by extending them in the client code that we control. Imagine adding the following lines at or near the top of your `StackDriver` file:

```
module CE = struct
   include CE
   type exp += Address of exp | Deref of exp
   let rec ubtree2exp2 ubt = match ubt with
      | Parser.UNode(op, subtree) -> (match op with
         | "*" -> Deref(ubtree2exp2 subtree)
         | "&" -> Address(ubtree2exp2 subtree)
         | _ -> CE.ubtree2exp ubt
        )
      | _ -> CE.ubtree2exp ubt
end;;

let rec pcompile2 (aexp, isLvalue) = match aexp with
   | CE.Address(bexp) -> pcompile2(bexp, true)
   | CE.Deref(bexp) -> (pcompile2(bexp, true)) @ [SL.Fetch]
   | _ -> Compile.pcompile(aexp, isLvalue)
```

Your client would then change any call to `Compile.pcompile` into just `pcompile2`. (If you do `let open Compile` then this would just be changing `pcompile` to `pcompile2`; you could also do lines `module Compile = struct` and `include Compile` before defining `pcompile2` as above—and insert `end;;` after it to re-close the module `Compile`.)

The implementation is correct of-itself: The `*` ensures that what it is dereferencing is an *lvalue* and adds an extra `fetch`; the handling of `&` via `Address` subtracts a `fetch` in statements like `p = &x;` so that it comes out as `p x store pop`. It is AOK here that we are storing `x` as a string value in `Storage.ml`, as it can later be fetched back onto the stack and *then* treated as a key to `store` something else to.[1]

But this code will fail on `p = &x;` or `y = *p;` or anything more complicated than just `&x` or `*p` by themselves. **Say why**. Then say why this issue is not easy to solve—short of rewriting the *entire* `Compile.pcompile` function, which defeats the object of code re-use.

(c) Run your code on a file of the C assignment statements on Assignment 6, problem (1). You may rewrite the line `i += 7*(i = 4);` as `i = i + 7*(i = 4);`.[2]

Which of the three C compiler `gcc`, `CC`, or `clang`, or the Java or Javascript systems, does your stack-code execution agree with, if any? If you get agreement, try some other "weird" expressions with embedded assignments and/or pre-or-post increments and see if the agreement persists. What does all this say about (i) the ability of OCaml to provide a *unique semantics* for C expressions and assignments, in relation to (ii) the difficulty of getting agreement on such a semantics?

This is the end of Assignment 8, giving 24 + 12 + 3×15 = 81 points total.

The rest gives some more project tips and options, some of which have already appeared on *Piazza*. Picking up from the previous list 1.–10. of OCaml how-tos and pitfalls:

11. If you do something like `exception SegFault of string` rather than just `exception SegFault`, you get the kind of exception where you can print a customizable er-

[1]The whole project could have had the direction of defining a function `allocate` that would "allocate" to any variable `x` a 6-digit (say) integer representing its binding address. Then the implementation of the storage would have been a hash table representing a map of type `int -> string` rather than `string -> string`. This would be more realistic for a compiled language, but actually our use of a symbol-table of variables by their string names is closer both to interpreted implementations of Python and to the working of (the "pure" part of) OCaml itself—where names are bound to values. Doing `allocate` would, however, complicate the fault detection in part (a). If you required addresses to be 64-bit word-aligned, then for any `store` to an unintended ("random") integer, you would have a 7-in-8 chance of its not being word-aligned. This is a fault in memory segmentation. Another complication is that whereas `x++` for an `int x` always means adding 1, for a pointer to an object of size `e` in bytes it means adding `e`. Then and for `ArrayEntry` you would really need to maintain the keys as integers. **Getting back to what we're actually doing**, we are fine having the keys be the literal variable names—and this would stay fine even if we had implemented `&` and pointer `*` from the get-go.

[2]Or you can implement `+=` in your code. Unlike the case of `&` and `*` in (b), you can fairly-readily do so without trying to extend the datatype and/or the `pcompile` function, at least when the item before `+=` is a simple variable name like `foo`. You would make your client code **rewrite** the substring `foo +=` as `foo = foo +` before it ever gets to the `Parser.parse` function; this is a little tricky but doable by giving your line of text to an `Swi.swi` object and using `getItem` and saving `foo` as the previous item when a `+=` is encountered. Incidentally, you can also (more easily) modify your client code to simply skip lines with `main`, `printf`, or curly braces in them; then you can run your OCaml code on actual C programs without having to make a separate file of the important lines.

ror message. When you `raise` it, however, you have to beware of the parentheses. If you go `raise SegFault("Tried to store to " ^ k)` with syntax typical of other languages, OCaml treats it as a "curry" of the `raise` operation and tries to execute just `raise SegFault`. That leads to a type error. The attitude in OCaml is that giving the string is needed to finish constructing the exception to begin with. It's not enough to move the first paren outside, because `raise (SegFault "Tried to store to " ^ k)` would then do a "curry grab" on the inside. It has to be `raise (SegFault ("Tried to store to " ^ k))`.

12. Here are the lines of code given in Piazza post @232 for reading lines from a file. They modify the routine given in a StackOverflow item. If you modify it further, then I would judge it crosses the line where you wouldn't need to cite StackOverflow. The behavior of skipping blank lines was added by KWR. You can extend its if-then-else logic to make it skip other kinds of lines if you wish, such as those suggested in footnote 2.

```
let readLines filename =
  let ic = open_in filename in
     let tryRead () = try Some (input_line ic) with End_of_file -> None in
        let rec loop acc = match tryRead () with
          | Some s -> let line = String.trim(s)
                      in if line = "" then loop acc     (*skip blank lines*)
                                      else loop (acc @ [line])
          | None -> close_in ic; acc
        in loop []
```

13. If your `EvalStack` or `Storage` module object does not compile, then there will be a further error in your `StackDriver` that the object is `unbound`. Fixing the former error should fix the latter, so don't worry further about it.

14. The one frequent conceptual error from questions this past week was thinking that `EvalStack` should store the list of stack-language *commands*. The strongest reply is to affirm general the software engineering principles of separation of concerns, single focus of control, and the kind of abstraction that simplifies life: Neither `EvalStack` nor `Storage` needs to know there are such things as stack-language commands. Neither should reference `Compile.ml`, nor `Parser.ml`, nor (yuck) `PolyOps.ml` (they can use `Swi.ml` as a general utility module but my code manages without). `Storage` may have methods you call `fetch` and `store`, but they don't have to read the `Fetch` and `Store` commands output by `pcompile` to act. Your client driver should invoke all that.

15. Adding extra stack-management methods to `EvalStack`, such as replacing the top two values by a third—rather than have the client call two pops and a push each time—is the best simple "pre-debugging" advice.

16. Another good idea is to make a separate function (or method of an extra class-or-object-or-module in the driver file) that processes just *one* command. Then have a different function recurse on the whole list of commands and call it for each one.

17. Test and debug your code using lines of C that do not have embedded assignments—on which the compilers used in Assignment 6 all agree. Only then try the weirder ones.