Extensions:

1. Additional Atomic Types and Functionality
2. Lazy programming
3. Wildcard Variable Names
4. Lexical Scoping (eval_l and eval_e)
5. Mutable Operators and Stores (eval_e)
6. Lists
7. Closure Lists (eval_l and eval_e)
8. "Syntactic Sugar"
   A) Easy list construction (e.g. [1;2;3] and [ ] and [4; 5;])
   B) Easy currying (e.g. let f x y z = … in …)
   C) `x; y` as shorthand for `let ( ) = x in y`

---

1. Additional Atomic Types and Functionality (all evaluators)

Binop *Equals*:

Added functionality for all atomic types

Added first-order static typing (i.e. 1 = true will raise an exception, but not [1] = [true])

Binop *LessThan*: Added functionality for floats

Int (as Num): Added binop *Divide*

Float: Added unop *FNegate*, Added binops *FPlus, FMinus, FTimes, FDivide*

Char, String: Added binop *Concat* (for String)

Unit: Added constructs `fun ( ) -> …`, `let ( ) = … in …`

These constructs are added as separate `expr` constructors, because `Fun` and `Let` do not allow for the `Unit` constructor to take the place of the variable name. Units were added as a separate constructor (instead of a matched "( )" `varid`) because the type-inference of type unit would be compromised, leading to a poisoning of all constructs that use type unit, such as all `varid` calls needing to check for a match to "( )". Also, a `varid`-match implementation would compromise the variable space inside the unit. The extended parser allows for the user to input "( )" and still get type unit.

2. Lazy Programming (all evaluators)

The implementation of the *FunUnit* (fun ( ) -> …) and *LetUnit* (let ( ) = … in …) constructs allow for the usage of unit-based lazy programming. The gensym function seen earlier in the semester can be fully implemented in the miniml.ml REPL (of course, only in local scopes).

3. Wildcard Variable Names (all evaluators)

Implemented the *FunWild* (fun _wildcard -> …) and *LetWild* (let _ = … in …) constructs. While extremely similar to the *FunUnit* and *LetUnit* constructs, the wildcard allows for the evaluator to completely ignore the results of an expression: let _ = 1 in 2 is well-formed, but let ( ) = 1 in 2 raises a contradiction between 1's int typing and ( )'s unit typing. The implementation of wildcard variables also serves to adhere to OCaml's convention that no variable name starts with '_'. This constraint is used later in the eval_e implementation of stores, where all location names begin with '_'.

4. Lexical Scoping (eval_l and eval_e)

Implemented lexical scoping of environments when defining functional constructs. Because of the use of value refs in the Env module's environment typing, dynamical and lexical environments had to be separated physically from each other in the *Let* construct. `Env.copy` was implemented to take an environment and create a structural copy, allowing for the lexical environment of a function to maintain independence from the dynamical environment, but still having access to structural changes as when a function's lexical variable is itself a dynamically mutable value.

5. Mutable Operators and Stores (eval_e)

eval_e takes advantage of its flexible typing in the project's specifications to take a third argument: a store environment. While eval_e could have included the store inside the established `env` argument, since all location names are filterable from variable names, the extra barrier between variables and locations was cleaner and more futureproof. The location names mentioned above come from the `Expr.new_refname` function, which is a parallel gensym that prefixes all location names with "_l". This simultaneously guarantees easy, first-order type inference for refs (i.e. a variable's value in the environment is immediately apparent to be a true value or a ref of a value), and also takes advantage of the wildcard implementation to prevent an accidental user-declaration of a location name. *Letrec* no longer uses OCaml's native ref system for implementation, now fully self-evaluated. In addition to the standard `ref`, `!`, and `:=` operators, the extended parser interprets `x; y` as `let ( ) = x in y`.

Another implementation choice was the flipping of evaluation order of binops. The textbook lists the semantic rules as P op Q evaluates with P's store update first then Q's second. However, this project made the choice of evaluating Q's store changes first and P's second. This is because of OCaml's native handling of binops in the same way. `let x = ref 0 in (incr x; !x) – (incr x; !x)` evaluates to 1 in the native REPL, and seemingly all binops (not just (-)) work the same way – even lists defined as [1;2;3] are evaluated starting from the right (because of the implicit (::) chain). As such, the extension uses Q-then-P evaluation for store updating.

6. Lists (all evaluators)

Implemented lists as *List* and *ClosList* constructors, with unops *Head* and *Tail* and binop *Cons*.

Lists were the most difficult extension to implement in this project. Learning recursive parsing for long lists was already difficult, but having to disambiguate the list-construct semicolon from the binary sequencing semicolon at the same time was quite challenging. Several layers of abstraction in the parser eventually led to a working solution.

7. Closure Lists (eval_l and eval_e)

Another disproportionate challenge were the functional lists possible in the lexically-scoped evaluators. `[(+), (-)]` is an (int -> int -> int) list, but in eval_l and eval_e, they are also closures. Incompatible with the basic *List* constructor, a different *ClosList* constructor allows for lists of closures, with a couple layers of abstraction. A *ClosList* is a list of pseudo-reference `varid`s, and do not actually contain closures. Instead, the list operators in the evaluator pull the *Closure* stored in the environment under that name. Such a roundabout method was necessary because of the separation of the declarations of types `Expr.expr` and `Evaluation.Env.value`, and also to not break the abstraction barrier of module Env by exposing type `env` as a list.

8. Syntactic Sugar (all evaluators)

This extension was carried out in the menhir lexer and parser. Mentioned before is the easy list construction of [ ], [1], [1;], [1; 2; 3; 4; …]. This required OCaml-specific regex syntax in the lexer, specific cases for empty lists and with-or-without semicolon endings, recursive parsing in the parser, as well as the separate abstraction of list internals and binary sequencing expressions for disambiguation.

The knowledge of recursive parsing, however, was useful in functional syntax. `let f x y z = … in …` shows that the currying of functions, as well as shorthand function definition is implemented in the parser.

Binary sequencing (x ; y) is implemented as `let ( ) = x in y`, taking advantage of the already-implemented unit constructs and avoiding the duplicate functionality of a binop implementation.