This assignment consists of five written exercises and one coding. Submit your written solutions on Gradescope under "A2 written". The coding assignment will automatically be uploaded to our server whenever you grade. Your Gradescope submission time for the written portion determines whether a late day will be used, so submit that after the deadline even if you only need extra time for the coding portion. Your code snapshotted at the time of the regular or late deadline, respectively, will be graded.

Part I
Equational and Inductive Proofs

Before starting on this exercise, review the material on inductive proofs in lecture and discussion.

Question 1: Map Fusion (Written, 25%) In this exercise, we will prove that mapping twice over a list, first with a function `g`, then with another function `f`, is equivalent to mapping once with the composition of `f` and `g`, which we define via a higher order function, `compose(f, g)`. This can be pronounced "`f` of `g`" (and some languages define an infix operator for it, e.g. dot in Haskell.)

Motivation. Due to memory locality and caching, it is faster to `map` once rather than twice, so this equivalence, known as "map fusion", can be applied as an optimization (and indeed, it is in many industrial functional languages and libraries developed using functional principles.)

Definition. Assuming that `a`, `b` and `c` are arbitrary Hazel types, consider the following definitions of `compose` and `map`.

```
let compose: ((b -> c), (a -> b)) -> (a -> c) =
  fun (f, g) ->
    fun x -> f(g(x))
in
let map: (a -> b, [a]) -> [b] =
  fun (f, xs) ->
    case xs
      | nil => nil
      | hd::tl => f(hd)::map(f, tl)
    end
in ...
```

Theorem (Map Fusion)**.** *For all* `xs: [a]` *and* `g: a -> b` *and* `f: b -> c` *, we have that*

$$\texttt{map(f, map(g, xs))} \equiv \texttt{map(compose(f, g), xs)}$$

Recall the induction principle for lists:

Definition (Induction Principle for Lists)**.** To prove a property $P(\texttt{xs})$, for all lists `xs`, it suffices to prove:

1. **Base Case**: $P(\texttt{nil})$
2. **Inductive Step**: $P(\texttt{hd::tl})$ for all `hd` and `tl`, assuming $P(\texttt{tl})$, which we call the *induction hypothesis* (IH).

The template is provided on the next page. You may combine function application and the subsequent `case` evaluation into a single equivalence step labeled "application of `f`" where `f` identifies the function. You do not need to inline the function definitions—you may assume they are defined as above. Show all other equivalence steps individually **and label each step**, including any applications of the IH. Write out all expressions fully—do **NOT** use ellipses (...) or other abbreviations. You may want to review the material in Lecture 5.

Proof. We proceed by list induction on `xs`.

(a) (5 points) What is the property, $P($`xs`$)$, that we will need to use for this proof? **HINT**: don't forget necessary "for all"s.

(b) (5 points) Prove the base case, $P($`nil`$)$.

(c) (5 points) What is the induction hypothesis, $P($`tl`$)$?

(d) (10 points) Prove the inductive step, i.e. $P($`hd::tl`$)$ for all `hd` and `tl` assuming the IH, $P($`tl`$)$.

□

Part II
Syntax

Question 2: Precedence and Associativity (Written, 20%)

1. Fully parenthesize the following expressions according to Hazel's concrete syntax. In other words, put parentheses around the top-level expression, as well as any child expressions and types, all the way down but not including the variables and numbers at the leaves of the tree.

   For example, given the following expression:

   ```
   let f : Int -> Int = fun x -> x + 3 * 2 in f(10)
   ```

   your answer should be:

   ```
   (let (f : (Int -> Int)) = (fun x -> (x + (3 * 2))) in (f(10)))
   ```

   HINT: you may want to enter these expressions into Hazel and walk around with your cursor, observing the visual indications provided. The original expression and its fully parenthesized version should behave identically (they are the same underlying structure). You may need to define suitable variables to test the behavior of the examples below. Otherwise, the variables will be undefined and highlighted with a red box.

   (a)
   ```
   5 * 3 + g(8) - f(6 - 2 * 4), 3
   ```

   (b)
   ```
   3 - 2::length(a) * 1::[] + 9::[]
   ```

   (c)
   ```
   let h: Bool -> String -> [Int] =
     fun b -> fun s ->
       if string_length(s) < 0 && !b
       then string_length(s)::[]
       else []
   in h(a)(b)
   ```

   (d)
   ```
   let foo = fun (a, b) ->
     let bar = fun x: Bool ->
       if a || x && b then x::b && a::[] else []
     in bar(false)
   in foo
   ```

2. Given the following hypothetical operator table, where higher precedence binds tighter, fully parenthesize the following expressions.

Operator	Precedence	Associativity
#	1	left
@	2	right
*	3	left
;	4	right
!	5	left

(e) `f * e @ d @ c * b ; a`

(f) `a # h ; h # e @ e * c ; s ! 4 ! 9 ! 0`

(g) `a * b ! c # d ; e @ f @ g ; h ! i @ j ! k`

Part III
AL

In this section, we will work with our first language definition. AL is a simple language of arithmetic expressions.

Syntax

We can define the structural and concrete syntax for AL expressions in the following syntax table, refactored slightly from lecture to group together all the operators. Here, n stands for integers. We give precedence and associativity inline for simplicity.

Note that this differs slightly from the definition in lecture in that we group together unary and binary operators into common structural forms.

			Structural Syntax	**Concrete Syntax**
Expr	e, v	::=	NumLit$[n]$	$\underline{n}$
		|	UnOp$[u](e)$	ue
		|	BinOp$[b](e, e)$	$e\, b\, e$
UnOp	u	::=	OpNeg	$-$ (precedence 3)
BinOp	b	::=	OpPlus	$+$ (left assoc., precedence 1)
		|	OpMinus	$-$ (left assoc., precedence 1)
		|	OpTimes	$*$ (left assoc., precedence 2)

Question 3: Parsing (Written, 10%) Parse the following AL expressions, i.e. write them in the corresponding structural syntax according to the table above.

1. $\underline{10} + -\underline{20} * \underline{30}$

2. $\underline{2} - \underline{3} + \underline{4} * \underline{5} - \underline{6}$

Evaluation Semantics

We will now consider the semantics of AL. Recall from Lecture 6 that we define the semantics of language as a collection of *judgements*, which are relations between syntactic structures defined by a collection of rules.

We will now define two judgements, e **val** and $e \Downarrow v$, that together constitute the evaluation semantics of AL.

$\boxed{e\ \textbf{val}}$ e is a value

$$\frac{}{\mathsf{NumLit}[n]\ \textbf{val}}\ \text{(V-NumLit)}$$

The value judgement, e **val**, has just one rule, which we call (V-NumLit), with no premises. It defines number literals as the only values in AL. This notation for rules is shorthand for the following logical statement:

We assume that all metavariables that appear in rules, like n in this rule, are universally quantified, i.e. under a "for all", without saying so explicitly. So this rule is just a fancy way of declaring that "for all n, the AL expression $\mathsf{NumLit}[n]$ is an AL value". Recall from your discrete math course that the notation above, using a horizontal line with nothing above, is just another way of stating an axiom, i.e. there are no "if ... then" conditions here – every number literal is a value.

We could also have written the same rule using concrete syntax (and we will do so in future assignments):

$$\frac{}{\underline{n}\ \textbf{val}}\ \text{(V-NumLit)}$$

We will use the metavariable v from here on out for expressions that we know to be values. In AL, these are only number literals, but in future languages, there will be more values.

$\boxed{e \Downarrow v}$ e evaluates to v

The evaluation judgement, $e \Downarrow v$, defines the evaluation behavior of AL expressions, i.e. it relates an expression e to its value v. It is defined by the following rules.

$$\frac{}{\mathsf{NumLit}[n] \Downarrow \mathsf{NumLit}[n]}\ \text{(Eval-NumLit)}$$

$$\frac{e \Downarrow \mathsf{NumLit}[n] \qquad -1 \times n = n'}{\mathsf{UnOp}[\mathsf{OpNeg}](e) \Downarrow \mathsf{NumLit}[n']}\ \text{(Eval-Neg)}$$

$$\frac{e_1 \Downarrow \mathsf{NumLit}[n_1] \qquad e_2 \Downarrow \mathsf{NumLit}[n_2] \qquad n_1 + n_2 = n}{\mathsf{BinOp}[\mathsf{OpPlus}](e_1, e_2) \Downarrow \mathsf{NumLit}[n]}\ \text{(Eval-Plus)}$$

$$\frac{e_1 \Downarrow \mathsf{NumLit}[n_1] \qquad e_2 \Downarrow \mathsf{NumLit}[n_2] \qquad n_1 - n_2 = n}{\mathsf{BinOp}[\mathsf{OpMinus}](e_1, e_2) \Downarrow \mathsf{NumLit}[n]}\ \text{(Eval-Minus)}$$

$$\frac{e_1 \Downarrow \mathsf{NumLit}[n_1] \qquad e_2 \Downarrow \mathsf{NumLit}[n_2] \qquad n_1 \times n_2 = n}{\mathsf{BinOp}[\mathsf{OpTimes}](e_1, e_2) \Downarrow \mathsf{NumLit}[n]}\ \text{(Eval-Times)}$$

In the rules above, the space-separated judgements above the line are called the *premises*. They must all be satisfied in order for the conclusion to hold. For example, the rule (Eval-Plus), if written out in long-hand, is:

For all e_1 *and* e_2 *and* n_1 *and* n_2 *and* n*, if* $e_1 \Downarrow \mathsf{NumLit}[n_1]$ *and* $e_2 \Downarrow \mathsf{NumLit}[n_2]$ *and* $n_1 + n_2 = n$ *then* $\mathsf{BinOp}[\mathsf{OpPlus}](e_1, e_2) \Downarrow \underline{n}$.

Question 4: Convert Rules to Concrete Syntax (Written, 10%) Rewrite the above rules **using concrete syntax** rather than structural syntax, but **without using the inference rule (i.e. horizontal rule) notation**. Instead, write them out long-hand as logical statements, as in the example above. Don't forget the "for all"s.

1. `Eval-NumLit`

2. `Eval-Neg`

3. `Eval-Plus`

4. `Eval-Minus`

5. `Eval-Times`

The following theorem, which you do not have to prove, states that evaluation always produces a value.

Theorem 1 (Evaluation)**.** *If* $e \Downarrow v$ *then* v **val**.

Why is this theorem true? Observe that all of the evaluation rules above result in a number literal, and that the (V-NumLit) rule establishes that all number literals are values.

To apply these rules to establish a conclusion of the form $e \Downarrow v$, we can stack these rules one atop another, labeling each step with the rule and recursively providing proofs of each of the premises above the line. This is called a *derivation*. For example, the following derivation proves that 2 and 2 adds up to 4 in AL:

$$\dfrac{\dfrac{}{\mathsf{NumLit}[2] \Downarrow \mathsf{NumLit}[2]}\ \text{(Eval-NumLit)} \qquad \dfrac{}{\mathsf{NumLit}[2] \Downarrow \mathsf{NumLit}[2]}\ \text{(Eval-NumLit)} \qquad \dfrac{}{2+2=4}\ \text{(arith.)}}{\mathsf{BinOp}[\mathsf{OpPlus}](\mathsf{NumLit}[2], \mathsf{NumLit}[2]) \Downarrow \mathsf{NumLit}[4]}\ \text{(Eval-Plus)}$$

To make things more concise you can define *derivation abbreviations* using a capital letter $\mathcal{D}$ and a subscript of your choice, e.g. we can write the above more neatly as follows:

$$\dfrac{\mathcal{D}_1 \qquad \mathcal{D}_1 \qquad \dfrac{}{2+2=4}\ \text{(arith.)}}{\mathsf{BinOp[OpPlus](NumLit[2], NumLit[2])} \Downarrow \mathsf{NumLit[4]}}\ \text{(Eval-Plus)}$$

where $\mathcal{D}_1 =$

$$\dfrac{}{\mathsf{NumLit[2]} \Downarrow \mathsf{NumLit[2]}}\ \text{(Eval-NumLit)}$$

Question 5: Derivation (Written, 15%) In the examples above we used the structural syntax. Derive the following judgement using your rules from the previous problem. You can define derivation abbreviations as needed.

$\mathsf{BinOp[OpMinus](BinOp[OpMinus](NumLit[4], BinOp[OpTimes](NumLit[3], NumLit[2])), NumLit[1])} \Downarrow \mathsf{NumLit}[-3]$

Question 6: Evaluator (Learn OCaml, 20%) Implement an evaluator (a.k.a. interpreter) for AL in OCaml, a real-world functional language similar to Hazel, following the rules given above. Instructions for accessing and using the Learn OCaml platform we will be using, and for understanding the OCaml language and its differences from Hazel, are in the appendix below.

`http://eecs490.eecs.umich.edu`

The correctness theorem for your implementation is given below. Here, $\lfloor e \rfloor$ refers to the OCaml encoding of the syntactic structure e. For example, $\lfloor \mathsf{NumLit[2]} \rfloor =$ `NumLit(2)` in OCaml.

Theorem 2 (Evaluator Correctness)**.** `eval_expr` $\lfloor e \rfloor \equiv \lfloor v \rfloor$ *iff* $e \Downarrow v$.

This theorem implies that e is the input and v is the output of your evaluator. You should produce exactly the outputs for which a derivation is possible. Keep the rules in front of you as you write your code. If a premise is an evaluation judgement, that corresponds to a recursive call. You may need to pattern match using **`match`** on the result of the recursive call.

Appendix

A Learn OCaml Platform

We will be using the Learn OCaml platform for coding questions on this assignment.

For this assignment, you will complete the coding exercise on the Learn OCaml platform at the following URL:

`http://eecs490.eecs.umich.edu`

Getting Started

1. Go to the URL above.
   - Note that `https` does not work, and if your browser forces https you may have to look into disabling that for the time being. Firefox tends to be a bit more relaxed about this than Chrome-based browsers.)
   - You must be on the UM network (either on campus or via VPN) to access the system. (We are looking into loosening this restriction which is imposed by the department.)
2. Generate a token by clicking "New user? Create a new token". **You must use your UMich uniqname as your nickname, i.e. the first part of your email address, otherwise you will not receive credit for your work.**

   Upon clicking "Create new token", Learn OCaml will generate a unique identifier token for you. **Write it down or save it somewhere private and accessible so you don't lose it**; you will need your token to access your saved work over the course of the semester. Treat it like a password.

   Note: If you lose your token, we can retrieve it for you but you will be penalized. **You will lose 3 points on the current assignment every time you lose your token.**

   Similarly, if you do not follow these instructions and we have to manually change your login information or find your assignment, **You will lose 3 points each time this is necessary.**
3. Click the **Exercises** button on the left. The exercise will be listed there.
4. Complete the exercise, clicking "Grade" as many times as you like for grading feedback. You will be asked to write tests, discover hidden bugs, and implement your solution as in the Hazel assignments, but the feedback looks a bit different.
5. Your solutions will be saved to our server whenever you click Compile or Grade, so you do not need to submit them to Gradescope yourself.

B From Hazel to OCaml

Every language feature in Hazel has a corresponding language feature in OCaml, usually with the same syntax. Some small differences are:

- Primitive types are written with lowercase letters, e.g. `int` rather than `Int`.
- Recursive functions must be defined using **`let rec`** rather than just **`let`**:

```
let rec fib = fun n ->
  if n < 2 then 1 else n * fac(n - 1)
```

- Top-level (module-level) **let** doesn't require an **in**, e.g. the example above, but **let** bindings inside functions do require an **in** still.
- OCaml has syntactic sugar for defining functions:

```
let rec fib n =
  if n < 2 then 1 else n * fac(n - 1)

(* same as above, but written without syntactic sugar *)
let rec fib = fun n ->
  if n < 2 then 1 else n * fac(n - 1)
```

- The implementation exercise asks you to use the function composition operator, `%`. This is defined in the prelude, and can be used as follows:

```
let y = f % g

(* same as *)
let y = fun x -> f(g(x))
```

- Function application can use a space rather than parentheses, which is a left-associative operator, e.g.

  ```
  f x y z
  ```

  is parenthesized

  ```
  (((f x) y) z)
  ```

  This is called currying. We will learn more about currying next week. For this week, the required exercises only involve single-argument functions. The optional exercises might exercise these ideas, though. For example, you can call a left fold as follows:

  ```
  fold_left f a xs
  ```

- Tuple types are written `T1 * T2 * ... * Tn` rather than using commas.
- List elements are separated using semicolons, e.g. `[1; 2; 3]`. The empty list is written `[]`.
- Types and functions can be parameterized over arbitrary types, written `'a`, `'b`, `'c`, etc.
  - Parameterized types are written with the parameter(s) before the type name, e.g. `'a list` is the type of lists with elements of some arbitrary type `'a`.
  - Multiple parameters are grouped using parentheses, e.g. `('a, 'b) p` is a type `p` with parameters `'a` and `'b`. Note that this has nothing to do with tuples!
  - Functions can be parameterized over arbitrary types as well, for example:

    ```
    fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
    ```

- Sum types are defined as follows. Note the use of `|` instead of `+` and **of** rather than parentheses.

```
type bread =
| White
| Wheat
| Multigrain of int
```

- Pattern matching has the syntax `match e with ...` rather than `case e ... end`:

```
let num_grains (b : bread) =
  match b with
  | White => 1
  | Wheat => 1
  | Multigrain n => n
```

Careful with nested matches – they should be parenthesized to avoid ambiguity:

```
match xs with
| [] -> 0
| hd::tl ->
  (match f(tl) with
  | 0 -> 1
  | n -> n)
```

Tips

1. Learn OCaml works best on Firefox or Chrome or Chromium-based browsers (Opera, Brave, Edge, etc.). If you are having trouble with it loading at any point this semester, try switching to such a browser.
2. Do not have multiple instances of Learn OCaml open on the same or different machines, as this may lead to issues with synchronizing your work to the server.
3. Don't forget to complete the test cases. Also note that the autograder will run its own test cases in the background, so your implementations will need to be correct.
4. Hitting "Compile" will check your code for errors. If it finds any, it will color the line number of the offending line along the left border of the editor. You can hover over the line number to see the error message.
5. Hitting "Grade!" will let run the autograder on your code and generate a grade report that will be synced with the server. You may grade your code as many times as you wish. The only thing that matters is the final grade report generated before the deadline.
6. Use the "Sync" button to save your work on the server. While we do our best to keep your work safe, the system does occasionally have hiccups especially if you open multiple instances of Learn OCaml. We strongly recommend that you save your work to a local file periodically to avoid losing work.
7. Use the "Eval code" button to load your code into the REPL and interact with it. This will also give more detailed compiler feedback.
8. If you would like to use a late day, submit your written portion late to Gradescope. If you do not do this, we will use the state of your code at the non-late deadline, even if you have made subsequent changes.

9. Getting 100% on Learn OCaml doesn't mean you'll get 100% on the exercise. The system uses random test cases, so you may want to grade multiple times to make sure you really have full credit.

   Additionally, points will be taken off for overly complex solutions or egregiously bad style. We won't take off points for small issues like variable names or minor indentation inconsistencies. We cannot provide an exhaustive list of what will result in deducted points. However, here are some common errors to avoid:

   - Unnecessary use of helper functions
   - Unnecessary use of exceptions
   - Using `List.hd` or `List.tl` instead of pattern matching.
   - Using imperative features of OCaml (loops, references) – we will not discuss these until the next assignment and you may not use them on any assignments unless otherwise specified.
   - Excessive base cases

     ```
     let rec fib n =
         match n with
             | 0 -> 0
             | 1 -> 1
             | 2 -> 1
             | 3 -> 2
             | 4 -> 3
             | 5 -> 5
             | 6 -> 8
             | _ -> fib (n - 1) + fib (n - 2)
     ```

   - Using the physical equality operator `==` rather than structural equality, `=`.
   - Misusing if expressions and equality when a **`match`** expression is more appropriate.

     ```
     type color =
         | Red
         | Orange
         | Yellow
         | Green
         | Blue
         | Purple

     (* poor style *)
     let is_warm c =
         if c = Red || c = Orange || c = Yellow then
             true
         else
             false

     (* good style *)
     let is_warm c =
     ```

```
match c with
| Red | Orange | Yellow -> true
| _ -> false
```