Project Overview:
This project involves the implementation of MicroCaml, a variant of OCaml with a reduced feature set and dynamic typing. Unlike OCaml, MicroCaml lacks compile-time type checking, similar to scripting languages like Python and Ruby, where type validation occurs during runtime. Additionally, the project includes the development of mutop (Microtop), a version of utop tailored for MicroCaml.

Part (A): Lexical Analysis & Parsing
The initial phase of this project involves the implementation of a lexer and parser for MicroCaml. The lexer function will translate a MicroCaml input string into a token list, while the parser function will consume these tokens to generate an abstract syntax tree (AST) representing either a MicroCaml expression or a mutop directive.

Below is an example invocation of the lexer and parser on a MicroCaml mutop directive provided as a string:

```
parse_mutop (tokenize "def b = let x = true in x;;")
```

This call will yield the following OCaml value representing the AST (further explanation to follow):

```
Def ("b", Let ("x", false, (Bool true), ID "x"))
```

Part (B): Implementation of Interpreter

In this segment of the project, your task is to develop an interpreter capable of executing the ASTs generated in Part (A).

You'll be creating two functions: eval_expr and eval_mutop. Both functions accept an environment (defined in types.ml) as an argument, serving as a mapping from variables to expressions.

The eval_expr function evaluates an expression within the provided environment, returning an expr, while eval_mutop processes a mutop (a top-level directive), returning a potentially updated environment along with any additional results.

You'll utilize Imperative OCaml, especially references, in this section, which is crucial. Further details are provided below.

Ground Rules and Additional Information

In addition to the standard OCaml features and those taught in class, you're permitted to employ library functions available in the Stdlib module, Str module, List module, and the Re module.

Testing and Submission

Submission is done by executing 'submit' after pushing your code to GitHub.

All tests will be conducted by directly calling your code and comparing the returned values against expected results. Any other output, such as for debugging purposes, will be disregarded. While error handling isn't explicitly specified, input that fails to be lexed/parsed according to the provided rules should raise an InvalidInputException. It's recommended to include relevant error messages with these exceptions to facilitate debugging. Although intelligent error messages pinpointing errors aren't mandatory, you might find them helpful as you progress through the project.

To test from the toplevel, execute 'dune utop src'. The necessary functions and types will be automatically imported for your convenience.

You can also devise your own tests to specifically assess the parser by supplying it with a custom token list. For instance, to examine how the expression 'let x = true in x' would be parsed, you can manually construct the token list (e.g., within utop).

```
parse_expr [Tok_Let; Tok_ID "x"; Tok_Equal; Tok_Bool true; Tok_In; Tok_ID "x"];;
```

This approach allows you to focus on developing the parser independently, even if your lexer isn't fully functional yet.

It's worth noting that a fully operational lexer and parser aren't prerequisites for implementing part (B) – you can conduct all testing directly on abstract syntax trees without relying on the lexer and parser.

```
eval_expr [] (Let ("x", false, (Bool true), ID "x"));;

- : expr = Bool true
```

To interact with the interpreter, provided you have functioning implementations of the parser, lexer, and evaluator, you can run the following command: `dune exec bin/mutop.exe`

This command launches the interpreter, enabling you to experiment with your interpreter implementation interactively.

```
$ dune exec bin/mutop.exe

mutop # def a = 3;;

val a = Int 3

mutop # def b = 5;;

val b = Int 5

mutop # a * b;;

- : val: Int 15

mutop #
```

Note that the `mutop` toplevel uses your implementations for `parse_mutop` and `eval_mutop` to execute MicroCaml expressions.

For Part A1, you're tasked with implementing the lexer, also known as the scanner or tokenizer. Your parser will rely on the output of this lexer, which converts the input string into a list of tokens.

Lexing is typically achieved using regular expressions, as demonstrated in class. While you're not obligated to use regex functions, they can be incredibly useful.

Your lexer implementation should reside in `lexer.ml`, and it should contain the following function:

ocaml

Copy code

```
tokenize : string -> token list
```

This function converts MicroCaml syntax provided as a string into a corresponding list of tokens.

Here's an example of how the function should behave:

ocaml

Copy code

```
tokenize "let x = 5 in x;;"
```

This call should return a list of tokens representing the provided MicroCaml syntax.

Examples:

tokenize "1 + 2" = [Tok_Int 1; Tok_Add; Tok_Int 2]

tokenize "1 (-1)" = [Tok_Int 1; Tok_Int (-1)] tokenize ";;" = [Tok_DoubleSemi]

tokenize "+ - let def" = [Tok_Add; Tok_Sub; Tok_Let; Tok_Def]

tokenize "let rec ex = fun x -> x || true;;" =

- [Tok_Let; Tok_Rec; Tok_ID "ex"; Tok_Equal; Tok_Fun; Tok_ID "x"; Tok_Arrow; Tok_ID "x"; Tok_Or; Tok_Bool true; Tok_DoubleSemi]

The `token` type is defined in types.ml.

Notes:

- The lexer input is case sensitive.
- Tokens can be separated by arbitrary amounts of whitespace, which your lexer should discard. Spaces, tabs ('\t') and newlines ('\n') are all considered whitespace.
- When excaping characters with `\` within Ocaml strings/regexp you must use `\\` to escape from the string and regexp.
- If the beginning of a string could match multiple tokens, the longest match should be preferred, for example:
    - "let0" should not be lexed as `Tok_Let` followed by `Tok_Int 0`, but as `Tok_ID("let0")`, since it is an identifier.
    - "330dlet" should be tokenized as `[Tok_Int 330; Tok_ID "dlet"]`. Arbitrary amounts of whitespace also includes no whitespace.
    - "(-1)" should not be lexed as `[Tok_LParen; Tok_Sub; Tok_Int(1); Tok_LParen]` but as `Tok_Int(-1)`. (This is further explained below)

Most tokens only exist in one form (for example, the only way for `Tok_Concat` to appear in the program is as `^` and the only way for `Tok_Let` to appear in the program is as `let`). However, a few tokens have more complex rules. The regular expressions for these more complex rules are provided here:

- `Tok_Bool of bool`: The value will be set to `true` on the input string "true" and `false` on the input string "false".

- *Regular Expression*: `true|false`

- `Tok_Int of int`: Valid ints may be positive or negative and consist of 1 or more digits. Negative integers must be surrounded by parentheses (without extra whitespace) to differentiate from subtraction (examples below). You may find the functions `int_of_string` and `String.sub` useful in lexing this token type.
  - *Regular Expression*: `[0-9]+` OR `(-[0-9]+)`
  - *Examples of int parenthesization*:
    - `tokenize "x -1" = [Tok_ID "x"; Tok_Sub; Tok_Int 1]`
    - `tokenize "x (-1)" = [Tok_ID "x"; Tok_Int (-1)]`
- `Tok_String of string`: Valid string will always be surrounded by `""` and should accept any character except quotes within them (as well as nothing). You have to "sanitize" the matched string to remove surrounding escaped quotes.
  - *Regular Expression*: `\"[^\"]*\"`
  - *Examples*:
    - `tokenize "330" = [Tok_Int 330]`
    - `tokenize "\"330\"" = [Tok_String "330"]`
    - `tokenize "\"\"\"" (* InvalidInputException *)`
- `Tok_ID of string`: Valid IDs must start with a letter and can be followed by any number of letters or numbers. Note: Keywords may be substrings of IDs.
  - *Regular Expression*: `[a-zA-Z][a-zA-Z0-9]*`
  - *Valid examples*:
    - "a"
    - "ABC"
    - "a1b2c3DEF6"
    - "fun1"
    - "ifthenelse"

MicroCaml syntax with its corresponding token is shown below, excluding the four literal token types specified above.

Token Name	**Lexical Representation**
`Tok_LParen`	`(`
`Tok_RParen`	`)`
`Tok_LCurly`	`{`

```
Tok_RCurly          }

Tok_Dot             .

Tok_Equal           =

Tok_NotEqual        <>

Tok_Greater         >

Tok_Less            <

Tok_GreaterEqual    >=

Tok_LessEqual       <=

Tok_Or              ||

Tok_And             &&

Tok_Not             not

Tok_If              if

Tok_Then            then

Tok_Else            else

Tok_Add             +

Tok_Sub             -

Tok_Mult            *
```

`Tok_Div`	`/`
`Tok_Concat`	`^`
`Tok_Let`	`let`
`Tok_Def`	`def`
`Tok_In`	`in`
`Tok_Rec`	`rec`
`Tok_Fun`	`fun`
`Tok_Arrow`	`->`
`Tok_DoubleSemi`	`;;`
`Tok_Semi`	`;`

Notes:

- Your lexing code will feed the tokens into your parser, so a broken lexer can cause you to fail tests related to parsing.
- In the grammars given below, the syntax matching tokens (lexical representation) is used instead of the token name. For example, the grammars below will use `(` instead of `Tok_LParen`.

Part A2: Parsing MicroCaml Expressions

In this section, your task is to implement `parse_expr`, a function that takes a stream of tokens as input and outputs an abstract syntax tree (AST) representing the MicroCaml expression corresponding to the given tokens. All parser code should be placed in `parser.ml` following the signature specified in `parser.mli`.

First, we'll provide a brief overview of `parse_expr`, followed by the definition of the AST types it should return, and finally, the grammar it should parse.

parse_expr

- Type: `token list -> token list * expr`
- Description: This function takes a list of tokens and returns an AST representing the MicroCaml expression corresponding to the given tokens, along with any tokens left in the token list.
- Exceptions: It raises `InvalidInputException` if the input fails to parse, i.e., does not match the MicroCaml expression grammar.

Examples:

```
parse_expr [Tok_Int(1); Tok_Add; Tok_Int(2)] = ([], Binop (Add, (Int 1), (Int 2)))
parse_expr [Tok_Int(1)] = ([], (Int 1))
parse_expr [Tok_Let; Tok_ID("x"); Tok_Equal; Tok_Bool(true); Tok_In; Tok_ID("x")] = ([], Let ("x", false, (Bool true), ID "x"))
```

`parse_expr [Tok_DoubleSemi]` (raises `InvalidInputException`)

You'll likely find it useful to implement your parser using the `lookahead` and `match_tok` functions provided. More information about these functions can be found at the end of this README.

Here's the `expr` abstract syntax tree (AST) type, which `parse_expr` returns. Note that for now, you can disregard the `environment` and `Closure of environment * var * expr` parts as they are only relevant to Part (B):

```
type op = Add | Sub | Mult | Div | Concat | Greater | Less | GreaterEqual | LessEqual | Equal | NotEqual | Or | And

type var = string

type label = Lab of var

type expr =

 | Int of int
```

```
 | Bool of bool

 | String of string

 | Closure of environment * var * expr (* not used in P4A *)

 | ID of var

 | Fun of var * expr (* an anonymous function: var is the parameter and expr is the body *)

 | Not of expr

 | Binop of op * expr * expr

 | If of expr * expr * expr

 | App of expr * expr

 | Let of var * bool * expr * expr (* bool determines whether var is recursive *)

 | Record of (label * expr) list

 | Select of  label * expr

and environment = (var * expr ref) list
```

The CFG below describes the language of MicroCaml expressions. This CFG is right-recursive, so something like `1 + 2 + 3` will parse as `Add (Int 1, Add (Int 2, Int 3))`, essentially implying parentheses in the form `(1 + (2 + 3))`.) In the given CFG note that all non-terminals are capitalized, all syntax literals (terminals) are formatted `as non-italicized code` and will come in to the parser as tokens from your lexer. Variant token types (i.e. `Tok_Bool`, `Tok_Int`, `Tok_String` and `Tok_ID`) will be printed *`as italicized code`*.

- Expr -> LetExpr | IfExpr | FunctionExpr | OrExpr
- LetExpr -> `let` Recursion *`Tok_ID`* `=` Expr `in` Expr
  - Recursion -> `rec` | ε
- FunctionExpr -> `fun` *`Tok_ID`* `->` Expr
- IfExpr -> `if` Expr `then` Expr `else` Expr
- OrExpr -> AndExpr `||` OrExpr | AndExpr
- AndExpr -> EqualityExpr `&&` AndExpr | EqualityExpr
- EqualityExpr -> RelationalExpr EqualityOperator EqualityExpr | RelationalExpr
  - EqualityOperator -> `=` | `<>`
- RelationalExpr -> AdditiveExpr RelationalOperator RelationalExpr | AdditiveExpr

- ○ RelationalOperator -> `<` | `>` | `<=` | `>=`
- AdditiveExpr -> MultiplicativeExpr AdditiveOperator AdditiveExpr | MultiplicativeExpr
  - ○ AdditiveOperator -> `+` | `-`
- MultiplicativeExpr -> ConcatExpr MultiplicativeOperator MultiplicativeExpr | ConcatExpr
  - ○ MultiplicativeOperator -> `*` | `/`
- ConcatExpr -> UnaryExpr `^` ConcatExpr | UnaryExpr
- UnaryExpr -> `not` UnaryExpr | AppExpr
- AppExpr -> SelectExpr PrimaryExpr | SelectExpr
- SelectExpr -> PrimaryExpr `.` `Tok_ID` | PrimaryExpr
- PrimaryExpr -> *`Tok_Int`* | *`Tok_Bool`* | *`Tok_String`* | *`Tok_ID`* | `(` Expr `)`| RecordExpr
- RecordExpr -> `{` RecordBodyExpr `}`
- RecordBodyExpr -> *`Tok_ID`* `=` Expr `;` RecordBodyExpr | *`Tok_ID`* `=` Expr

To illustrate `parse_expr` in action, we show several examples of input and their output AST.

Example 1: Basic math

Input:

```
(1 + 2 + 3) / 3
```

Output (after lexing and parsing):

```
Binop (Div,

  Binop (Add, (Int 1), Binop (Add, (Int 2), (Int 3))),

  (Int 3))
```

In other words, if we run `parse_expr (tokenize "(1 + 2 + 3) / 3")` it will return the AST above.

Example 2: Records

Input:

```
{}
```

Output (after lexing and parsing):

Record []

Input:

{x=10; y=20}

Output (after lexing and parsing):

Record [(Lab "x", Int 10); (Lab "y", Int 20)]

Input:

{x=10; y=20}.x

Output (after lexing and parsing):

Select (Lab "x", Record [(Lab "x", Int 10); (Lab "y", Int 20)])

Input:

e.type

Output (after lexing and parsing):

Select (Lab "type", ID "e")

Example 3: `let` expressions

Input:

let x = 2 * 3 / 5 + 4 in x - 5

Output (after lexing and parsing):

```
Let ("x", false,
  Binop (Add,
    Binop (Mult, (Int 2), Binop (Div, (Int 3), (Int 5))),
    (Int 4)),
```

Binop (Sub, ID "x", (Int 5)))

Example 4: `if then ... else ...`

Input:

let x = 3 in if not true then x > 3 else x < 3

Output (after lexing and parsing):

Let ("x", false, (Int 3),

If (Not ((Bool true)), Binop (Greater, ID "x", (Int 3)),

Binop (Less, ID "x", (Int 3))))

Example 5: Anonymous functions

Input:

let rec f = fun x -> x ^ 1 in f 1

Output (after lexing and parsing):

Let ("f", true, Fun ("x", Binop (Concat, ID "x", (Int 1))),

App (ID "f", (Int 1)))

Remember, the parser's role doesn't extend to identifying type errors; that's the interpreter's responsibility in Part (B).

For instance, if the parser encounters the input "1 1", it should parse it as `App ((Int 1), (Int 1))`. However, when executed by the interpreter, this expression would trigger a type error at that point.
Example 5: Recursive Anonymous Functions

Observe how the AST for let expressions utilizes a boolean flag to distinguish recursive functions from non-recursive ones. When defining a recursive anonymous function using the syntax `let rec f = fun x -> ... in ...`, the identifier `f` will be bound to the function `fun x -> ...` during function evaluation. Handling this behavior, along with cases where `rec` is used without anonymous functions and attempted recursion without using `rec`, falls under the interpreter's jurisdiction.

For the time being, let's construct an infinite recursive loop for demonstration purposes:
Input:

```
let rec f = fun x -> f (x*x) in f 2
```

Output (after lexing and parsing):

```
Let ("f", true,

  Fun ("x", App (ID "f", Binop (Mult, ID "x", ID "x"))),

  App (ID "f", (Int 2))))
```

Example 6: Currying

We will ONLY be currying to create multivariable functions as well as passing multiple arguments to them. Here is an example:

Input:

```
let f = fun x -> fun y -> x + y in (f 1) 2
```

Output (after lexing and parsing):

```
Let ("f", false,

  Fun ("x", Fun ("y", Binop (Add, ID "x", ID "y"))),

  App (App (ID "f", (Int 1)), (Int 2)))
```

Part A3: Parsing Mutop Directives

In this section, your task is to implement `parse_mutop` in `parser.ml` according to the signature specified in `parser.mli`. This function processes a token list generated by lexing a string that represents a top-level MicroCaml directive (mutop), and returns an AST of OCaml type `mutop`. Your implementation of `parse_mutop` will leverage your existing `parse_expr` implementation, requiring minimal additional effort.

First, let's provide a brief overview of the function, followed by the definition of AST types it should return, and finally the grammar it should parse.

`parse_mutop`

- Type: `token list -> token list * mutop`
- Description: Takes a list of tokens and returns an AST representing the MicroCaml expression at the `mutop` level corresponding to the given tokens, along with any tokens left in the token list.
- Exceptions: Raise `InvalidInputException` if the input fails to parse i.e does not match the MicroCaml definition grammar.
- Examples:

parse_mutop [Tok_Def; Tok_ID("x"); Tok_Equal; Tok_Bool(true); Tok_DoubleSemi] = ([], Def ("x", (Bool true)))

parse_mutop [Tok_DoubleSemi] = ([], NoOp)

parse_mutop [Tok_Int(1); Tok_DoubleSemi] = ([], Expr ((Int 1))))

parse_mutop [Tok_Let; Tok_ID "x"; Tok_Equal; Tok_Bool true; Tok_In; Tok_ID "x"; Tok_DoubleSemi] =

- ([], Expr (Let ("x", false, (Bool true), ID "x")))

AST and Grammar of `parse_mutop`

Below is the AST type `mutop`, which is returned by `parse_mutop`, followed by the CFG that it parses for MicroCaml expressions at the `mutop` level. This CFG is similar (and similarly formatted) to the CFG of `parse_expr` and relies on its implementation of Expr.

type mutop =

| Def of var * expr

| Expr of expr

| NoOp

The CFG is as follows:

- Mutop -> DefMutop | ExprMutop | `;;`
- DefMutop -> `def` *`Tok_ID`* `=` Expr `;;`
- ExprMutop -> Expr `;;`

Notice how a valid input for the `parse_mutop` must always terminate with `Tok_DoubleSemi` and input of just `Tok_DoubleSemi` to the parser is considered valid as per the AST.

For this part, we created a new keyword `def` to refer to top-level MicroCaml expressions to differentiate local `let`. In essence, `def` is similar to top-level (global) `let` expressions in normal (OCaml) `utop`. This means `def` will create global definitions for variables while running `mutop`. Another key difference between `def` and the `let` expressions defined in Part A2 is that `def` should be *implicitly recursive*. (Note that `def rec x = ...;;` is not valid as per the given AST---basically the `rec` is implicit).

Here are some example mutop directives. Note that `parse_mutop` should return a tuple of (updated token list, parsed AST), but in these examples we omit the updated token list since it should always just be an empty list.

Example 1: Global definition

Input:

```
def x = let a = 3 in if a <> 3 then 0 else 1;;
```

Output (after lexing and parsing):

```
Def ("x",
  Let ("a", false, (Int 3),
    If (Binop (NotEqual, ID "a", (Int 3)), (Int 0), (Int 1))))
```

Example 2: Implicit recursion on `f`

Input:

```
def f = fun x -> if x > 0 then f (x-1) else "done";;
```

Output (after lexing and parsing):

```
Def ("f",
  Fun ("x",
   If (Binop (Greater, ID "x", (Int 0)),
    App (ID "f", Binop (Sub, ID "x", (Int 1))),
    (String "done"))))
```

Example 3: Expression

Input:

```
(fun x -> "(" ^ x ^ ")") "parenthesis";;
```

Output (after lexing and parsing):

```
Expr (
  App (Fun ("x",
    Binop (Concat, (String "("),
      Binop (Concat, ID "x", (String ")")))),
    (String "parenthesis")))
```

Functions Provided

In the `parser.ml` file, you'll find some helper functions that can assist you in implementing both parsers. While you're not obligated to use these functions, it's highly recommended as they can streamline your implementation process.

`match_token`

- Type: `token list -> token -> token list`
- Description: Takes the list of tokens and a single token as arguments, and returns a new token list with the first token removed IF the first token matches the second argument.
- Exceptions: Raise `InvalidInputException` if the first token does not match the second argument to the function.

`match_many`

- Type: `token list -> token list -> token list`
- Description: An extension of `match_token` that matches a sequence of tokens given as the second token list and returns a new token list that matches each token in the order in which they appear in the sequence. For example, `match_many toks [Tok_Let]` is equivalent to `match_token toks Tok_Let`.
- Exceptions: Raise `InvalidInputException` if the tokens do not match.

`lookahead`

- Type: `token list -> token option`
- Description: Returns the top token in the list of tokens as an option, returning `None` if the token list is empty. In constructing your parser, the lack of lookahead token (None) is fine for the epsilon case.

`lookahead_many`

- Type: `token list -> int -> token option`
- Description: An extension of `lookahead` that returns a token at the nth index in the list of tokens as an option, returning `None` if the token list is empty at the given index or the index is negative. For example, `lookahead_many toks 0` is equivalent to `lookahead toks`.

Part (B): MicroCaml Interpreter

In this segment, your objective is to develop a Micro oCaml interpreter by implementing two functions: `eval_expr` and `eval_mutop`.

To accomplish this task, you'll need to incorporate a bit of Imperative OCaml, particularly references. While the usage of Imperative OCaml is minimal, it plays a crucial role in this part of the project. Further details are provided below.

Part B1: Evaluating Expressions

`eval_expr`

- Type: `environment -> expr -> expr`
- Description: This function takes in an environment `env` and an expression `e`, and returns the result of `e` evaluated within the environment `env`, which just so happens to be another expression. The returned expression can be thought of as a "reduction" of the input expression `e`.

The environment `env` is a `(var * expr ref) list`, where `var` refers to a variable name (a string), and the `expr ref` refers to its corresponding expression in the `environment`; it's a `ref` because the expression could change, due to implementing recursion, as discussed for `Let` below. Elements earlier in the list shadow elements later in the list.

Exceptions

Here's a list of all the possible error cases and exceptions (can also be found in `types.ml`):

exception TypeError of string

exception DeclareError of string

exception SelectError of string

exception DivByZeroError

- A `TypeError` happens when an operation receives an argument of the wrong type
- A `DeclareError` happens when an ID is seen that has not been declared
- A `SelectError` happens when trying to select a nonexistent label from a record
- A `DivByZeroError` happens on attempted division by zero

Note that we do not enforce what messages you use when raising `TypeError`, `DeclareError`, or `SelectError` exceptions. That's up to you.

Evaluation

For consistent error matching, it's essential to evaluate subexpressions from left to right.
Now we describe what your interpreter should do for each kind of `expr`:

```
type expr =

  | Int of int

  | Bool of bool

  | String of string

  | ID of var

  | Fun of var * expr

  | Not of expr

  | Binop of op * expr * expr

  | If of expr * expr * expr

  | App of expr * expr

  | Let of var * bool * expr * expr

  | Closure of environment * var * expr

  | Record of (label * expr) list

  | Select of label * expr
```

ID

An identifier evaluates to whatever expression it is mapped to by the environment. Should raise a `DeclareError` if the identifier has no binding.

```
eval_expr [("x", ref (Int 1))] (ID "x") = Int 1

eval_expr [] (ID "x") (* DeclareError "Unbound variable x" *)
```

See the discussion of `Let` below for advice about managing environments.

Not

The unary `not` operator operates only on booleans and produces a `Bool` containing the negated boolean value of the contained expression. If the expression in the `Not` is not a boolean (or does not evaluate to a boolean), a `TypeError` should be raised.

eval_expr [("x", ref (Bool true))] (Not (ID "x")) = Bool false

eval_expr [("x", ref (Bool true))] (Not (Not (ID "x"))) = Bool true

eval_expr [] (Not (Int 1)) (* TypeError "Expected type bool" *)

Binary Operators

There are five categories of binary operators in MicroCaml:

- Operators performing integer arithmetic
- Operators performing integer ordering comparisons
- Operator performing string concatenation
- Operator performing equality (and inequality) comparisons
- Operators implementing boolean logic.

Add, Sub, Mult, and Div

Arithmetic operators work on integers; if either argument evaluates to a non-`Int`, a `TypeError` should be raised. An attempt to divide by zero should raise a `DivByZeroError` exceptio.

eval_expr [] (Binop (Add, (Int 1), (Int 2))) = Int 3

eval_expr [] (Binop (Add, (Int 1), (Bool false))) (* TypeError "Expected type int" *)

eval_expr [] (Binop (Div, (Int 1), (Int 0))) (* DivByZeroError *)

Greater, Less, GreaterEqual, and LessEqual

These relational operators operate only on integers and produce a `Bool` containing the result of the operation. If either argument evaluates to a non-`Int`, a `TypeError` should be raised.

eval_expr [] (Binop(Greater, (Int 1), (Int 2))) = Bool false

eval_expr [] (Binop(LessEqual, (Bool false), (Bool true))) (* TypeError "Expected type int" *)

Concat

This operation returns the result of concatenating two strings; if either argument evaluates to a non-`String`, a `TypeError` should be raised.

eval_expr [] (Binop (Concat, (Int 1), (Int 2))) (* TypeError "Expected type string" *)

eval_expr [] (Binop (Concat, (String "hello "), (String "ocaml"))) = String "hello ocaml"

Equal and NotEqual

The equality operators require both arguments to be of the same type. The operators produce a `Bool` containing the result of the operation. If the two arguments to these operators do not evaluate to the same type (e.g., one boolean and one integer), a `TypeError` should be raised. Moreover, we *cannot compare two closures for equality* -- to do so risks an infinite loop because of the way recursive functions are implemented; trying to compare them also raises `TypeError` (OCaml does the same thing in its implementation, BTW).

eval_expr [] (Binop(NotEqual, (Int 1), (Int 2))) = Bool true

eval_expr [] (Binop(Equal, (Bool false), (Bool true))) = Bool false

eval_expr [] (Binop(Equal, (String "hi"), (String "hi"))) = Bool true

eval_expr [] (Binop(NotEqual, (Int 1), (Bool false))) (* TypeError "Cannot compare types" *)

Or and And

These logical operations operate only on booleans and produce a `Bool` result. If either argument evaluates to a non-`Bool`, a `TypeError` should be raised.

eval_expr [] (Binop(Or, (Int 1), (Int 2))) (* TypeError "Expected type bool" *)

eval_expr [] (Binop(Or, (Bool false), (Bool true))) = Bool true

If

The `If` expression consists of three subexpressions - a guard, the true branch, and the false branch. The guard expression must evaluate to a `Bool` - if it does not, a `TypeError` should be raised. If it evaluates to `Bool true`, the true branch should be evaluated; else the false branch should be.

eval_expr [] (If (Binop (Equal, (Int 3), (Int 3)), (Bool true), (Bool false))) = Bool true eval_expr [] (If (Binop (Equal, (Int 3), (Int 2)), (Int 5), (Bool false))) = Bool false

Notes:

- Only one branch should be evaluated, not both.
- The true and false branches could evaluate to expressions having different types. This is an effect of MicroCaml being dynamically typed.

Let

The `Let` consists of four components - an ID's name `var` (which is a string); a boolean indicating whether or not the bound variable is referenced in its own definition (i.e., whether it's *recursive*); the *initialization expression*; and the *body expression*.

Non-recursive bindings

For a non-recursive `Let`, we first evaluate the initialization expression, which produces an expression *ex* or raises an error. If the former, we then return the result of evaluating the body expression in an environment extended with a mapping from the `Let`'s ID variable to *ex*. (Evaluating the body might cause an exception to be raised.)

eval_expr [] (Let ("x", false,

  Binop (Add, Binop (Mult, (Int 2),

    Binop (Div, (Int 3), (Int 5))), (Int 4)),

  Binop (Sub, ID "x", (Int 5)))) = Int (-1)

Recursive bindings

For a recursive `Let`, we evaluate the initialization expression in an environment extended with a mapping from the ID we are binding to a temporary placeholder; this way, the initialization expression is permitted to refer to itself, the ID being bound. Then, we *update* that placeholder to *v*, the result, before evaluating the body.

The AST given in this example corresponds to the MicroCaml program `let rec f = fun x -> if x = 0 then x else (x + (f (x-1))) in f 8`:

eval_expr [] (Let ("f", true,

  Fun ("x",

```
If (Binop (Equal, ID "x", (Int 0)), ID "x",
  Binop (Add, ID "x",
    App (ID "f", Binop (Sub, ID "x", (Int 1)))))),
App (ID "f", (Int 8)))) = Int 36
```

Environments

Being able to modify the placeholder is made possibly by using references; this is why the type `environment` given in `types.ml` is `(var * expr ref) list` and not `(var * expr) list`. To make it easy to work with this kind of environment, we recommend you use the functions given at the top of `eval.ml`:

- `extend env x e` produces an environment that extends `env` with a mapping from `x` to `e`
- `lookup env x` returns `e` if `x` maps to `e` in `env`; if there are multiple mappings, it chooses the most recent.
- `extend_tmp x` produces an environment that extends `env` with a mapping from `x` to a temporary placeholder.
- `update env x e` produces an environment that updates `env` in place, modifying its most recent mapping for `x` to be `e` instead (removing the placeholder).

Fun

The `Fun` is used for anonymous functions, which consist of two components - a parameter, which is a string as an ID's name, and a body, which is an expression. A `Fun` evaluates to a `Closure` that captures the current environment, so as to implement lexical (aka static) scoping.

```
eval_expr [("x", ref (Bool true))] (Fun ("y", Binop (And, ID "x", ID "y")))
 = Closure ([("x", ref (Bool true))], "y", Binop (And, ID "x", ID "y"))
eval_expr [] (Fun ("x", Fun ("y", Binop (And, ID "x", ID "y"))))
      = Closure ([], "x", Fun ("y", Binop (And, ID "x", ID "y")))
```

App

`App` has two subexpressions. We evaluate the first to a `Closure(A,x,e)` (otherwise, a `TypeError` should be raised) and the second to a expression *v*. Then we evaluate `e` (the closure's body) in environment `A` (the closure's environment), returning the result.

eval_expr [] (App ((Int 1), (Int 1))) (* TypeError "Not a function" *)

eval_expr [] (Let ("f", false, Fun ("x", Fun ("y", Binop (Add, ID "x", ID "y"))),

  App (App (ID "f", (Int 1)), (Int 2)))) = Int 3

The AST in the second example is equivalent to the MicroCaml expression `let f = fun x -> fun y -> x + y in (f 1) 2`.

Record and Select

A `Record` consists of a list of fields, or more specifically a list of `(label * expr)` tuples:

eval_expr [] (Record [(Lab "x", Int 10); (Lab "y", Int 20)])

  = Record [(Lab "x", Int 10); (Lab "y", Int 20)]

A `Select` consists of a `label` and an `expr`. We first evaluate the `expr` to a `Record` type (otherwise a `TypeError` should be raised). Then we try to look up the corresponding `label` within the `Record`. If the `label` exists, we return the evaluated `expr` that it's tied to. Otherwise, we raise a `SelectError`.

eval_expr [] (Select (Lab "x", (Record [(Lab "x", Int 10); (Lab "y", Int 20)])))

  = Int 10

eval_expr [] (Select (Lab "x", (Bool false)))

  (* TypeError "Not a record" *)

eval_expr [] (Select (Lab "z", (Record [(Lab "x", Int 10); (Lab "y", Int 20)])))

  (* SelectError "Label not found" *)

Part B2: Evaluating Mutop Directive

`eval_mutop`

- Type: `environment -> mutop -> environment * (expr option)`

- Description: This function evaluates the given `mutop` directive in the given `environment`, returning an updated environment with an optional `expr` as the result.

There are three kinds of `mutop` directive (as defined in types.ml):

```
type mutop =

  | Def of var * expr

  | Expr of expr

  | NoOp
```

Def

For a `Def`, we evaluate its `expr` in the given environment, but with a placeholder set for `var` (see the discussion of recursive `Let`, above, for more about environment placeholders), producing expression *ex*. We then update the binding for `var` to be *ex* and return the extended environment, along with the expression itself.

```
eval_mutop [] (Def ("x", (Bool(true)))) =  ([("x", {contents = Bool true})], Some (Bool true))

eval_mutop [] Def ("f",

  Fun ("y",

    If (Binop (Equal, ID "y", (Int 0)), (Int 1),

    App (ID "f", Binop (Sub, ID "y", (Int 1)))))) =

([("f",

  {contents =

    Closure (<cycle>, "y",

      If (Binop (Equal, ID "y", (Int 0)), (Int 1),

        App (ID "f", Binop (Sub, ID "y", (Int 1)))))})],

  Some

    (Closure ([("f", {contents = <cycle>})], "y",

      If (Binop (Equal, ID "y", (Int 0)), (Int 1),
```

App (ID "f", Binop (Sub, ID "y", (Int 1)))))))

Expr

For a `Expr`, we should evaluate the expression in the given environment, and return that environment and the resulting expression.

eval_mutop [] (Expr (App (Fun ("x",

Binop (Concat, (String "("),

Binop (Concat, ID "x", (String ")")))),

(String "parenthesis")))) = ([], Some (String "(parenthesis)"))

NoOp

The `NoOp` should return the original environment and no expression (`None`).

eval_mutop [] NoOp = ([], None)