Project Description
MiniOCaml

To submit the project, upload a `.ml` file on your personal status page in the CMS until January 4th 23:59. You may upload as often as you like beforehand. The latest submission before the deadline is graded.

The project is conducted individually. Team work is not permitted.

Tasks

Complete the MiniOCaml Interpreter[1] that we developed in the last two lectures. The MiniOCaml interpreter misses support for 'let rec', lambda, and function application. These have to be added to all phases.

For the lexer and parse, use the following concrete syntax:

- let rec: `let rec` f (x : t_1) : t_2 = e_1 `in` e_2
  (e.g. `let rec f (x : int) : int = if x ≤ 1 then x else x * f (x − 1) in f 5`)
- lambda: `fun` (x : t) → e
  (e.g. `fun (x : int) → x + 1`)
- application: e_1 e_2
  (e.g. `(fun (x:int) → x + 1) 2`)

Your development must provide the functions:

- `lex : string → token list`
- `parse_expr : token list → exp * token list`
- `check_ty : tenv → exp → ty option`
- `eval : venv → exp → va`

Following the naming schema, name the tokens with upper letters of their concrete syntax counterpart. Name your constructors for the AST `Lam`, `App`, and `Letrec`.

Exercise EP.1 *(Lexer 3 bonus points)*

(a) Extend the provided lexer for proper number literals.

(b) Add all missing tokens in the token type and their implementation in the lexing function.

Note that there is a typo in line 126. The line should be `id_classify (String.sub s i (j − i)) j l .`

[1] `https://cdltools.cs.uni-saarland.de/soocaml/share/b95038d9e6a62bb87e7b1c7c8dfd56f576e151ae0aea55245c5f3de24cb6c7a7`

Exercise EP.2 *(Parser 6 bonus points)*

(a) Add `let rec`, lambda, and function application to the abstract syntax.

(b) In order to parse `let rec` and lambda, you need support for parsing and representing types. Refer to the lecture notes for the abstract syntax of types.

(c) Write a parser for types using the same principles we have used for expression parsing.
Hint: You can use the precedence parser to parse function types by properly

customizing it. You don't have to copy & paste it!

(d) Extend the parser for these constructs.
Hint: Function application is also just a binary operator. You can use the infrastructure of the precedence parser to add support for it in a clever way.

(e) Add support for expressions in parentheses (tokens `LP`, `RP`).

(f) Make sure your parser also raises an error if the input cannot be fully consumed. Right now the parser will parse `1 + 2 let` to the binary expression `Binary(Add, Icon 1, Icon 2)` but will leave a remaining `let` token behind.
Hint: Add and End-of-input token and make sure it is properly parsed.

Exercise EP.3 *(Semantics 6 bonus points)*

(a) Add support for `let rec`, lambda, and function application in the type checker.

(b) Add support for `let rec`, lambda, and function application in the evaluator.