BU CS320 Theory Assignment 3

Parser Combinators

1. Basic Combinators

We encourage you to first review the parsers.ml file that is posted. This file contains some parsers that were covered in lecture and contains some parsers that are covered in the pre lecture videos. You must review these first before attempting this question.
Describe the purpose (in plain English using few sentences) of the following combinators that are present in parsers.ml

- pure
  Pure is a return function. pure is the simplest function that returns a value.
  pure is used in conjunction with bind
  combinators are parsers, some of them, the type of pure is prime a' to a' parser, it returns a parser.
  -gives back parser in parser form (changes from a' parser to b' parser in bind)
- >>=
  binds until you get to a parser (or a pure, which returns a parser)
  breaks int of 872 to [8;7;2]
  (bind is a function that returns a function)
  Binds to result of first parser, takes parser p and func q to return another parser
- <|>
  if one parser fails, run the other.
  two parsers p1 and p2. If p1 contains something or doesn't 'fail', then it just returns p1 applied to the list. Otherwise, it returns p2 applied to the list.

- fail

combinators are a higher order functions, functions that create functions
returns None, it is incompatible/when parser reaches a failure case (an example would be this is used in bind),

- read
  going to make sure there is a single character

2. Derived Combinators

We encourage you to first review the parsers.ml file that is posted. This file contains some parsers that were covered in lecture and contains some parsers that are covered in the pre lecture videos. You must review these first before attempting this question.
Describe the purpose (in plain english using few sentences) of the following combinators that are present in parsers.ml

- many
  many runs a parser many times to put it into a list. Many keeps running with zero or one instances

- many1

  many1 runs a parser many times to put it into a list but returns None when zero instances are found.

- char
  if it has char is true then return char parser, otherwise false, then return None. Calls satisfy to receive char parser from char.

- literal
  explodes string into a list of string, goes into loop and char loop parser
  check that every letter of the string is represented in the beginning of the input, input starts with that string.

  like in bool_parser
  “12345true” = FAIL
  “[t;r;u;e;a;b;c]” = success ((), [a;b;c])

3. Combinator Usage

In the provided parsers.ml file, the type of a parser is given as follows:

type 'a parser = char list -> ('a * char list) option

Now, instead assume that the type of the parser is:

type 'a parser = char list -> [('a * char list)]

Rewrite the following combinators:

- <|>

  ```
  let disj (p1 : 'a parser) (p2 : 'a parser) : 'a parser =
    fun ls ->
    match p1 ls with
    |  (x, ls)::_  -> [(x, ls)]
    | [] -> p2 ls

  let (<|>) = disj
  ```

- >>=

  ```
  let bind (p : 'a parser) (q : 'a -> 'b parser) : 'b parser =
    fun ls ->
    match p ls with
    |  (a, ls)::_ -> q a ls
    | [] -> []

  let (>>=) = bind
  let (let*) = bind
  ```

- return

  ```
  let return (x : 'a) : 'a parser =
    fun ls -> [ (x, ls) ]
  ```

- fail

  ```
  let fail : 'a parser = fun ls -> []
  ```

4. Formal Grammars

- Consider the following grammar.

```
<bool>      ::= true | false
<bool_tree> ::= <bool> | ( <bool_tree> ^ <bool_tree> )
```

Write a leftmost derivation for the following sentences:

```
( true ^ ( true ^ false ) )
<bool_tree> ::= (<bool_tree> ^ <bool_tree>)
            ::= (<bool> ^ <bool_tree>)
            ::= (true ^ (<bool_tree> ^ <bool_tree>))
            ::= (true ^(<bool> ^ <bool_tree>))
            ::= (true ^ (true ^ <bool_tree>))
           ::= (true ^ (true ^ <bool>))
            ::= (true ^ (true ^ false))
(<bool> (true ^ false))
```

Write a rightmost derivation for the following sentences:

```
( ( true ^ true ) ^ ( false ^ false ) )
```

((true ^ true) ^ (false ^ false))
<bool_tree> ::= (<bool_tree> ^ <bool_tree>)
::= (<bool_tree> ^ (<bool_tree> ^ <bool_tree>))
::= (<bool_tree> ^ (<bool_tree> ^ <bool>))
::= (<bool_tree> ^ (<bool_tree> ^ false))
::= (<bool_tree> ^ (<bool> ^ false))
::= (<bool_tree> ^ (false ^ false))
::= ((<bool_tree> ^ <bool_tree>) ^ (false ^ false))
::= ((<bool_tree> ^ <bool>) ^ (false ^ false))
::= ((<bool_tree> ^ true) ^ (false ^ false))
::= ((<bool> ^ true) ^ (false ^ false))
::= ((true ^ true) ^ (false ^ false))

- Consider the following grammar.

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<nat>   ::= <digit> | <digit><nat>
<opr>   ::= + | - | * | /
<expr>  ::= <nat> | ( <opr> <expr> <expr> )
```

Write a leftmost derivation for the following sentences:

```
( * ( + 111 23 ) 9 )
```

```
<expr>  ::= ( <opr> <expr> <expr> )
           ( * <expr> <expr> )
           ( * ( <opr> <expr> <expr> ) <expr> )
           ( * ( + <expr> <expr> ) <expr> )
           ( * ( + <nat> <expr> ) <expr> )
           ( * ( + <digit><nat> <expr> ) <expr> )
           ( * ( + 1<nat> <expr> ) <expr> )
           ( * ( + 1<digit><nat> <expr> ) <expr> )
           ( * ( + 11<nat> <expr> ) <expr> )
           ( * ( + 11<digit> <expr> ) <expr> )
           ( * ( + 111 <expr> ) <expr> )
           ( * ( + 111 <nat> ) <expr> )
           ( * ( + 111 <digit><nat> ) <expr> )
           ( * ( + 111 2<nat> ) <expr> )
           ( * ( + 111 2<digit> ) <expr> )
           ( * ( + 111 23 ) <expr> )
           ( * ( + 111 23 ) <nat> )
           ( * ( + 111 23 ) <digit> )
           ( * ( + 111 23 ) 9 )
```

Write a rightmost derivation for the following sentences:

```
( / 10 ( - 11 11 ) )
<expr>  ::= ( <opr> <expr> <expr> )
           ( <opr> <expr> ( <opr> <expr> <expr> ) )
           ( <opr> <expr> ( <opr> <expr> <nat> ) )
           ( <opr> <expr> ( <opr> <expr> <digit><nat> ) )
           ( <opr> <expr> ( <opr> <expr> <digit><digit> ) )
           ( <opr> <expr> ( <opr> <expr> <digit>1 ) )
           ( <opr> <expr> ( <opr> <expr> 11 ) )
           ( <opr> <expr> ( <opr> <nat> 11 ) )
           ( <opr> <expr> ( <opr> <digit><nat> 11 ) )
           ( <opr> <expr> ( <opr> <digit><digit> 11 ) )
           ( <opr> <expr> ( <opr> <digit>1 11 ) )
           ( <opr> <expr> ( <opr> 11 11 ) )
           ( <opr> <expr> ( - 11 11 ) )
           ( <opr> <nat> ( - 11 11 ) )
           ( <opr> <digit><nat> ( - 11 11 ) )
           ( <opr> <digit><digit> ( - 11 11 ) )
           ( <opr> <digit>0 ( - 11 11 ) )
           ( <opr> 10 ( - 11 11 ) )
           ( / 10 ( - 11 11 ) )
```