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
  return the parsed values
  (Parse x, then return it)

- >>=
  compose parser functions and return optional results
  (if p is Some parsed values, then do q (parse again by another different type parser), then return a new type parsed value, else (if None), return None)

- <|>
  disjunct all the values to either one parser or the other and return optional results
  (If p1 is Some parsed values, then return p1, else (if None), do p2 (parse by another same type parser))

- fail
  return optional result None, represents the parsing failed
  (return None)

- read
  read the first char of the char list and return optional results
  (if a char list, then return Some parsed value (form as Some (char * char list)), else, return None)

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
  run the parser zero or more times

- many1
  run the parser one or more times

- char
  run the parser if satisfies that the type of input is char

- literal
  check if the input string literally equal to the char list

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) list

Rewrite the following combinators:

- <|>
- >>=
- pure
- fail

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

let disj (p1 : 'a parser) (p2 : 'a parser) : 'a parser =
  fun ls ->
  match p1 ls with
  | (x, ls)::t -> [(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)::t -> q a ls
  | [] -> []

let (>>=) = bind

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

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 ) )

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

Write a rightmost derivation for the following sentences:

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

<sentence> ::= <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 )

<sentence> ::= <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 ) )

<sentence> ::= <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 ) )
```