Hype For Types Homework 6: Monadic Parser Combinators

Email solutions to mmcquaid@andrew.cmu.edu

```
type 'a parser = char list -> ('a * char list) option
infix >>= ++ %
fun return x = fn i => SOME (x,i)
fun p >>= f = fn i =>
    case p i of
        SOME (x,r) => f x r
    |   n => n
fun p ++ q = fn i =>
    case p i of
        NONE => q i
    |   r => r
fun p % s = p (String.explode s)
```

As is usual with Hype For Types homework, please don't spend more than an hour on this if you don't want to. Incorrect or incomplete answers will be accepted if a reasonable attempt is present. Recommended handin format is an SML file with the answer to the first question in a commment, but you can write the whole thing in Latex if you wish. A couple basic parser functions are listed above for your convenience. For more, see the notes (listed on the website).

1. Briefly describe the function and purpose of `return`, `>>=` and `++` for the type `'a parser`.

2. Using any of the combinators in the notes, write the following combinator:

   `pair : 'a parser * 'b parser -> ('a * 'b) parser`

   which takes two parsers `p`,`q` as input, and produces a parser which first runs `p`, then `q`, and then returns their results as tuple.

   For example:

   `pair (char #"a",char #"b") % "abc"` ⟹ `SOME ((#"a",#"b"),[#"c"])`

3. Using any of the combinators in the notes, write the following combinator:

   `map : ('a -> 'b) -> 'a parser -> 'b parser`

   which maps a function over the result of a parser.

   For example:

   `map (Int.toString o (fn x => x+1)) int % "149"` ⟹ `SOME ("150",[])`

4. Using only `pair` and `map`, implement the following combinator:

   `liftA2 : ('a * 'b -> 'c) -> ('a parser * 'b parser) -> 'c parser`

   which lifts a binary function into the domain of parsers.

   For example:

   `liftA2 op@ (many (string "a"), many (string "x")) % "aaxxx"` ⟹ `SOME (["a","a","x","x","x"],[])`

5. Using any of the combinators in the notes, write the following combinator:

   `prefix : ('a -> 'a) parser -> 'a parser -> 'a parser`

   which takes an operator parser and an operand parser, and produces a parser which parses zero or more instances of a prefixed operator, followed by one instance of an operand, and returns the result of applying all instances of the operator to the operand.

   For example:

   `prefix (string "s" >> return (fn x => x+1)) (string "z" >> return 0) % "sssz"` $\Longrightarrow$ `SOME (3,[])`

6. We extend our untyped λ-calculus term type from the notes to include *let* expressions:

```
datatype term = VAR of string
              | LAM of string * term
              | AP of term * term
              | LET of (string * term) * term
```

   We extend our concrete syntax to include terms of the form $let\, VARIABLE = TERM\ in\, TERM$. The scope of a *let* expression extends as far the right of the *in* as possible, and can be ended with parentheses. For example, we might write $let\ y = \lambda x.x\ in\ y\ y$, which would be represented in abstract syntax as `LET ((VAR "y",LAM("x",VAR "x")),AP(VAR "y",VAR "y"))`. Extend the λ-calculus parser at the end of the notes to handle *let* expressions.

7. **OPTIONAL:** Consider the following datatype representing regular expressions, restricted to not include the Kleene star (taken from 15-150):

```
datatype regex = Zero
               | One
               | Char of char
               | Times of regex * regex
               | Plus of regex * regex
```

   Write a parser `regexP : regex parser`, which transforms strings representing regular expressions into values of type `regex`. Your parser must support parentheses. We use concatenation to represent `Times` and `+` to represent `Plus`. So "$(ab) + c$" should be parsed to

   `Plus (Times (Char #"a",Char #"b"), Char #"c").`

   Hint: You'll definitely want to use `chainl1`, or just `chainl` if you're feeling clever.

8. **OPTIONAL:** We extend our `regex` datatype to include the Kleene star:

```
datatype regex = ...
               | Star of regex
```

   Extend your parser to handle regular expression strings containing the postfix Kleene star. For example, "$(a^*b)^{**}c^*$" should be parsed to

   `Times (Star (Star (Times (Star (Char #"a"),Char #"b"))),Star (Char #"c"))`

   Hint: You'll probably want to write a `postfix` combinator, which functions dually to the `prefix` combinator you've already written.