# 6

# USER INTERACTION

## 6.1 MONADIC PARSER COMBINATORS

A common approach to building parsers in functional programming is to model parsers as functions. Higher-order functions on parsers act as *combinators*, which are used to implement complex parsers in a modular way from a set of primitive ones. In this setting, parsers exhibit a monad algebraic structure, which can be used to simplify the combination of parsers. A technical report on **monadic parser combinators** can be found on [HM96].

The use of monads for parsing was discussed for the first time in [Wad85], and later in [Wad90] and [HM98]. The parser type is defined as a function taking an input `String` and returning a list of pairs, representing a successful parse each. The first component of the pair is the parsed value and the second component is the remaining input. The Haskell code for this definition is

```
-- A parser takes a string an returns a list of possible parsings with
-- their remaining string.
newtype Parser a = Parser (String -> [(a,String)])
parse :: Parser a -> String -> [(a,String)]
parse (Parser p) = p
-- A parser can be composed monadically, the composed parser (p >>= q)
-- applies q to every possible parsing of p. A trivial one is defined.
instance Monad Parser where
  return x = Parser (\s -> [(x,s)]) -- Trivial parser, directly returns x.
  p >>= q  = Parser (\s -> concat [parse (q x) s' | (x,s') <- parse p s ])
```

where the monadic structure is defined by `bind` and `return`. Given a value, the `return` function creates a parser that consumes no input and simply returns the given value. The »= function acts as a sequencing operator for parsers. It takes two parsers and applies the second one over the remaining inputs of the first one, using the parsed values on the first parsing as arguments.

An example of primitive **parser** is the `item` parser, which consumes a character from a non-empty string. It is written in Haskell code using pattern matching on the string as

```
item :: Parser Char
item = Parser (\s -> case s of "" -> []; (c:s') -> [(c,s')])
```

and an example of **parser combinator** is the `many` function, which creates a parser that allows one or more applications of the given parser

```
many :: Paser a -> Parser [a]
many p = do
  a  <- p
  as <- many p
  return (a:as)
```

in this example `many item` would be a parser consuming all characters from the input string.

## 6.2 PARSEC

**Parsec** is a monadic parser combinator Haskell library described in [Lei01]. We have chosen to use it due to its simplicity and extensive documentation. As we expect to use it to parse user live input, which will tend to be short, performance is not a critical concern. A high-performace library supporting incremental parsing, such as **Attoparsec** [O'S16], would be suitable otherwise.

## 6.3 VERBOSE MODE

As we explained previously on the evaluation section, the simplification can be analyzed step-by-step. The interpreter allows us to see the complete evaluation when the `verbose` mode is activated. To activate it, we can execute `:verbose on` in the interpreter.

The difference can be seen on the following example, which shows the execution of $1 + 2$, first without intermediate results, and later, showing every intermediate step.

```
mikro> plus 1 2
λa.λb.(a (a (a b))) ⇒ 3

mikro> :verbose on
verbose: on
mikro> plus 1 2
```

```
((plus 1) 2)
((λλλλ((4 2) ((3 2) 1)) λλ(2 1)) λλ(2 (2 1)))
(λλλ((λλ(2 1) 2) ((3 2) 1)) λλ(2 (2 1)))
λλ((λλ(2 1) 2) ((λλ(2 (2 1)) 2) 1))
λλ(λ(3 1) (λ(3 (3 1)) 1))
λλ(2 (λ(3 (3 1)) 1))
λλ(2 (2 (2 1)))

λa.λb.(a (a (a b))) ⇒ 3
```

The interpreter output can be colored to show specifically where it is performing reductions. It is activated by default, but can be deactivated by executing `:color off`. The following code implements *verbose mode* in both cases.

```haskell
-- | Shows an expression, coloring the next reduction if necessary
showReduction :: Exp -> String
showReduction (Lambda e)         = "λ" ++ showReduction e
showReduction (App (Lambda f) x) = betaColor (App (Lambda f) x)
showReduction (Var e)            = show e
showReduction (App rs x)         =
  "(" ++ showReduction rs ++ " " ++ showReduction x ++ ")"
showReduction e                  = show e
```

## 6.4 SKI MODE

Every $\lambda$-term can be written in terms of SKI combinators. SKI combinator expressions can be defined as a binary tree having S, K, and I as possible leafs.

```haskell
data Ski = S | K | I | Comb Ski Ski | Cte String
```

The SKI-abstraction and bracket abstraction algorithms are implemented on Mikrokosmos, and they can be used by activating the *ski mode* with `:ski on`. When this mode is activated, every result is written in terms of SKI combinators.

```
mikro> 2
λa.λb.(a (a b)) ⇒ S(S(KS)K)I ⇒ 2
mikro> and
λa.λb.((a b) a) ⇒ SSK ⇒ and
```

The code implementing these algorithms follows directly from the theoretical version in [HSo8].

```haskell
-- | Bracket abstraction of a SKI term, as defined in Hindley-Seldin
-- (2.18).
```

```haskell
bracketabs :: String -> Ski -> Ski
bracketabs x (Cte y) = if x == y then I else Comb K (Cte y)
bracketabs x (Comb u (Cte y))
  | freein x u && x == y = u
  | freein x u           = Comb K (Comb u (Cte y))
  | otherwise            = Comb (Comb S (bracketabs x u)) (bracketabs x (Cte y))
bracketabs x (Comb u v)
  | freein x (Comb u v)  = Comb K (Comb u v)
  | otherwise            = Comb (Comb S (bracketabs x u)) (bracketabs x v)
bracketabs _ a           = Comb K a


-- | SKI abstraction of a named lambda term. From a lambda expression
-- creates a SKI equivalent expression. The following algorithm is a
-- version of the algorithm (9.10) on the Hindley-Seldin book.
skiabs :: NamedLambda -> Ski
skiabs (LambdaVariable x)      = Cte x
skiabs (LambdaApplication m n) = Comb (skiabs m) (skiabs n)
skiabs (LambdaAbstraction x m) = bracketabs x (skiabs m)
```