Overview:
- Parser State
- Simple Parsers
- Lexical Analyzes
- Attribute Handling
- Parsing a Language
- Combining Monads

References:
- David J. King and Philip Wadler, “Combining Monads”, Functional Programming, 1992
Parsers

- A parser takes an input string into an abstract syntax tree. Both the input string and the abstract syntax tree represent a "parser state”.

- There are several techniques to parse a input string:
  - Top-down parsing
  - Recursive descent parsing
  - Bottom-up parsing

- Hand-written parsers are usually implemented using recursive descent parsing. Since a parser maintains a state, we use a monadic style to implement a concrete parser in Haskell.
Parser State

- A parser takes an input state to list of possible results:

  ```haskell
  data Parser a = PS (ParserState -> [(a,ParserState)])
  ```

The ParseState represents the input string which is equipped with some additional information needed to calculate a concrete abstract syntax tree (e.g. ParserState = (Int,String)).

We use the list type to handle parser error. That is, if a specific parser fails to accept a given input string, then the empty list is returned.
instance Monad Parser where

\[ p >>= mp = PS f \]
\[ \text{where } f s = [(y,s'')|(x,s') \leftarrow \text{apply } p s, (y,s'') \leftarrow \text{apply } (mp x) s'] \]

\[ \text{return } v = PS f \]
\[ \text{where } f ps = [(v,ps)] \]

- “return v” takes a value (abstract syntax tree) and returns a parser that succeeds on every input state.

- “p >>= mp” combine two parsers. That is, the resulting parser accepts two syntax categories sequentially and returns a nonempty list if the parsing is successfully.
**Simple Parsers**

- **epsilon:**
  
  epsilon :: a -> Parser a
  
  epsilon a = PS (\ps -> [(a,ps)])

- **literal:**
  
  literal :: Terminal -> Parser ()
  
  literal t = PS (\(i,s) -> let (nt,v,cs) = getNextToken s
  
  in if nt == t then [(),(i,cs)] else []

- **value:**
  
  value :: Terminal -> Parser String
  
  value t = PS (\(i,s) -> let (nt,v,cs) = getNextToken s
  
  in if nt == t then [(v,(i,cs))] else []

Com S 541
Parser Combination

- Alternation:

  (!) :: Parser a -> Parser a -> Parser a
  p1 ! p2 = PS f
  where f s = (apply p1 s) ++ (apply p2 s)

- Repetition:

  parser_star :: Parser a -> Parser [a]
  parser_star p = do {x <- p;
                      xs <- parser_star p;
                      return (x:xs)} ! return []

  parser_plus :: Parser a -> Parser [a]
  parser_plus p = do {x <- p; xs <- parser_star p; return (x:xs)}
Lexical Analyzes

- A scanner transforms an input string into a token (terminal) string. Tokens and input string form a scanner state.

- `getNextToken :: String -> (Terminal, String, String)`

`getNextToken` takes a string and returns a triple consisting of a token, the string representation of the token, and the unprocessed input string.

Example:

- `getNextToken [] = (T_EOF, "", [ ] )`
- `getNextToken ('(':cs) = (T_LP, "", cs)`
- `getNextToken ('*':cs) = (T_MUL, "", cs)`

T_EOF, T_LP, and T_MUL are token codes used by the parser.
Handling Attributes

- Parser attributes are used for both to parameterize a parsing rule and to yield a result, which is an abstract syntax tree.

- In case of context free grammars there is, however, no need to parameterize a parser rule.

- In the monadic style one uses the epsilon-parser to yield results:

  ```
  parse_name = value T_NAME >>= (\n -> epsilon (Loc n))
  ```

  ```
  parse_call = value T_NAME >>=
               (\n -> literal T_LP >>
                 (parse_expression_list ! return []) >>=
                  (\el -> literal T_RP >> epsilon (Call n el)))
  ```
The parser for a language is defined top-down.

The result of the parser and all sub-parsers is an abstract syntax tree or the empty list (i.e. “fail”).

The grammar of the language must not have any left recursive rules, since these would lead to infinite recursive calls.

Left recursive rules can be transformed into semantically equivalent right recursive rules.

Adding error messages requires a detailed analysis of the current parser state. Never report an error if there is an alternative available. Once the first token of a syntax rule has been processed, one cannot report an error involving the left-hand-side of the rule.
Elimination of Left-recursion

data ProjZ = Zempty | ZComp Name ProjZ  

z_parse_proj :: Parser ProjZ  
z_parse_proj = (literal T_DOT >> value T_NAME >>=  
\n -> z_parse_proj >>= (\p -> epsilon (ZComp n p)))) !  
(epsilon ZEmpty)

z_parse_primvalue :: Parser Value  
z_parse_primvalue = parse_primvalue' >>=  
(\v1 -> z_parse_proj >>=  
(\v2 -> epsilon (build_projection_value v1 v2)))

parse_primvalue :: Parser Value  
parsed_primvalue = z_parse_primvalue
Combining Monads

- Sometimes we want to combine two monads in order to use the features of both simultaneously. However, monads do not compose. Instead, we use a different operation, called monad transformers.

- Combining the monad \( m \) with the non-deterministic choice:
  
  ```haskell
data STC m a = STC (m [a])
```

- Combining the monad \( m \) with the state-monad:
  
  ```haskell
data STM m a = STM (State -> m (a,State))
```

- A monad transformer is an operation \( \tau \), taking monads to monads, with the property that it is possible to promote computations from the underlying monad into the transformed monad:

  ```haskell
  promote :: Monad m => m \( \alpha \) -> \tau m \( \alpha \)
  ```
data STC m a = STC (m [a])

instance Monad m => Monad (STC m) where
  return v = STC (return [v])

  STC mc >>= fmc = STC (mc >>= (\xs -> flatSTC (map fmc xs)))

flatSTC :: Monad m => [STC m a] -> m [a]
flatSTC [] = return []
flatSTC ((STC x):xs) = do {y <- x;
                        ys <- (flatSTC xs);
                        return (y ++ ys)}

promoteSTC :: IO a -> STC IO a
promoteSTC g = STC (g >>= (\v -> return [v]))
Monad STM – IO + State

data STM m a = STM (State -> m (a,State))

instance Monad m => Monad (STM m) where
  return v = STM f
    where f st = return (v, st)

  mst >>= fmst = STM f
    where f st = do { (v, st') <- applySTM mst st; applySTM (fmst v) st' }

applySTM :: STM m a -> State -> m (a,State)
applySTM (STM f) st = f st

promoteSTM :: IO a -> STM IO a
promoteSTM g = STM f
    where f st = do { x <- g; return (x, st) }

Com S 541
In order to combine the three features IO, non-deterministic choice, and state, what is the correct definition to get the desired result?

\[
\text{data STMC } m \ a = \text{STMC } (\text{State } \rightarrow (m \ a,\text{State}))
\]

But then it appears impossible to write a suitable definition for >>=. A solution to this problem is to define the desired feature explicitly:

\[
\text{data STMC } m \ a = \text{STMC } (\text{State } \rightarrow m ([a],\text{State}))
\]

However, this definition is incorrect with respect to evaluation history represented by “State”. We have to maintain separate images of “State” for every evaluation step.
We get the correct implementation of the combination of IO, non-deterministic choice by combining the original definitions of STC and STM, that is, we use STC IO as parameter to build a concrete instance of STM:

$$\text{STMC } a = \text{STM } (\text{State} \rightarrow \text{STC } \text{IO } (a, \text{State}))$$

We only need to define one new predicate to promote a monad STC into a monad STMC:

$$\text{promoteSTMC :: STC } \text{IO } a \rightarrow \text{STM } (\text{STC } \text{IO}) \ a$$

$$\text{promoteSTMC } g = \text{STM } f$$

$$\text{where } f \text{ st } = \text{do } \{x \leftarrow g; \text{return } (x, \text{st})\}$$