Parsec Parsing

Parsec

- Parsec one of the standard libraries for building libraries.
- It is a combinator parser
- A parser parses a sequence of elements to create a structured value.
- It is a monadic computation, so it may support many non-standard morphisms

Specializing Parsec

- Parsec is abstract over numerous issues
  - What it means to be an input sequence
  - What kind of elements the sequence contains
  - What kind of internal state (e.g. row, column, file information) the parser tracks
  - What kind of Monadic structure (in addition to state) the parser supports.
- This makes it very general, but sometimes hard for beginners to use.

Example

```
type MParser a =
   ParsecT
     String     -- The input is a
                -- sequence of Char
     ()         -- The internal state
     Identity   -- The underlying monad
     a          -- the type of the
                --  object being parsed
```

Issues

- Some important issues when building parsers
  - Tokenizing -- splitting the input into tokens
    - Token classes -- identifiers, constants, operators, etc
  - Handling white space
  - Handling comments (another form of white space?)
  - Handling errors
  - Handling choice
  - Handling repetitions

Language Styles

- Parsec has a tool (library) for handling tokens, white space, and comments called language styles.
- It captures some common idioms associated with parsing programming languages.
- Aggregates small parsers for individual elements of a language style.

Example

```
myStyle = LanguageDef
  { commentStart   = "{-"
  , commentEnd     = "-}"
  , commentLine    = "--"
  , nestedComments = True
  , identStart     = lower
  , identLetter    = alphaNum <|> char '_' <|> char '\''
  , opStart        = oneof ":!#$%&*+./<=>?@\\^|-~"
  , opLetter       = oneof ":!#$%&*+./<=>?@\\^|-~"
  , caseSensitive  = True
  , reservedOpNames =
    ["<", "=", "+", "-", "*"]
  , reservedNames  =
    ["if","then","else", "while", "begin", "end"]
  }
```

Token Parsers

- Styles are used to create token parsers

```
myTP = makeTokenParser myStyle
```

- Token parsers specialize parsers for common elements of language parsing

Introduces abstract parsing elements

- lexeme
- whiteSpace
- identifier
- reserved
- symbol
- reservedOp
- operator
- comma

Tim’s Conventions

- lexemE x = lexeme myTp x

- I define specialized parsing elements over a token parser (like `myTP`) by using a Capital letter as the last letter of the name

Examples

```
lexemE p    = lexeme myTP p
parenS p    = between (symboL "(") (symboL ")") p
braceS p    = between (symboL "{") (symboL "}") p
bracketS p  = between (symboL "[") (symboL "]") p
symboL      = symbol myTP
whiteSp     = whiteSpace myTP
idenT      = identifier myTP
keyworD    = reserved myTP
commA       = comma myTP
resOp       = reservedOp myTP
opeR       = operator myTP
```

Simple Parsers

```
natural     = lexemE(number 10 digit)
arrow       = lexemE(string "->")
larrow      = lexemE(string "<-")
dot         = lexemE(char '.')
character c = lexemE(char c)

number ::  Integer -> MParser Char -> MParser Integer
number base baseDigit
    = do{ digits <- many1 baseDigit
        ; let n = foldl acc 0 digits
              acc x d = base*x + toInteger (digitToInt d)
        ; seq n (return n)
        }
signed p = do { f <- sign; n <- p ; return(f n)}
  where sign = (character '-' >> return (* (-1))) <|>
               (character '+' >> return id) <|>
               (return id)
```

Running Parsers

- A parser is a computation. To run it, we turn it into a function with type

```
Seq s -> m (Either ParseError a)
```

- Since it is monadic we need the “run” morphisms of the monads that make it up.

```
runMParser parser name tokens =
  runIdentity
    (runParserT parser () name tokens)
```

Special Purpose ways to run parsers

```
-- Skip whitespace before you begin
parse1 file x s = runMParser (whiteSp >> x)
  file s

-- Raise the an error if it occurs
parseWithName file x s =
  case parse1 file x s of
   Right(ans) -> ans
   Left message -> error (show message)

-- Parse with a default name for the input
parse2 x s = parseWithName "keyboard input" x s
```

More ways to parse

```
-- Parse and return the internal state
parse3 p s = putStrLn (show state) >> return object
  where (object,state) =
            parse2 (do { x <- p
                       ; st <- getState
                       ; return(x,st)}) s

-- Parse an t-object, return
-- (t,rest-of-input-not-parsed)
parse4 p s =
   parse2 (do { x <- p
              ; rest <- getInput
              ; return (x,rest)}) s
```

Parsing in other monads

```
-- Parse a string in an arbitray monad
parseString x s =
  case parse1 s x s of
   Right(ans) -> return ans
   Left message -> fail (show message)

-- Parse a File in the IO monad
parseFile parser file =
    do possible <- Control.Exception.try (readFile file)
       case possible of
         Right contents ->
            case parse1 file parser contents of
              Right ans -> return ans
              Left message -> error(show message)
         Left err -> error(show (err::IOError))
```

A richer example

- In this example we build a parser for simple imperative language.
- This language uses an underlying state monad that tracks whether a procedure name is declared before it is used.

```
type MParser a =
   ParsecT
    String    -- The input is a sequence of
  Char
    ()        -- The internal state
    (StateT (String -> Bool) Identity)
              -- The underlying monad
    a    -- the type of the object being parsed
```

The non-standard morphism

```
addProcedure:: String -> MParser ()
addProcedure s =
      lift (withStateT (extend s True)
                       (return ()))
  where extend:: Eq a => a -> b -> (a -> b) -> (a -> b)
        extend x y f =
         \ s -> if (s==x)
                   then y
                   else f s
```

Running Parsers must deal with the state

```
-- Extract a computation from the Parser Monad
-- Note the underlying monad is
-- (StateT (String -> Bool) Identity)
runMParser parser name tokens =
  runIdentity
    (runStateT
       (runParserT parser () name tokens)
       (const False))
```

Abstract Syntax

```
type name = String
type operator = String

data Exp = Var name
  | Int Int
  | Bool Bool
  | Oper Exp operator Exp


data Stmt = Assign name Exp
  | While Exp Stmt
  | If Exp Stmt Stmt
  | Call name [Exp]
  | Begin [Decl] [Stmt]

data Decl = Val name Exp
  | Fun name [name] Stmt
```

Simple Expressions

```
simpleP:: MParser Exp
simpleP = bool <|> var <|> int <|> parenS expP
  where var = fmap Var idenT
        int = do { n <- int32
                 ; return(Int n)}
        bool = (symboL "True" >>
                return (Bool True)) <|>
               (symboL "False" >>
                return (Bool False))
```

Handling Precedence

```
liftOp oper x y = Oper x oper y

-- A sequence of simple separated by "*"
factor = chainl1 simpleP mulop

mulop = (resOp "*" >> return (liftOp "*"))

-- A seqence of factor separated by "+" or "-"
term = chainl1 factor addop

addop = (resOp "+" >> return (liftOp "+")) <|>
        (resOp "-" >> return (liftOp "-"))
```

Finally general expressions

```
-- Expressions with different precedence
  levels
expP:: MParser Exp
expP = chainl1 term compareop

compareop =
   (resOp "<" >>
     return (liftOp "<")) <|>
   (resOp "=" >>
     return (liftOp "="))
```

Statements

- Here is where we use the state

```
data Stmt = Assign name Exp
   | While Exp Stmt
   | If Exp Stmt Stmt
   | Call name [Exp]
   | Begin [Decl] [Stmt]
```

The name must have been declared earlier in the program.

Parsing statements

```
stmtP =
  whileP <|> ifP <|> callP <|> blockP <|> assignP

assignP =
  do { x <- idenT
     ; symboL ":="
     ; e <- expP
     ; return (Assign x e)}

whileP =
  do { keyworD "while"
     ; tst <- expP
     ; keyworD "do"
     ; s <- stmtP
     ; return (While tst s )}
```

Continued

```
ifP =
  do { keyworD "if"
     ; tst <- expP
     ; keyworD "then"
     ; s <- stmtP
     ; keyworD "else"
     ; s2 <- stmtP
     ; return (If tst s s2)}

callP =
  do { keyworD "call"
     ; f <- idenT
     ; b <- testProcedure f
     ; if b
          then return ()
          else (unexpected ("undefined procedure call: "++f))
     ; xs <- parenS(sepBy expP commA)
     ; return (Call f xs)}
```

Blocks

- Parsing blocks is complicated since they have both declarations and statements.

```
blockP =
  do { keyworD "begin"
     ; xs <- sepBy (fmap (Left) declP <|>
                    fmap (Right) stmtP)
                   (symboL ";")
     ; keyworD "end"
     ; return(split [] [] xs)}
```

Splitting blocks

```
split ds ss [] = Begin ds ss
split ds [] (Left d : more) =
   split (ds ++ [d]) [] more
split ds ss (Left d : more) =
   Begin ds
        ( ss ++
          [split [] []
                 (Left d : more)])
split ds ss (Right s : more) =
   split ds (ss ++[s]) more
```