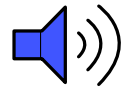
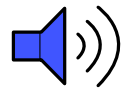
Lecture 13 – Parsec - continued



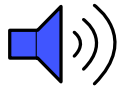
Outline:

- Build a language parser
- Lexer, error messages, and parse failure.
- Monadic parser
- Applicative parser

Parser for a little language – Fun



- ML like language with arithmetic and comparison expressions, integers, and variables.
- If-then-else, let expression, and function application.
- Anonymous function.
- Function declaration (named) and variable declaration.

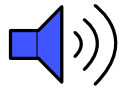
Fun Grammar 

- Use EBNF since it is closer to parser representation.

```
1  --  <DeclList> ::= { (<FunDecl> | <ValDecl>) }
2  --  <FunDecl>  ::= 'fun' <Ident> <Ident> '=' <Exp>
3  --  <ValDecl>  ::= 'val' <Ident> '=' <Exp>
4  --  <Expr> ::= <Comp>
5  --           | 'if' <Expr> 'then' <Expr> 'else' <Expr>
6  --           | 'let' <DeclList> 'in' <Expr> 'end'
7  --           | 'fn' <Ident> '=>' <Expr>
8  --  <Comp> ::= <Plus> { ('>' | '=' | '<') <Plus> }
9  --  <Plus> ::= <Mult> { ('+' | '-') <Mult> }
10 --  <Mult> ::= <App>  { ('*' | '/') <App> }
11 --  <App>  ::= <Fact> { <Fact> }
12 --  <Fact> ::= '(' <Expr> ')'
13 --           |  <Integer>
14 --           |  <Identifier>
```

- $\{e\}$ means zero or more e; $(e_1|e_2)$ means either e_1 or e_2.
- <Decl> is an arbitrary list of <FunDecl> and <ValDecl>.
- Grammar is carefully written to avoid left-recursion.

Fun Grammar – Lexer

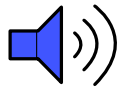


- Use EBNF since it is closer to parser representation.

```
1  --  <DeclList> ::= { (<FunDecl> | <ValDecl>) }
2  --  <FunDecl> ::= 'fun' <Ident> <Ident> '=' <Exp>
3  --  <ValDecl> ::= 'val' <Ident> '=' <Exp>
4  --  <Expr> ::= <Comp>
5  --           | 'if' <Expr> 'then' <Expr> 'else' <Expr>
6  --           | 'let' <DeclList> 'in' <Expr> 'end'
7  --           | 'fn' <Ident> '=>' <Expr>
8  --  <Comp> ::= <Plus> { ('>' | '=' | '<') <Plus> }
9  --  <Plus> ::= <Mult> { ('+' | '-') <Mult> }
10 --  <Mult> ::= <App>  { ('*' | '/') <App> }
11 --  <App>  ::= <Fact> { <Fact> }
12 --  <Fact> ::= '(' <Expr> ')'
13 --           |  <Integer>
14 --           |  <Identifier>
```

- Need to get rid of white spaces and recognize tokens.
- Symbols: `=>`, `=`, `>`, `<`, `+`, `-`, `*`, `/`, `(`, `)`.
- Keywords: fun, val, if, then, else, let, in, end, fn
- Literals: `<Integer>`, `<Ident>`, `<Identifer>`

Lexical analyzer



- Parsec does not need separate lexer. One parser can build tokens and recognize phrase structure the same time.
- Define Parser type that parses [Char] input.

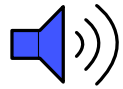
```
1 import Text.Parsec
2 type Parser = Parsec [Char] ()
```

- Get rid of trailing white space with lexeme.

```
1 lexeme :: Parser a -> Parser a
2 lexeme p = do x <- p
3               skipMany space <?> ""   -- <?> "" is to avoid
4               return x                --    spurious errors
```

- All symbols remove trailing spaces.

```
1 symbol :: String -> Parser String
2 symbol name = lexeme $ string name
3
4 eq_op = symbol "=" :: Parser String
```

Better error message 

- Parse raw identifier

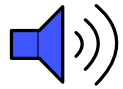
```
ident :: Parser String
ident = (lexeme $ many1 letter) <?> "identifier"
```

`p <?> "identifier"` emits error message that "expects identifier" if p fails to parse. This message is more meaningful than the internal error message of p such as "expects letter".

- Parse integer and return numeric value

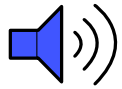
```
integer :: Parser Integer
integer = (read <$> (lexeme $ many1 digit)) <?> "integer"
```

Since Parser is also a functor, we can use '`read <$> p`' to convert the 'Parser String' to 'Parser Integer'.

Allow backtrack 

- Keyword passers use ‘try’ to enable backtrack. This is needed since keywords may share prefix with identifiers.

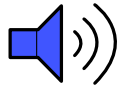
```
keyword :: String -> Parser ()
keyword name = (try $ symbol name) >> return ()

-- keyword parsers
if_ = keyword "if"
then_ = keyword "then"
else_ = keyword "else"
fn_ = keyword "fn"
fun_ = keyword "fun"
let_ = keyword "let"
in_ = keyword "in"
end_ = keyword "end"
val_ = keyword "val"
```

Parse failure 

- Identifiers for variable names may not collide with reserved words. If 'ident' returns a reserved word, then it is a parse error but 'try' call allows backtrack to try alternative parser.

```
1  import Text.Parsec.Prim (unexpected)
2
3  identifier :: Parser String
4  identifier = try $ do
5      name <- ident
6      if name `elem` reserved
7        then unexpected ("reserved word " ++ name) -- error
8        else return name
9    where
10     reserved = ["if", "then", "else", "fn", "fun",
11                           "let", "in", "end", "val"]
```

- 'unexpected' function returns parse error with customized error message.
- May also use 'fail' function or return parse error directly (though that is mostly in low-level definitions).

Run parser 

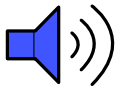
- ‘run’ print the result of the parser or error message since Parser returns an Either monad.

```haskell
import Data.Either

run :: Show a => Parser a -> String -> IO()
run p input = putStrLn $ case parse p "" input of
        Left error -> show error
        Right x    -> show x
```

- ‘runParser’ just returns the Either monad.

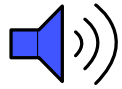
```haskell
runParser :: Parser a -> String -> Either ParseError a
runParser p input = parse p "" input
```

AST



- Fun parser should return a list of AST – one for each declaration.

```
1  -- function/variable declaration
2  data Decl = Fun String String Exp -- fun f x = e;
3            | Val String Exp        -- val x = e;
4
5  -- expressions
6  data Exp = Lt Exp Exp      -- e1 < e2
7           | Gt Exp Exp      -- e1 > e2
8           | Eq Exp Exp      -- e1 = e2
9           | Plus Exp Exp    -- e1 + e2
10          | Minus Exp Exp   -- e1 - e2
11          | Times Exp Exp   -- e1 * e2
12          | Div Exp Exp     -- e1 div e2
13          | Var String      -- x
14          | If Exp Exp Exp  -- if e0 then e1 else e2
15          | Fn String Exp   -- fn x => e
16          | Let [Decl] Exp  -- let val x = e0
17                            --     fun f = e1
18                            -- in e2 end
19
20          | App Exp Exp     -- e1 e2
21          | Const Integer   -- n
```

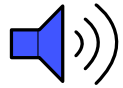
Top level parser 

- ‘prog’ is the top-level parser to skip leading white spaces and gets the result of ‘decl’, which is a list of ‘Decl’ AST node.

```
1 prog :: Parser [Decl]
2 prog = skipMany space >> declList
```

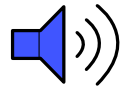
- ‘decl’ is any combination of ‘val’ declaration and ‘fun’ declaration.

```
1 declList:: Parser [Decl]
2 declList = many $ valDecl <|> funDecl
```

Monadic parser 

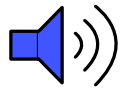
- Parsec is a monadic parser library. The base design without error messages is simple. **This is NOT the actual library.**

```
1  -- base type for parser
2  newtype Parser a = Parser {
3                       runParser :: String      -- input string
4                                 -> Consumed a -- 'a' is output
5                     }
6  data Consumed a =
7       Consumed (Reply a) -- consumed some string
8     | Empty (Reply a)    -- did not consume any string
9
10 data Reply a =
11      Ok a String  -- succeed with `a' and remaining string
12    | Error        -- just error - no details
```

Monadic parser 

- Parser is a Functor

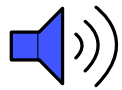
```haskell
newtype Parser a = Parser {runParser :: String -> Consumed a}

data Consumed a = Consumed (Reply a)
                | Empty (Reply a)

data Reply a = Ok a String | Error

-- Functor instances

instance Functor Reply where
  fmap f (Ok a rest) = Ok (f a) rest
  fmap _ Error = Error

instance Functor Consumed where
  fmap f (Consumed reply) = Consumed $ fmap f reply
  fmap f (Empty reply) = Empty $ fmap f reply

instance Functor Parser where
  fmap f (Parser p) = Parser $ \input -> fmap f (p input)
```

Monadic parser 

- Parser is an Applicative

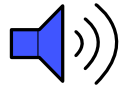
```haskell
newtype Parser a = Parser {runParser :: String -> Consumed a}

data Consumed a = Consumed (Reply a) | Empty (Reply a)

data Reply a = Ok a String | Error

-- Applicative instance
instance Applicative Parser where
  pure x = Parser $ \input -> Empty $ Ok x input

  Parser p <*> Parser q = Parser $ \input ->
     case p input of
        Consumed (Ok f rest) -> Consumed $ case q rest of
              Consumed reply -> fmap f reply
              Empty reply    -> fmap f reply

        Empty (Ok f rest) -> fmap f $ q rest

        Consumed Error -> Consumed Error
        Empty Error    -> Empty Error
```

Monadic parser



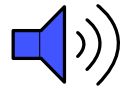
- Parser is a Monad

```haskell
newtype Parser a = Parser {runParser :: String -> Consumed a}

data Consumed a = Consumed (Reply a) | Empty (Reply a)

data Reply a = Ok a String | Error

-- Monad instance
instance Monad Parser where
  return = pure

  Parser p >>= f = Parser $ \input ->
    case p input of
        Consumed (Ok x rest) -> Consumed $
           case runParser (f x) rest of
              Consumed reply -> reply
              Empty reply    -> reply

        Empty (Ok x rest) -> runParser (f x) rest

        Consumed Error -> Consumed Error
        Empty Error    -> Empty Error
```

Low-level parsers 

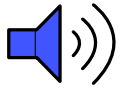
- Boolean parser: accept or reject a charactor.

```
1  import Data.Char (isAlpha, isDigit)
2
3  newtype Parser a = Parser {runParser :: String -> Consumed a}
4
5  data Consumed a = Consumed (Reply a) | Empty (Reply a)
6
7  data Reply a = Ok a String | Error
8
9  -- Boolean parser
10 satisfy :: (Char -> Bool) -> Parser Char
11 satisfy test = Parser $ \input ->
12    case input of
13         []                -> Empty Error -- unexpected end
14         c:cs | test c     -> Consumed (Ok c cs) -- success
15              | otherwise -> Empty Error         -- failure
16
17 char c = satisfy (== c)          -- parse specific charactor
18 letter = satisfy isAlpha         -- parse any letter
19 digit  = satisfy isDigit         -- parse any digit
20
21 oneOf str = satisfy (\c -> c `elem` str)
22 space = oneOf " \t\n"
```

Low-level parsers 

- Choice parser: if left fails to consume, then run right.

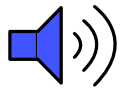
```haskell
newtype Parser a = Parser {runParser :: String -> Consumed a}

data Consumed a = Consumed (Reply a) | Empty (Reply a)

data Reply a = Ok a String | Error

-- choice parser
(<|>) :: Parser a -> Parser a -> Parser a

Parser p <|> Parser q =
   Parser $ \input ->
     case p input of
        Empty Error -> q input    -- run 'q' if 'p' fails

        Empty ok    -> case q input of
                         -- use 'p' if 'q' doesn't consume
                         Empty _  -> Empty ok
                         -- otherwise, use 'q'
                         consumed -> consumed

        -- 'p' may have failed after consuming some input
        consumed    -> consumed
```

Low-level parsers 

- Try parser: backtrack if fails to parse after consuming input.

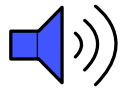
```haskell
newtype Parser a = Parser {runParser :: String -> Consumed a}

data Consumed a = Consumed (Reply a) | Empty (Reply a)

data Reply a = Ok a String | Error

-- try parser
try :: Parser a -> Parser a

try (Parser p) = Parser $ \input ->
    case p input of

        Consumed Error -> Empty Error  -- signal backtrack.

        other          -> other
```

Low-level parsers



- Example of using monadic operator.

```
1  newtype Parser a = Parser {runParser :: String -> Consumed a}
2
3  data Consumed a = Consumed (Reply a) | Empty (Reply a)
4
5  data Reply a = Ok a String | Error
6
7  -- string parser
8  string :: String -> Parser ()
9  string ""     = return ()
10
11 -- '>>' propagate effects (input string, success or failure)
12 string (c:cs) = char c >> string cs
13
14 -- list parser
15 many1 :: Parser a -> Parser [a]
16 many1 p = do
17    x  <- p                        -- '<-' extracts value
18    xs <- many1 p <|> return []    -- and propagates effects
19    return $ x:xs                  -- return value/effects
```

Applicative parsers 

- Since Parsec is Applicative and Functor, <*>, its cousins <* and *>, and its little friends <$ and <$> may come in handy.

```
1  (<*>) ::f (a -> b) -> f a -> f b
2  (<* ) :: f a -> f b -> f a
3  ( *>) :: f a -> f b -> f b
4
5  lexeme :: Parser a -> Parser a
6  lexeme p = do { x <- p; skipMany space; return x }
7
8  lexeme' p = p <* skipMany space -- shorter!
9
10 (<$>) :: (a -> b) -> f a -> f b
11 (<$ ) :: a -> f b -> f a
12
13 p  = char '+' >> return ' ' -- parse '+' but returns ' '
14
15 p' = ' ' <$ char '+'  -- shorter!
```