

Programming Languages CPSC-314 — Slides 8

Functional Parsers

Jaakko Järvi

February 26, 2009

Parser Type

- Parser takes tokens to syntax trees
- Now, tokens are just characters

```
type Parser = String → Tree
```

- To accommodate for not consuming the entire input, we use the type:

Parser Type

- Parser takes tokens to syntax trees
- Now, tokens are just characters

```
type Parser = String → Tree
```

- To accommodate for not consuming the entire input, we use the type:

```
type Parser = String → (Tree, String)
```

- We know that parses are ambiguous, so we generalize the result to:

Parser Type

- Parser takes tokens to syntax trees
- Now, tokens are just characters

```
type Parser = String → Tree
```

- To accommodate for not consuming the entire input, we use the type:

```
type Parser = String → (Tree, String)
```

- We know that parses are ambiguous, so we generalize the result to:

```
type Parser = String → [(Tree, String)]
```

- And what kind of Tree really we want as a result?

Parser Type

- Parser takes tokens to syntax trees
- Now, tokens are just characters

```
type Parser = String → Tree
```

- To accommodate for not consuming the entire input, we use the type:

```
type Parser = String → (Tree, String)
```

- We know that parses are ambiguous, so we generalize the result to:

```
type Parser = String → [(Tree, String)]
```

- And what kind of Tree really we want as a result?

```
type Parser a = String → [(a, String)]
```

- And what if we are parsing token streams instead of character streams?

Parser Type

- Parser takes tokens to syntax trees
- Now, tokens are just characters

```
type Parser = String → Tree
```

- To accommodate for not consuming the entire input, we use the type:

```
type Parser = String → (Tree, String)
```

- We know that parses are ambiguous, so we generalize the result to:

```
type Parser = String → [(Tree, String)]
```

- And what kind of Tree really we want as a result?

```
type Parser a = String → [(a, String)]
```

- And what if we are parsing token streams instead of character streams?

```
type TokenParser b a = [b] → [(a, [b])]
```

Building blocks

- Most basic parser is the one that consumes no input and always succeeds:

```
return :: a → Parser a
return v = \inp → [(v, inp)]
```

- Its counterpart `failure` always fails

```
failure :: Parser a
failure = \inp → []
```

- Examples:

```
> return 7 "parse this"
[(7, "parse this")]
> failure "parse this"
[]
```

Item parser

- `item` extracts the first character and succeeds with that character as the result, fails if no characters left

```
item :: Parser Char
```

```
item = \inp → case inp of
```

```
    [] → []
```

```
    (x:xs) → [(x,xs)]
```

- Example:

```
> item "parse this"
```

```
[( 'p', "arse this")]
```

The parse function

- Even though parsers are functions and can thus be applied directly, sometimes more explicit to do that with the help of the `parse` function:

```
parse :: Parser a → String → [(a, String)]  
parse p inp = p inp
```

- Example:

```
> parse (item) "parse this"
```

Choice

- What if we have to backtrack? Try to first parse p , then q ?
- We can define a composition operator for two parsers

```
(+++)  
p +++ q = \inp → case parse p inp of  
                [] → parse q inp  
                [(v, out)] → [(v, out)]
```

- Example:

```
> parse (failure) "abc"  
[]  
> parse (failure +++ item) "abc"  
[('a', "bc")]
```

Sequencing

- Commonly we want to sequence parsers, e.g., think of typical grammars

```
<if-stmt> :: if ( <expr> ) then <stmt>
```

- First parse `if`, then `(`, then `<expr>`, ...
- How to combine results?

```
Parser a → Parser b → Parser (a, b) -- ?
```

Sequencing

- Commonly we want to sequence parsers, e.g., think of typical grammars

```
<if-stmt> :: if ( <expr> ) then <stmt>
```

- First parse `if`, then `(`, then `<expr>`, ...
- How to combine results?

```
Parser a → Parser b → Parser (a, b) -- ?
```

- Possible, but awkward
(try to combine n parsers, where n is large)

The “monadic” way

- Parser sequencing operator

$(\gg=) :: \text{Parser } a \rightarrow (a \rightarrow \text{Parser } b) \rightarrow \text{Parser } b$

```
p >>= f = \inp → case parse p inp of
    [] → []
    [(v, out)] → parse (f v) out
```

- $p \gg= f$
 - fails if p fails
 - otherwise applies f to the result of p
 - this results in a new parser, which is then applied
- Example:

```
> parse ((failure +++ item) >>= (\_ → item)) "abc"
[('b', "c")]
```

- The key benefit: **the result of first parse is available for the subsequent parsers**

```
> parse (item >>= (\x → item >>= (\y → return (y:[x]))) "ab"
```

The “monadic” way

- Parser sequencing operator

$(\gg=) :: \text{Parser } a \rightarrow (a \rightarrow \text{Parser } b) \rightarrow \text{Parser } b$

```
p >>= f = \inp → case parse p inp of
    [] → []
    [(v, out)] → parse (f v) out
```

- $p \gg= f$
 - fails if p fails
 - otherwise applies f to the result of p
 - this results in a new parser, which is then applied
- Example:

```
> parse ((failure +++ item) >>= (\_ → item)) "abc"
[('b', "c")]
```

- The key benefit: **the result of first parse is available for the subsequent parsers**

```
> parse (item >>= (\x → item >>= (\y → return (y:[x]))) "ab"
[("ba", "")]
```

Sequencing

- Parsers are typically structured this way:

```
p1 >>= \v1 →
p2 >>= \v2 →
...
pn >>= \v3 →
return (f v1 v2 ... vn)
```

- Syntactic sugar for this construct is provided:

```
do v1 ← p1
   v2 ← p2
   ...
   vn ← pn
return (f v1 v2 ... vn)
```

- if some `vi` is not needed, `vi ← pi` can be written as `vi`
 - corresponds to:

```
pi >> \_ → ...
```

Examples

```

rev3 =
  item >>= \v1 →
  item >>= \v2 →
  item >>= \_ →
  item >>= \v3 →
  return $
    reverse (v1:v2:v3:[])

```

```

rev3 =
  do v1 ← item
     v2 ← item
     item
     v3 ← item
  return $
    reverse (v1:v2:v3:[])

```

- Example use:

```

> rev3 "abcdef"
[("dba","ef")]

```

- What happens here?

```

> (rev3 >>= (\_ → item)) "abcde"

```

```

> (rev3 >>= (\_ → item)) "abcd"

```

Examples

```

rev3 =
  item >>= \v1 →
  item >>= \v2 →
  item >>= \_ →
  item >>= \v3 →
  return $
    reverse (v1:v2:v3:[])

```

```

rev3 =
  do v1 ← item
     v2 ← item
     item
     v3 ← item
  return $
    reverse (v1:v2:v3:[])

```

- Example use:

```

> rev3 "abcdef"
[("dba","ef")]

```

- What happens here?

```

> (rev3 >>= (\_ → item)) "abcde"
[('e',"")]
> (rev3 >>= (\_ → item)) "abcd"
[]

```

sat

- Parser that accepts a character satisfying a predicate:

```
sat :: (Char → Bool) → String → [(Char, String)]
sat p = item >>= \v1 →
    if p v1 then return v1 else failure
```

- Examples:

```
> parse (sat (=='a')) "abc"
[('a', "bc")]
```

```
> parse (sat (=='b')) "abc"
[]
```

```
> parse (sat isLower) "abc"
[('a', "bc")]
```

```
> parse (sat isUpper) "abc"
[]
```

Parsers derived from sat

```
digit :: Parser Char  
digit = sat isDigit
```

```
lower :: Parser Char  
lower = sat isLower
```

```
upper :: Parser Char  
upper = sat isUpper
```

```
letter :: Parser Char  
letter = sat isAlpha
```

```
alphanum :: Parser Char  
alphanum = sat isAlphaNum
```

```
char :: Char → Parser Char  
char x = sat (== x)
```

Accept a particular string

- We can use sequencing recursively

```
string :: String → Parser String
string [] = return []
string (x:xs) = char x >>= \c →
                string xs >>= \cs →
                return (c:cs)
```

- Note that entire `string` parse fails if any of the recursive calls fail
- Example:

```
> parse (string "if") "if (a<b) return;"
[("if", " (a<b) return;")]
```

many

- Apply the same parser many times:

```
many1 :: Parser a → Parser [a]
```

```
many1 p = p >>= \v →
    many p >>= \vs →
    return (v:vs)
```

```
many :: Parser a → Parser [a]
```

```
many p = many1 p +++ return []
```

- Examples:

```
*Main> parse (many digit) "123"
```

```
*Main> parse (many digit) "a123"
```

many

- Apply the same parser many times:

```
many1 :: Parser a → Parser [a]
many1 p = p >>= \v →
    many p >>= \vs →
    return (v:vs)
```

```
many :: Parser a → Parser [a]
many p = many1 p +++ return []
```

- Examples:

```
*Main> parse (many digit) "123"
[("123","")]
*Main> parse (many digit) "a123"
[("", "a123")]
```

First semi-realistic example

```
ident :: Parser String
ident = letter >>= \x →
    many alphanum >>= \xs →
    return (x:xs)
```

```
> parse ident "2len = 5"
[]
> parse ident "len2 = 5"
[("len2"," = 5")]
```

Second semi-realistic example

```
nat :: Parser Int
nat = many1 digit >>= \xs →
  return (read xs)
```

```
list = string "[" >>= \_ →
  nat >>= \v →
  many (string "," >>= \_ → nat) >>= \vs →
  string "]" >>= \_ →
  return (v:vs)
```

```
> list "[1,2,3,4]"
[[1,2,3,4], ""]
> ( sum . fst . head . list ) "[1,2,3,4]"
10
```

Sequence and ignore

- We saw many of these

```
p >>= \_ → ...
```

- We can capture this pattern into a combinator of its own:

```
(>>=) :: Parser a → (a → Parser b) → Parser b
```

```
p >>= f = \inp → case parse p inp of
```

```
    [] → []
```

```
    [(v, out)] → parse (f v) out
```

```
(>>) :: Parser a → Parser b → Parser b
```

```
p >> q = p >>= \_ → q
```

- Examples:

```
> (item >> item >> item) "abcd"
```

```
[( 'c', "d" )]
```

Larger example

```
list = string "[" >>= \_ →
      nat >>= \v →
      many (string "," >>= \_ → nat) >>= \vs →
      string "]" >>= \_ →
      return (v:vs)
```

```
list = string "[" >>
      nat >>= \v →
      many (string "," >> nat) >>= \vs →
      string "]" >>
      return (v:vs)
```

Parsing tokens

- Often whitespace uninteresting

```
space :: Parser ()
space = many (sat isSpace) >>
      return ()
```

```
token :: Parser a → Parser a
token p = space >>
        p >>= \v →
        space >>
        return v
```

- Some parsers that ignore differences in spacing:

```
symbol :: String → Parser String
symbol xs = token (string xs)
```

```
natural :: Parser Int
natural = token nat
```

```
identifier :: Parser String
identifier = token ident
```

List that ignores spacing

```
list = string "[" >>
      nat >>= \v →
      many (string "," >> nat) >>= \vs →
      string "]" >>
      return (v:vs)
```

```
list = symbol "[" >>
      natural >>= \v →
      many (symbol "," >> natural) >>= \vs →
      symbol "]" >>
      return (v:vs)
```

```
> list " [1, 2, 3 , 4 ] "
[[1,2,3,4], ""]
```

The Parser monad

```
import Monad

newtype Parser a = P (String → [(a,String)])

instance Monad Parser where
  return v = P (\inp → [(v,inp)])
  p >>= f = P (\inp → case parse p inp of
                        []           → []
                        [(v,out)] → parse (f v) out)
```

The same in “do”-notation

```
list =  
  symbol "[" >>  
  natural >>= \v →  
  many (symbol "," >> natural) >>= \vs →  
  symbol "]" >>  
  return (v:vs)
```

```
list = do  
  symbol "["  
  v ← natural  
  vs ← many (symbol "," >> natural)  
  symbol "]"  
  return (v:vs)
```