15. Case Study: Parsing

15.1 Basic Parsing
Parsing is the translation of a string into a syntax tree
15.1 Basic Parsing

Parsing is the translation of a string into a syntax tree according to some grammar.

Example

```
"a+b*c"  →  
```

```
  +
  /  
 a   *
     /  
 b   c
```

```
type Parser = String -> Tree

Parser type
```

```
type Parser a = String -> a
```

```
Parser type
```

What if something is left over, e.g., "a+b*c#"?
Parser type

type Parser = String \rightarrow Tree

type Parser a = String \rightarrow a

What if something is left over, e.g., "a+b*c#" ?

type Parser a = String \rightarrow (a, String)

What if there is a syntax error, e.g., "++" ?

type Parser a = String \rightarrow [(a, String)]
Parser type

type Parser = String -> Tree

type Parser a = String -> a

What if something is left over, e.g., "a+b*c#"?
type Parser a = String -> (a,String)

What if there is a syntax error, e.g., "++"?
type Parser a = String -> [(a,String)]
  [] syntax error
  [x] one result x
  [x,y,...] multiple results, ambiguous language

---

15.1 Basic Parsing

Parsing is the translation of a string into a syntax tree according to some grammar.

Example

"a+b*c" →

```
+  
/ 
/  
a  b  c
```

What if something is left over, e.g., "a+b*c#"?
type Parser a = String -> (a,String)

What if there is a syntax error, e.g., "++"?
type Parser a = String -> [(a,String)]
  [] syntax error
  [x] one result x
  [x,y,...] multiple results, ambiguous language
For unambiguous languages:

\[
\text{type } \text{Parser } a = \text{String } \rightarrow \text{Maybe } (a, \text{String})
\]
```plaintext
Basic parsers

one :: (Char -> Bool) -> Parser Char
one pred (x:xs) = if pred x then [(x, xs)] else []
one _ [] = []

char :: Char -> Parser Char
char c = one (== c)

Example
char 'a' "abc" =
```

```plaintext
Basic parsers

one :: (Char -> Bool) -> Parser Char
one pred (x:xs) = if pred x then [(x, xs)] else []
one _ [] = []

char :: Char -> Parser Char
char c = one (== c)

Example
char 'a' "abc" = [('a', 'bc')]
```
Basic parsers

one :: (Char -> Bool) -> Parser Char
one pred (x:xs) = if pred x then [(x,xs)] else []
one [] = []

char :: Char -> Parser Char
char c = one (== c)

Example
char 'a' "abc" = [('a','bc')]
char 'b' "abc" = []

Combining parsers

Parse anything that p1 or p2 can parse:

(|||) :: Parser a -> Parser a -> Parser a
p1 ||| p2 =

Combining parsers

Parse anything that p1 or p2 can parse:

(|||) :: Parser a -> Parser a -> Parser a
p1 ||| p2 = \
Example
(char 'b' ||| char 'a') "abc" =
Combining parsers

Parse anything that p1 or p2 can parse:

$$ (\text{|||}) :: \text{Parser } a \rightarrow \text{Parser } a \rightarrow \text{Parser } a $$

$$ p1 \text{ ||| } p2 = \text{\langle } \text{cs} \rightarrow p1 \text{ cs } +\text{\rangle } p2 \text{ cs} $$

Example

$$ \text{(char 'b' ||| char 'a')} "abc" = $$

Combining parsers

Parse first with p1, then the remainder with p2:

$$ (***) :: \text{Parser } a \rightarrow \text{Parser } b \rightarrow \text{Parser } (a,b) $$

$$ (p1 ** p2) xs = $$
Combining parsers

Parse first with p1, then the remainder with p2:

\[
(***) :: \text{Parser } a \rightarrow \text{Parser } b \rightarrow \text{Parser } (a,b)\\
(p1 *** p2) \text{ } xs = [((a,b),zs) \mid (a,ys) \leftarrow p1 \text{ } xs, (b,zs) \leftarrow p2 \text{ } ys]
\]

Example

(char 'b' *** char 'a') "bac" =

Combining parsers

Parse first with p1, then the remainder with p2:

\[
(***) :: \text{Parser } a \rightarrow \text{Parser } b \rightarrow \text{Parser } (a,b)\\
(p1 *** p2) \text{ } xs = [((a,b),zs) \mid (a,ys) \leftarrow p1 \text{ } xs, (b,zs) \leftarrow p2 \text{ } ys]
\]

Example

(char 'b' *** char 'a') "bac" = [((a',b'), "c")]

(one isAlpha *** one isDigit *** one isDigit) "a12"
Combining parsers

Parse first with p1, then the remainder with p2:

\[ (***) :: \text{Parser} \ a \rightarrow \text{Parser} \ b \rightarrow \text{Parser} \ (a,b) \]

\[ (p1 \ ** \ p2) \ \text{xs} = \]
\[ (((a,b),zs) \mid (a,ys) \leftarrow p1 \ \text{xs}, (b,zs) \leftarrow p2 \ \text{ys}) \]

Example

(char 'b' ** char 'a') "bac" = \[ (('a','b'),"c") \]

(one isAlpha ** one isDigit ** one isDigit) "a12" = \[ (('a','1','2'),")" \]

Transforming the result

Parse with p, transform result with f:

\[ (>>>) :: \text{Parser} \ a \rightarrow (a \rightarrow b) \rightarrow \text{Parser} \ b \]

\[ p \ >>> f = \ \text{\lambda} \ \text{xs} \rightarrow [(f \ a,ys) \mid (a,ys) \leftarrow p \ \text{xs}] \]

Example

((char 'b' ** char 'a') >>> ((x,y) -> [x,y])) "bac"
Transforming the result

Parse with p, transform result with f:

\[
(\triangleright\triangleright\triangleright) :: \text{Parser } a \to (a \to b) \to \text{Parser } b
\]

\[
p \triangleright\triangleright\triangleright f = \ \text{\{xs} \to [(f \ a,ys) \mid (a,ys) \gets p \ \text{\{xs}}]
\]

Example

\[
((\text{char 'b' } \star\star\star \text{ char 'a'}) \triangleright\triangleright ((\text{x,y} \to [x,y]))) \text{ "bac"}
\]

\[
= \text{[(["ab"], "c")]}\]

Parsing a list of objects

Auxiliary functions:

\[
\text{uncurry} :: (a \to b \to c) \to (a,b) \to c
\]

\[
\text{uncurry } f (a,b) = f \ a \ b
\]

Auxiliary functions:

\[
\text{uncurry} :: (a \to b \to c) \to (a,b) \to c
\]

\[
\text{uncurry } f (a,b) = f \ a \ b
\]

\[
\text{success} :: a \to \text{Parser } a
\]

\[
\text{success } a \ \text{\{xs} = [\text{(a,\{xs\})}]
\]
Parsing a list of objects

Auxiliary functions:

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b) = f a b

success :: a -> Parser a
success a xs = [(a, xs)]

The parser transformer:

list :: Parser a -> Parser [a]
list p = (p *** list p) >>= uncurry ()

---------

Parsing a list of objects

Auxiliary functions:

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b) = f a b

success :: a -> Parser a
success a xs = [(a, xs)]

The parser transformer:

list :: Parser a -> Parser [a]
list p = (p *** list p) >>= uncurry ()

---------

Parsing a list of objects

Auxiliary functions:

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b) = f a b

success :: a -> Parser a
success a xs = [(a, xs)]

The parser transformer:

list :: Parser a -> Parser [a]
list p = (p *** list p) >>= uncurry ()

---------

Parsing a list of objects

Auxiliary functions:

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b) = f a b

success :: a -> Parser a
success a xs = [(a, xs)]

The parser transformer:

list :: Parser a -> Parser [a]
list p = (p *** list p) >>= uncurry ()

---------
Parsing a list of objects

Auxiliary functions:

```
uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b) = f a b
```

```
success :: a -> Parser a
success a xs = [(a,xs)]
```

The parser transformer:

```
list :: Parser a -> Parser [a]
list p = (p *** list p) >> uncurry ()
  ||| success []
```
Parsing a list of objects

Auxiliary functions:

uncurry :: (a -> b -> c) -> (a,b) -> c
uncurry f (a,b) = f a b

success :: a -> Parser a
success a xs = [(a, xs)]

The parser transformer:

list :: Parser a -> Parser [a]
list p = (p *** list p) >>> uncurry ()
        ||| success []

Example
list (one isAlpha) "abc123" =

Parsing a non-empty list of objects

list1 :: Parser a -> Parser [a]
list1 p = (p *** list p) >>> uncurry ()

Parsing identifiers

ident :: Parser String

Example
ident (one isAlpha) "abc123" = ["abc", "123"]
Parsing identifiers

```haskell
ident :: Parser String
ident = (list1(one isAlpha) *** list(one isDigit))

Example
ident "abc0++" =
```

Parsing identifiers

```haskell
ident :: Parser String
ident = (list1(one isAlpha) *** list(one isDigit))
    >>> uncurry (++)

Example
ident "abc0++" = ["abc0", "++"]
```
spaces :: Parser String
spaces = list (one isSpace)
sp :: Parser a -> Parser a
sp p = (spaces *** p) >>> snd

Example
(sp indent) " abc d" =
### Handling spaces

```
spaces :: Parser String
spaces   = list (one isSpace)

sp :: Parser a -> Parser a
sp p   = (spaces *** p) >>> snd
```

**Example**
```
(sp ident) " abc d" = ["abc", " d"]
```

---

### 15.2 Application: Parsing pico-Haskell expressions

Context-free grammar (= BNF notation) for expressions:

```
expr ::= identifier
      | ( expr expr )
      | ( \ identifier . expr )
```

**Examples**
a, (f x)
15.2 Application: Parsing pico-Haskell expressions

Context-free grammar (= BNF notation) for expressions:

```
expr ::= identifier
      | ( expr expr )
      | ( \ identifier . expr )
```

Examples a, (f x), (\x. (f x))

The tree representation:

data Expr = Id String | App Expr Expr | Lam String Expr

Examples Id "a"
        App (Id "f") (Id "x")

Pico-Haskell parser

ch c = sp (char c)
id = sp ident

expr =

```
ch c = sp (char c)
id = sp ident

expr =

15.2 Application: Parsing pico-Haskell expressions
Context-free grammar (= BNF notation) for expressions:

expr ::= identifier
    | ( expr expr )
    | ( \ identifier . expr )

Examples a, (f x), (\x. (f x))

The tree representation:
data Expr = Id String | App Expr Expr | Lam String Expr

Examples Id "a"
    App (Id "f") (Id "x")
    Lam "x" (App (Id "f") (Id "x"))
15.3 Improved Parsing

Example

data Token =
  LParant | RParant | BSlash | Dot | Ident String

"(\x1 . x2)" "Lexer→
  [LParant, BSlash, Ident "x1", Dot, Ident "x2", RParant]

Why?
  - Lexer based on regular expressions
    ⇒ lexer can be more efficient than general parser

Generalizing the implementation

So far:
type Parser a = String → [(a,String)]
15.3 Improved Parsing

String $\xrightarrow{\text{Lexer}}$ [Token] $\xrightarrow{\text{Parser}}$ Tree

**Example**

data Token =
    LParant | RParant | BSlash | Dot | Ident String
"(\x1 . x2)" $\xrightarrow{\text{Lexer}}$
    [LParant, BSlash, Ident "x1", Dot, Ident "x2", RParant]

*Why?*
- Lexer based on regular expressions
  - lexer can be more efficient than general parser
- Lexer can already remove spaces and comments
  - simplifies parsing

Generalizing the implementation

So far:

\[
\text{type } \text{Parser } a = \text{String} \rightarrow [(a, \text{String})]
\]

Now:

\[
\text{type } \text{Parser } a \ b = [a] \rightarrow [(b, [a])]
\]

None of the parser combinators $\ast\ast\ast$, $\mid\mid\mid$, $\gg\gg\gg$ change,
only their types become more general!

So far:

\[
(\ast\ast\ast) :: \text{Parser } a \rightarrow \text{Parser } b \rightarrow \text{Parser } (a, b)
\]

Now:

\[
(\ast\ast\ast) ::
\]

Some literature:
- Chapter 8 of Hutton's *Programming in Haskell*
- Section 17.5 in Thompson's Haskell book (3rd edition)
- Many papers on functional parsers