Language Engineering | Lecture 9.

What is a parser? Here are some candidates:

```
String → a
```

This is a bit too optimistic; it assumes that the parse always succeeds with the value of type a.

```
String → Maybe a
```

This acknowledges that things may fail, but still too optimistic: there may be ambiguous but correct results:

```
String → [a]
```

Usually we also want to keep hold of the rest of the input that was not parsed:

```
String → [(String, a)]
```

possible parses

- `String` (input string / token stream)
- `String` (left-over input)
- `a` (parsed value)

Example:

Suppose we want to parse

1 + 2 + 3

Our grammar might be:

(1) expr : num
| expr "+" expr

this is equivalent to:

(2) expr : num ["+" expr]*

The p* represents many copies of the parser p (≥ 0)

The p+ represents some copies of the parser p (> 0)

The grammar (1) is troublesome because it expresses unguarded recursion.

Recursion is guarded when some token is consumed before the recursive call.

We might construct a parser

```
expr :: String → [(String, Expr)]
```

where the type Expr could be:

```
data Expr = Num Int
          | Plus Expr Expr
```

Possible results of applying expr:

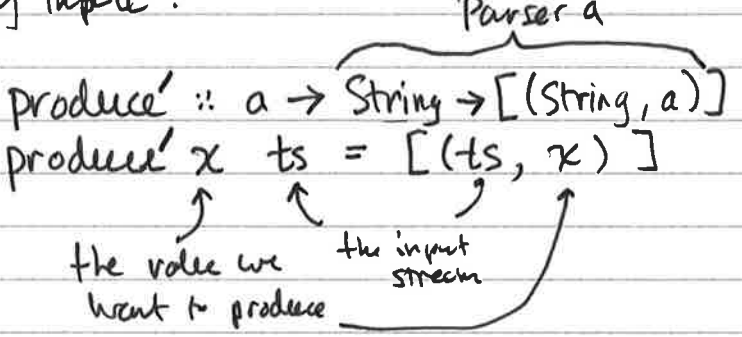
```
expr "1+2+3"

[("", Plus (Num 1) (Plus (Num 2) (Num 3))
, ("", Plus (Plus (Num 1) (Num 2)) (Num 3))
, ("1", "+2+3", Num 1)
, ("+3", Plus (Num 1) (Num 2))
]
```

Given our grammar(s) these are the only possible results.

We will assemble a library of small parsers and operations to combine them.

First, a parser that doesn't consume any input:



It's going to get tedious to write `String → [(String, a)]`, so we'll define a datatype for us:

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

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

Using this datatype, we need to define our parsers slightly differently:

```
produce :: a → Parser a
produce x = Parser (λts → [(ts, x)])
```

Note: $f = \lambda x \rightarrow f\ x$

Example:

$f\ x = x * x$

$\equiv$

$f = \lambda x \rightarrow x * x$ (an anonymous function)

In other words, we could have defined produce as:

```
produce :: a → Parser a
produce x = Parser (produce' x)
```

Using a λ allows us to not define the produce', but instead use its body directly in the definition of produce.

So concretely:

```
produce' x ts = [(ts, x)]
```

$\equiv$

```
produce' x = λts → [(ts, x)]
```

We can write:

```
parse (produce 42) :: String → [(String, Int)]
```

So:

```
parse (produce 42) "hello" = [("hello", 42)]
```

Another parser is the one that fails all the time:

```
failure :: Parser a
failure = Parser (λts → [])
```

We can use this as follows:

```
    parse failure "hello" = []
= { def failure }
    parse (Parser (λts → [])) "hello"
= { def parse }
    (λts → []) "hello"
= { def λ }
    []
```