Language Engineering	Lecture 13

```
item :: Parser Char

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

The following parser checks to see if the first character on the input satisfies some predicate:

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do  c <- item
                if p c
                   then produce c
                   else fail
```

This is syntactic sugar for the following equivalent definition:

```
satisfy' :: (Char -> Bool) -> Parser Char
satisfy' p = item >>= \c ->
             if p c
                then produce c
                else fail
```

With satisfy, we have the building blocks for lots of parsing.

```
char :: Char -> Parser Char
char c = satisfy (c ==)
```

NOTE: $(c==) :: Char \to Bool$
and is defined by:

$$(c ==)\,x = c == x$$

This parser parses a single Char and returns it if it succeeds.

Now we can use this to write a parser that recognises strings:

```
string :: String -> Parser String
string [] = produce []
string (c:cs) = do char c
                   string cs
                   produce (c:cs)
```

This definition is syntactic sugar for:

```
string (c:cs) = char c >>= λ c' ->
                string cs >>= λ cs' ->
                produce (c:cs)
```

Another definition which will be more like parsing grammars in general is the following:

```
string :: ~~Pars~~ String -> Parser String
string cs = foldr (\x pxs ->
                (:) <$> char x <*> pxs )
            (produce []) cs
```

parses a char

parses the string xs

This is the derivation:

given $cs = c_0 : c_1 : c_2 : c_3 : []$

string cs

$c_0 \quad c_1 \quad c_2 \quad c_3 \quad []$

```
(:) <$> char c0 <*> ...
... (:) <$> char c2 <*> ((:) <$> char c3 <*> (produce [])))
```