

Administrivia

Assignment 7 is due Friday by 11:59PM.

There will be no discussion sections this week.

Parsing II: Combinators

Principles of Programming Languages

Lecture 16

Objectives

Introduce *parser combinators* as a way to build complex parsers in a simple and composable way.

See several *examples* to get some practice.

Keywords

Practice Problem

$\langle \text{expr} \rangle$	$::=$	$\langle \text{expr2} \rangle$	$ $	$\langle \text{expr2} \rangle + \langle \text{expr} \rangle$
$\langle \text{expr2} \rangle$	$::=$	x	$ $	$(\langle \text{expr} \rangle)$

Convert the above grammar to Chomsky Normal Form, i.e., find a grammar which recognizes the same sentences and only has rules of the form

$\langle \text{start} \rangle ::= \epsilon$
 $\langle \text{non-term} \rangle ::= \langle \text{non-term1} \rangle \langle \text{non-term2} \rangle$
 $\langle \text{non-term} \rangle ::= \text{term}$

Parser Combinators

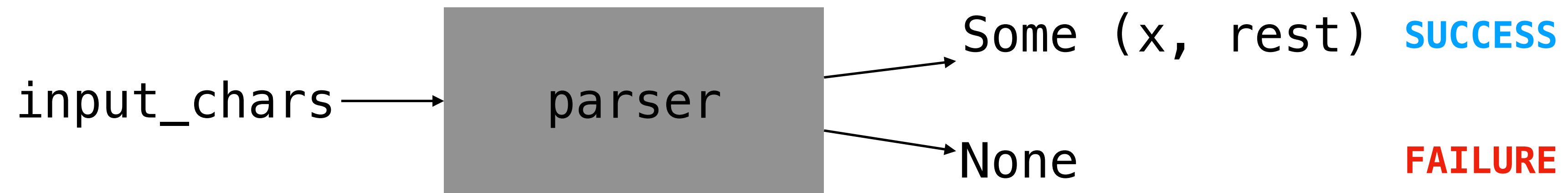
What is a parser?

```
type 'a parser =  
char list -> ('a * char list) option
```

*A **parser** is a function which takes a list of characters and consumes part of the beginning of it, converting it into an 'a, and returning the rest.*

*A parser **fails** if it returns **None**.*

The Picture



*If we think of these parsers as **black boxes** can we write functions which combine them and compose them in useful ways?*

(this is the game we will be playing)

An Aside: Imperative Parsing

call 1: next_token()

call 2: next_token()

call 3: next_token()

global state

```
input = "1 + 2"  
out = []
```

global state

```
input = "+ 2"  
out = [Tok 1]
```

global state

```
input = "2"  
out = [Tok 1, AddT]
```

In an imperative language a parser is of type **string** \rightarrow 'a, where the input is ***global***.

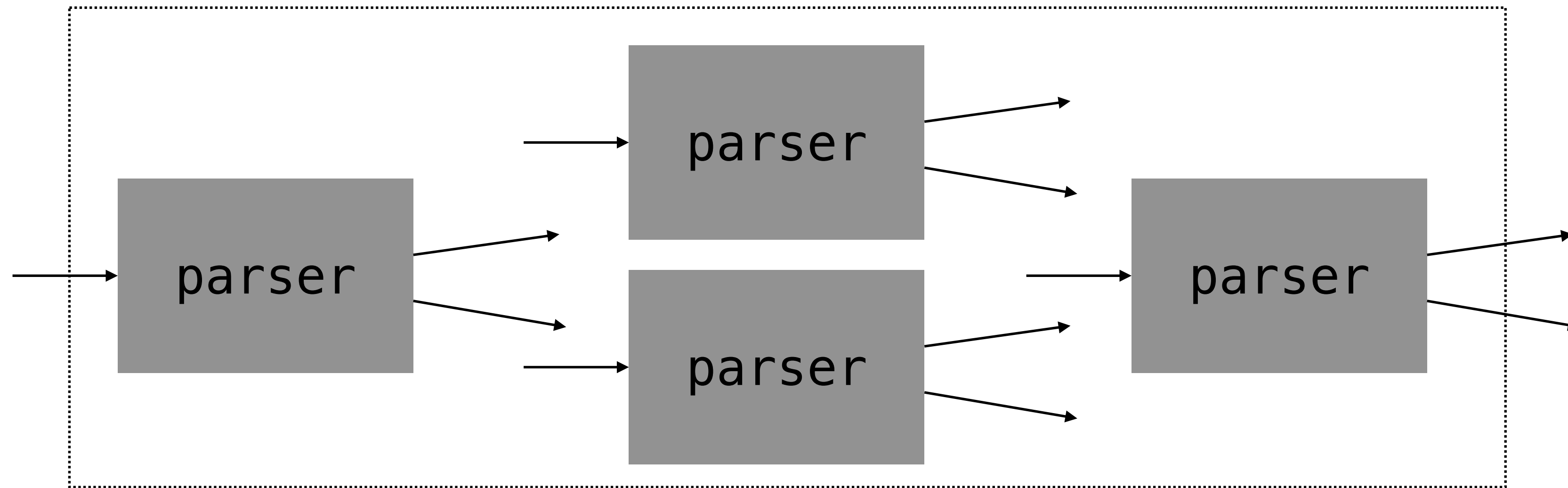
In a functional language we ***carry around the state information***.

How do we use a parser?

```
let parse
  (p : 'a parser)
  (s : string) : 'a option =
  match p (explode s) with
  | Some (t, []) -> Some t
  | _ -> None
```

Apply it to a (exploded) string and check if everything has been consumed.

The Big Picture



Rather than writing ad hoc code to parse, we'll build a small ***interface*** for composing and transforming existing parsers.

This small interface will be powerful enough to cover most parsing tasks we need to handle (and will be based on some very interesting theory).

Parser Combinators

Combinators

```
let flip f y x = f x y
let const x y = x
let id x = x
```

A **combinator** is just a higher-order function.

We define a basic collection of combinators and then define other functions as ***combinations*** of combinators.

Basic Input Reading

```
let char (c : char) : unit parser = function
| d :: cs -> if c = d then Some ((), cs) else None
| _ -> None
```

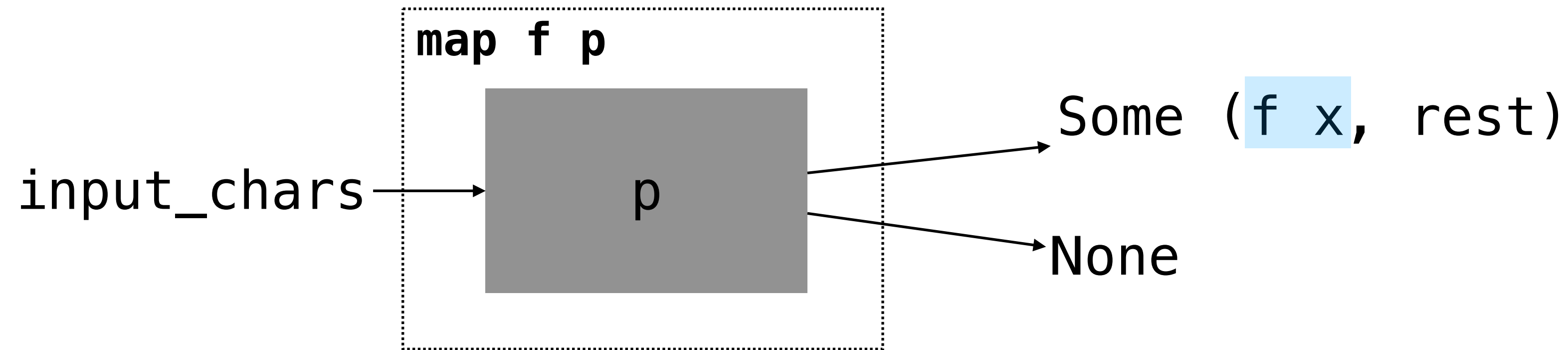
We need some basic combinators to **consume** parts of a list of characters.

This one just checks the first character, and fails if it does not match the input character.

demo

(basic parser combinators in OCaml)

Mapping



*Convert **p** into the same parser but with a function applied to its output value.*

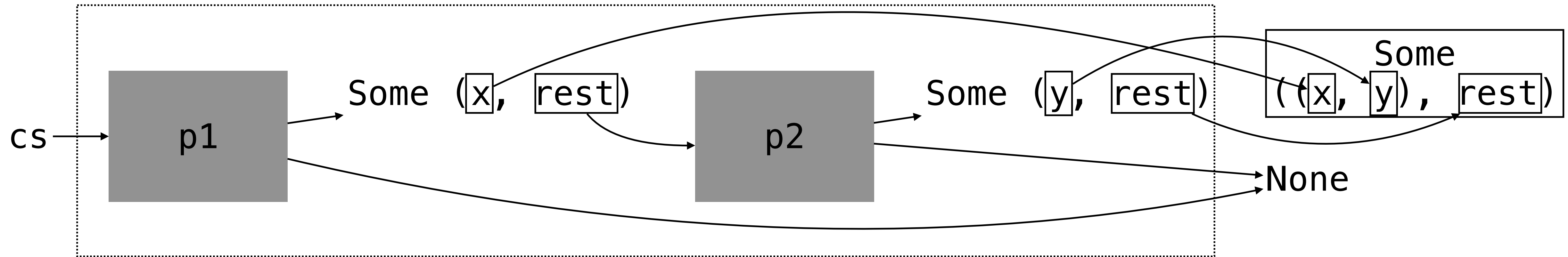
demo

(mapping in OCaml)

Mapping

```
let map (f : 'a -> 'b) (p : 'a parser) : 'b parser =  
  fun cs ->  
    Option.map (fun (s, rest) -> (f s, rest)) (p cs)  
  
let (>|=) p f = map f p
```

Sequencing



*Run **p1** and then **p2**. Combine their results, and fail if either one fails.*

demo

(sequencing in OCaml)

Sequencing

```
let seq (p1 : 'a parser) (p2 : 'b parser) : ('a * 'b) parser = fun cs ->
  match p1 cs with
  | None -> None
  | Some (a, rest) ->
    match p2 rest with
    | None -> None
    | Some (b, rest) -> Some ((a, b), rest)

let (<<) p1 p2 = map fst (seq p1 p2)
let (>>) p1 p2 = map snd (seq p1 p2)
```

Bootstrapping mapping and sequencing

```
let seq2    p1 p2 = seq p1 p2
let map2 f p1 p2 = map (fun (x, y) -> f x y) (seq p1 p2)

let seq3    p1 p2 p3 = map2 (fun a (b, c) -> (a, b, c)) p1 (seq p2 p3)
let map3 f p1 p2 p3 = map (fun (a, b, c) -> f a b c) (seq3 p1 p2 p3)

let seq4    p1 p2 p3 p4 = map3 (fun a b (c, d) -> (a, b, c, d)) p1 p2 (seq p3 p4)
let map4 f p1 p2 p3 p4 = map (fun (a, b, c, d) -> f a b c d) (seq4 p1 p2 p3 p4)
```

We can map over two arguments after sequencing, then three then four...

These combinators will be useful for implementing ***mixfix operators***.

Understanding Check

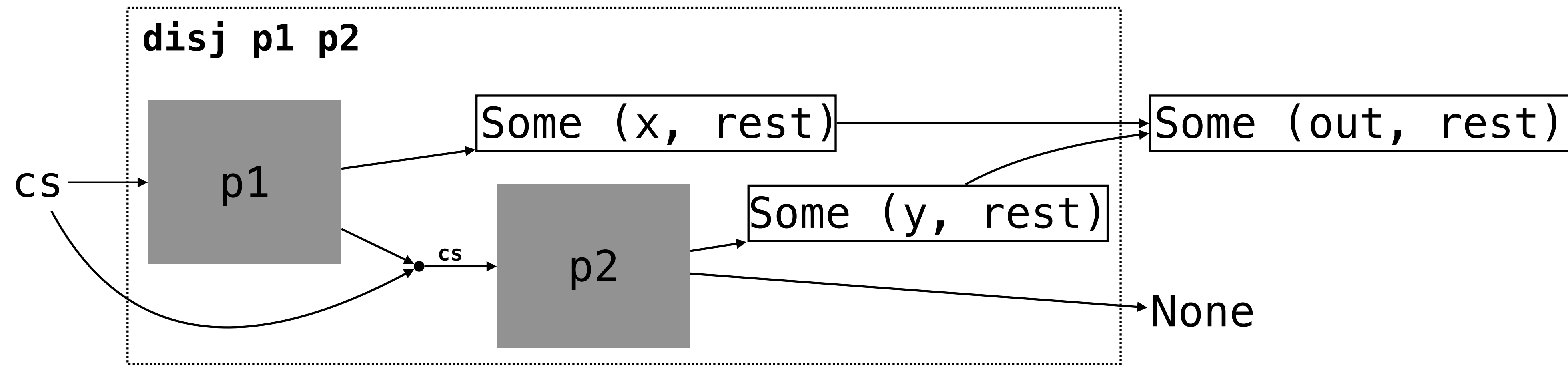
Write the parser combinator **between**, which given

```
» opn : 'a parser  
» cls : 'a parser  
» p   : 'b parser
```

returns the **'b parser** which

```
» runs opn, ignoring the output  
» runs p, keeping the output  
» runs cls, ignoring the output
```

Alternatives



*Try running **p1** and returning its output.*

*If **p1** fails, run **p2** and return its output.*

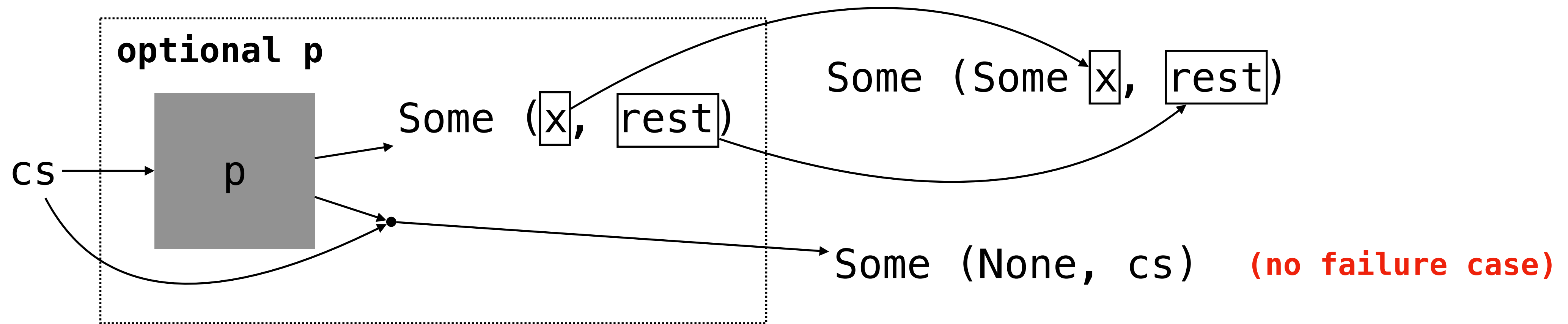
demo

(alternatives in OCaml)

Alternatives

```
let disj (p1 : 'a parser) (p2 : 'a parser) : 'a parser = fun cs ->
  match p1 cs with
  | Some (x, rest) -> Some (x, rest)
  | None ->
    match p2 cs with
    | Some (x, rest) -> Some (x, rest)
    | None -> None
```

Optionals



*Run **p**. If it succeeds then return the output value in the **Some** constructor.*

*If it fails return **None** without consuming anything.*

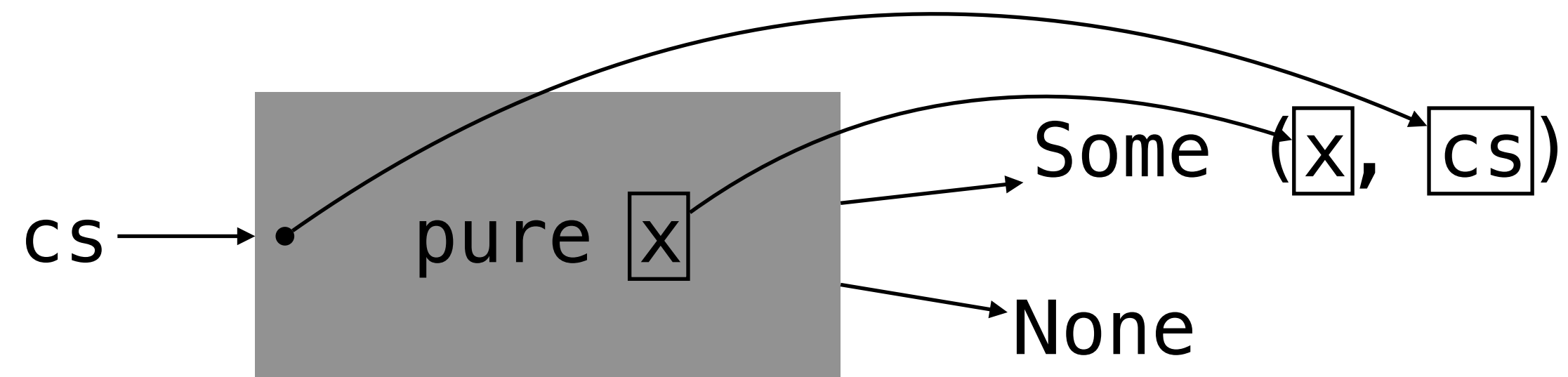
demo

(optionals in OCaml)

Optionals

```
let optional (p : 'a parser) : 'a option parser =  
  fun cs ->  
    match p cs with  
    | Some (x, rest) -> Some (Some x, rest)  
    | None -> Some (None, cs)
```

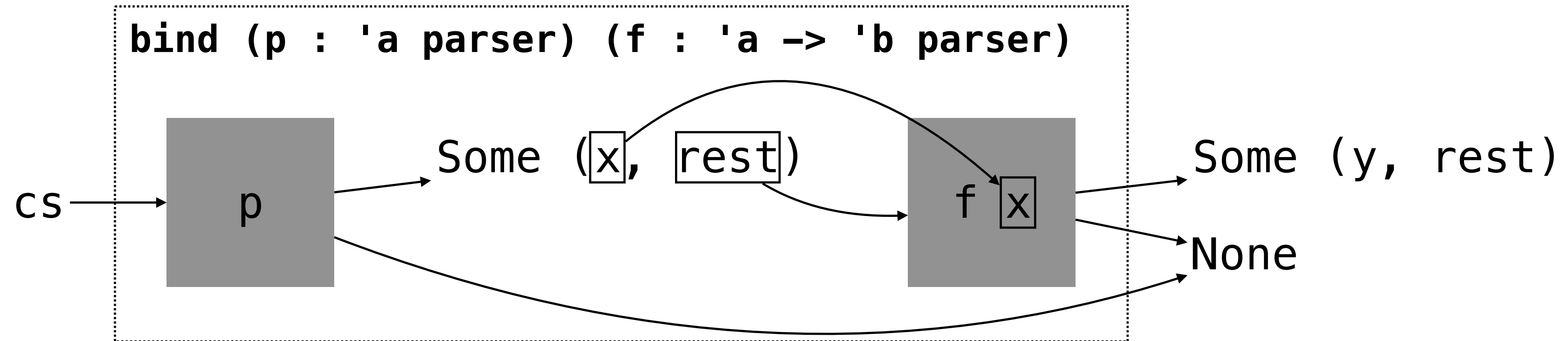
Pure and Fail



***pure** returns its argument and consumes nothing.*

***fail** always returns **None**.*

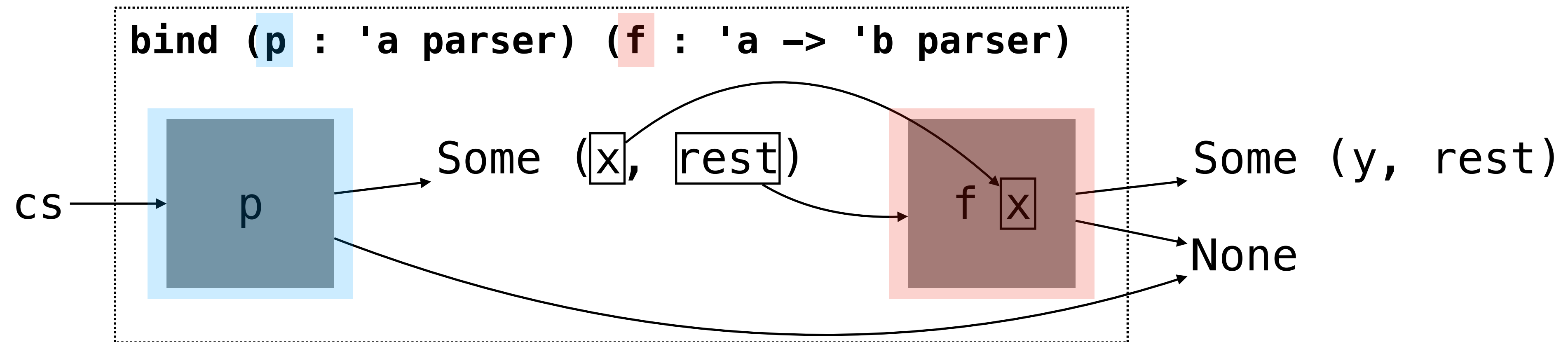
Bind



*Use the output of **p** to build a new parser from **f** and run that after running **p**.*

Fail if either one fails.

Bind



*Use the output of **p** to build a new parser from **f** and run that after running **p**.*

Fail if either one fails.

demo

(pure, fail, and bind in OCaml)

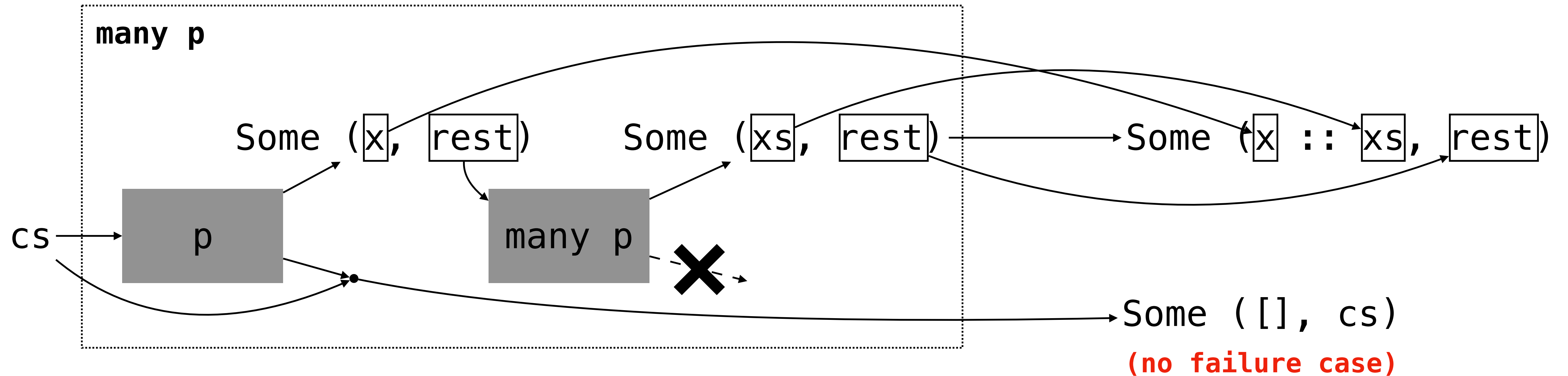
Pure, Fail, Bind

```
let pure (x : 'a) : 'a parser = fun cs -> Some (x, cs)

let fail : unit parser = fun _ -> None

let bind (p : 'a parser) (f : 'a -> 'b parser) : 'b parser =
  fun cs ->
    match p cs with
    | Some (x, rest) -> f x rest
    | None -> None
```

Repetition



Run p as many times as possible. Collect all the successes in a list and return it.

demo

(repetition in OCaml)

Repetition

```
let rec many (p : 'a parser) : 'a list parser = fun cs ->
  match p cs with
  | None -> Some ([], cs)
  | Some (x, rest) ->
    match many p rest with
    | None -> Some ([x], rest)
    | Some (xs, rest) -> Some (x :: xs, rest)
```

A Note on Whitespace

As we design more complex parsers, we need to deal with whitespace. A couple tips.

- » We will typically consume our whitespace ***after*** consuming the relevant part of a string.
- » We will typically wait until the ***last layer*** of combinators to deal with whitespace.
- » It is generally good practice to ***avoid polluting*** your code with a bunch of whitespace removing (e.g., don't remove whitespace both in front of and behind a term).

An Aside: Relation to EBNF grammars

<code><s> ::= term</code>	<code>str "term"</code>	terminal
<code><s> ::= <p1> <p2></code>	<code>seq p1 p2</code>	sequence
<code><s> ::= <p1> <p2></code>	<code>disj p1 p2</code>	alternative
<code><s> ::= [<p>]</code>	<code>optional p</code>	optional
<code><s> ::= {<p>}</code>	<code>many t</code>	repetition

Understanding Check

*Implement the combinator **many1**, which is the same as **many** except that it fails if the list is empty.*

demo

(extended example: tokenizing)

demo

(extended example: expressions)

One Last Issue: Recursive Parsers

this does not work...

```
let rec many p =  
  map2 (fun x xs -> x :: xs) p (many p) <|> pure []
```

If we want to handle things like parentheses, we have to write recursive parsers like this one.

As we've presented things so far, *we can't use recursion.*

Thunks

```
let thunk () : int = 5

let thunked_parser x : unit -> 'a parser =
  fun () -> pure x
```

A **thunk** is a function which takes a **unit** and returns a value.

If it returns a value that is computationally expensive, then we can ***put off*** computing it until we need it.

Advanced: Monads and Bind

```
let many p =  
  let rec go () = elems () <|> pure []  
  and elems () =  
    let* x = p in  
    let* xs = go () in  
    pure (x :: xs)  
  in go ()
```

Another way to write recursive parsers is to use **bind**.

There is special syntax for working with sequences of binds.

Advanced: Monads and Bind

```
let many p =  
  let rec go () = elems () <|> pure []  
  and elems () =  
    let* x = p in      if p succeeds,  
    let* xs = go () in make x its output  
    pure (x :: xs)  
in go ()
```

Another way to write recursive parsers is to use **bind**.

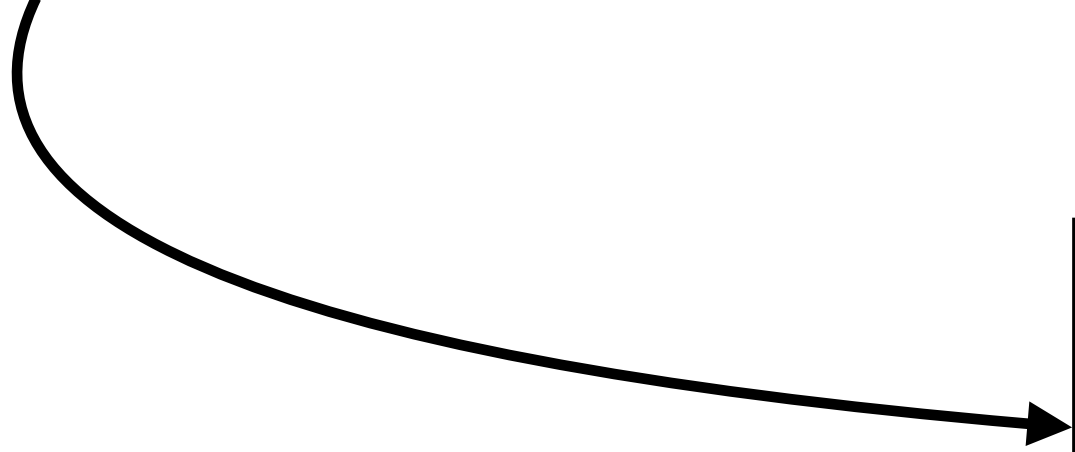
There is special syntax for working with sequences of binds.

last demo

(extended example: expressions with parentheses)

When in doubt...

```
let rec parse_expr : expr parser =  
  (* some code *)  
  keyword "(" >> parse_expr () << keyword ")"  
  (* some code *)
```



```
let rec parse_expr () : expr parser =  
  (* some code *)  
  let* _ = pure () in  
    keyword "(" >> parse_expr () << keyword ")"  
  (* some code *)
```

If you're stuck and you just want to write recursive parser:

- » Turn it into a thunk
- » wrap recursive calls in a **bind**

A Practical Note

There's a lot we're missing, including basic things like *reading in files*.

We'll get *practice with combinators* in the next assignment.

In the projects, we'll be *very clear* about what combinators will be useful.