Parsing with FParsec

a

Parser

is a program

that interprets a stream of symbols

using a specified grammar

- Natural language
- Computer languages
- File formats
- git committish

Formal Language Theory

(in five minutes)

/regex/

```
(\(\d{3}\) )?(\d{3})-(\d{4})
```

/regex/

```
(\(\d{3}\) )?(\d{3})-(\d{4})
```

/regex/

```
(?i)\b((?:[a-z][\w-]+:(?:/{1,3}|[a-z0-9%])|www\d{0,3}[.]|[a-z0-9.\-]+
[.][a-z]{2,4}/)(?:[^\s()<>]+|\(([^\s()<>]+|(\([^\s()<>]+\)))*\))+
(?:\(([^\s()<>]+|(\([^\s()<>]+\)))*\)|[^\s`!()\[\]{};:'".,<>?«»“”‘’]))
```

http://www.flickr.com/photos/theclyde/2819332618/

```
/* Infix notation calculator.  */

%{
  #include <math.h>
  #include <stdio.h>
  int yylex (void);
  void yyerror (char const *);
%}

/* Bison declarations.  */
%define api.value.type {double}
%token NUM
%left '-' '+'
%left '*' '/'
%precedence NEG  /* negation--unary minus */
%right '^'       /* exponentiation */
```

```
%% /* The grammar follows.  */
 input:
   %empty
 | input line
 ;
 line:
   '\n'
 | exp '\n' { printf ("\t%.10g\n", $1); }
 ;

 exp:
   NUM                 { $$ = $1;   }
 | exp '+' exp         { $$ = $1 + $3;  }
 | exp '-' exp         { $$ = $1 - $3;  }
 | exp '*' exp         { $$ = $1 * $3;  }
 | exp '/' exp         { $$ = $1 / $3;  }
 | '-' exp  %prec NEG  { $$ = -$2;  }
 | exp '^' exp         { $$ = pow ($1, $3); }
 | '(' exp ')'         { $$ = $2;   }
;
%%
```

```
/* from http://www.gnu.org/software/bison/
       manual/html_node/Infix-Calc.html */
```

```
/* Infix notation calculator.  */

%{
  #include <math.h>
  #include <stdio.h>
  int yylex (void);
  void yyerror (char const *);
%}

/* Bison declarations.  */
%define api.value.type {double}
%token NUM
%left '-' '+'
%left '*' '/'
%precedence NEG  /* negation--unary minus */
%right '^'       /* exponentiation */
```

```
%% /* The grammar follows.  */
 input:
   %empty
 | input line
 ;
 line:
   '\n'
 | exp '\n' { printf ("\t%.10g\n", $1); }
 ;

 exp:
   NUM                 { $$ = $1;   }
 | exp '+' exp         { $$ = $1 + $3;  }
 | exp '-' exp         { $$ = $1 - $3;  }
 | exp '*' exp         { $$ = $1 * $3;  }
 | exp '/' exp         { $$ = $1 / $3;  }
 | '-' exp  %prec NEG  { $$ = -$2;  }
 | exp '^' exp         { $$ = pow ($1, $3); }
 | '(' exp ')'         { $$ = $2;   }
 ;
%%
```

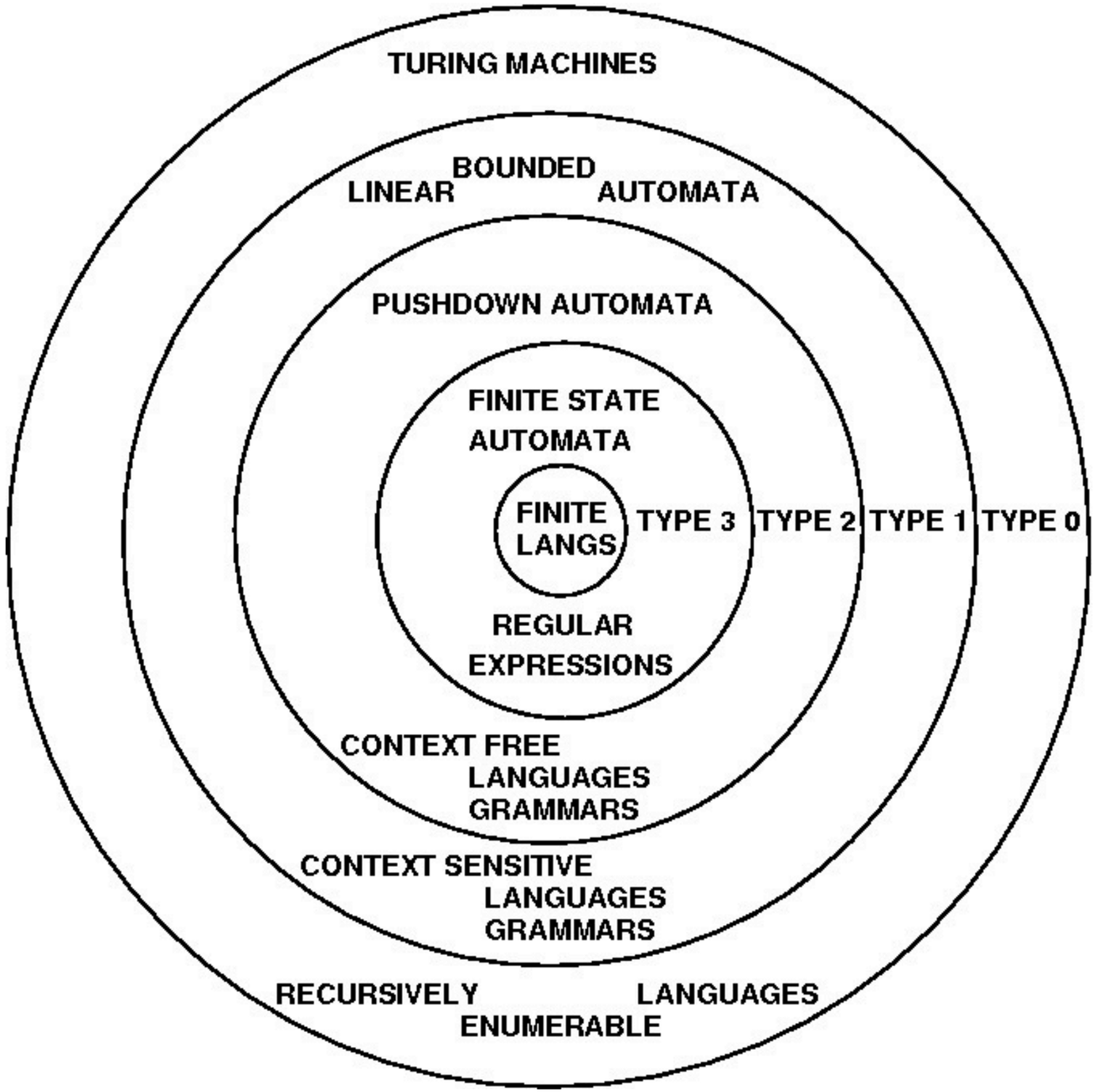
```
/* from http://www.gnu.org/software/bison/
       manual/html_node/Infix-Calc.html */
```

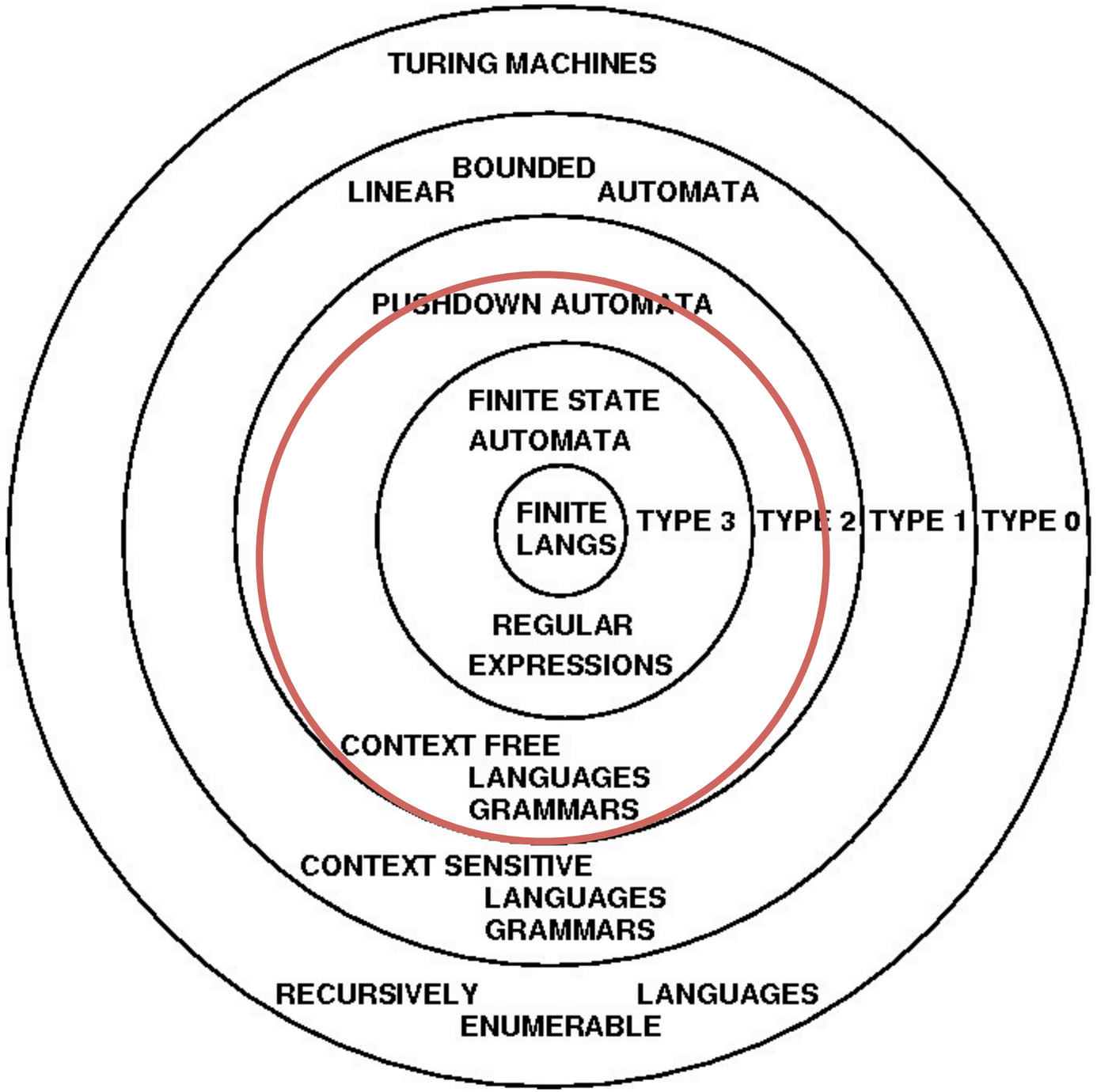
```
/* Infix notation calculator.  */

%{
  #include <math.h>
  #include <stdio.h>
  int yylex (void);
  void yyerror (char const *);
%}

/* Bison decl
%define api.v
%token NUM
%left '-' '+'
%left '*' '/'
%precedence N                                   /
%right '^'      /* exponentiation *
```

BNF

```
%% /* The grammar follows.  */
 input:
   %empty
 | input line
 ;
 line:
   '\n'
 | exp '\n' { printf ("\t%.10g\n", $1); }
 ;

 exp:
   NUM                 { $$ = $1;   }
 | exp '+' exp         { $$ = $1 + $3;  }
 | exp '-' exp         { $$ = $1 - $3;  }
 | exp '*' exp         { $$ = $1 * $3;  }
 | exp '/' exp         { $$ = $1 / $3;  }
 | '-' exp  %prec NEG  { $$ = -$2;  }
 | exp '^' exp         { $$ = pow ($1, $3); }
 | '(' exp ')'         { $$ = $2;   }
 ;
%%
```

```
/* from http://www.gnu.org/software/bison/
       manual/html_node/Infix-Calc.html */
```


TURING MACHINES
BOUNDED
LINEAR
AUTOMATA
PUSHDOWN AUTOMATA
FINITE STATE
AUTOMATA
FINITE
LANGS
TYPE 3
TYPE 2
TYPE 1
TYPE 0
REGULAR
EXPRESSIONS
CONTEXT FREE
LANGUAGES
GRAMMARS
CONTEXT SENSITIVE
LANGUAGES
GRAMMARS
RECURSIVELY
LANGUAGES
ENUMERABLE


TURING MACHINES
BOUNDED
LINEAR
AUTOMATA
PUSHDOWN AUTOMATA
FINITE STATE
AUTOMATA
FINITE
LANGS
TYPE 3
TYPE 2
TYPE 1
TYPE 0
REGULAR
EXPRESSIONS
CONTEXT FREE
LANGUAGES
GRAMMARS
CONTEXT SENSITIVE
LANGUAGES
GRAMMARS
RECURSIVELY
LANGUAGES
ENUMERABLE

Parser Combinators

a

Parser

is a program

that interprets a stream of symbols

using a specified grammar

a

Parser

is a *function*

that interprets a stream of symbols

using a specified grammar

a

Combinator

is a function

that combines functions

a

Parser Combinator

is set of functions

that interprets a stream of symbols

by combining simple parsing functions

FParsec

is a parser combinator library

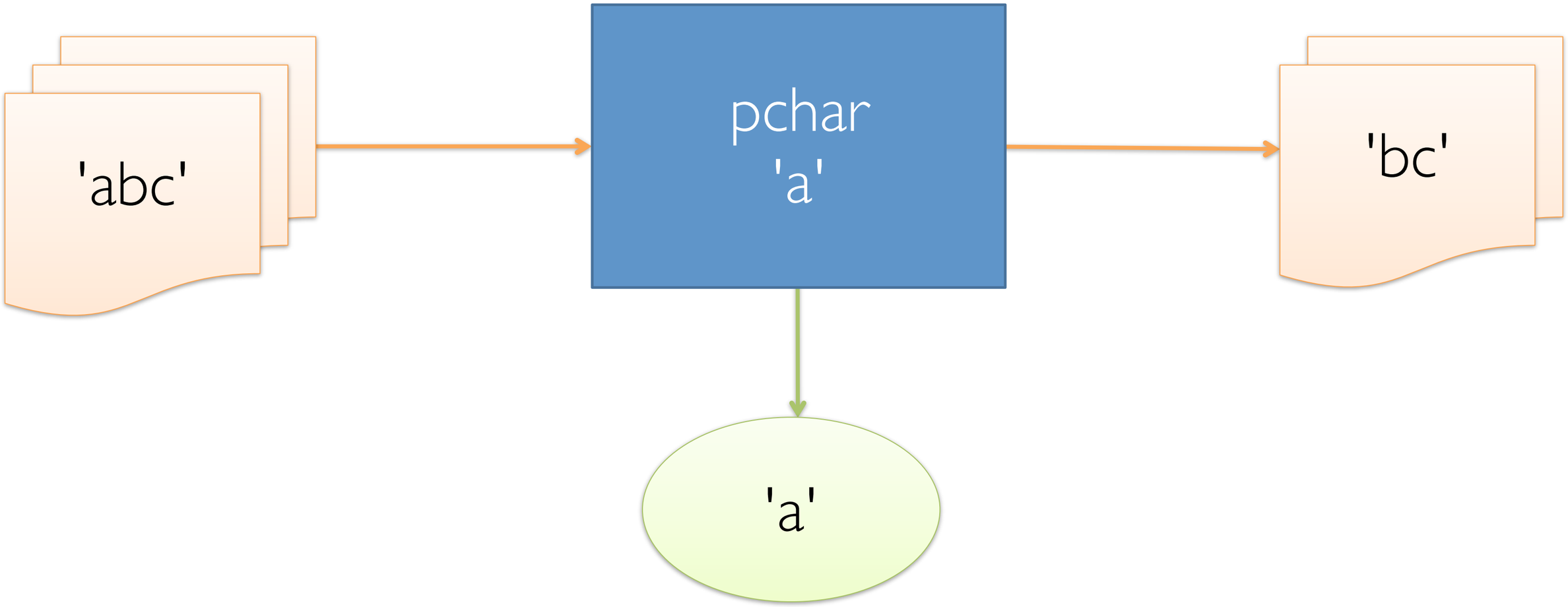
for F#

FParsec

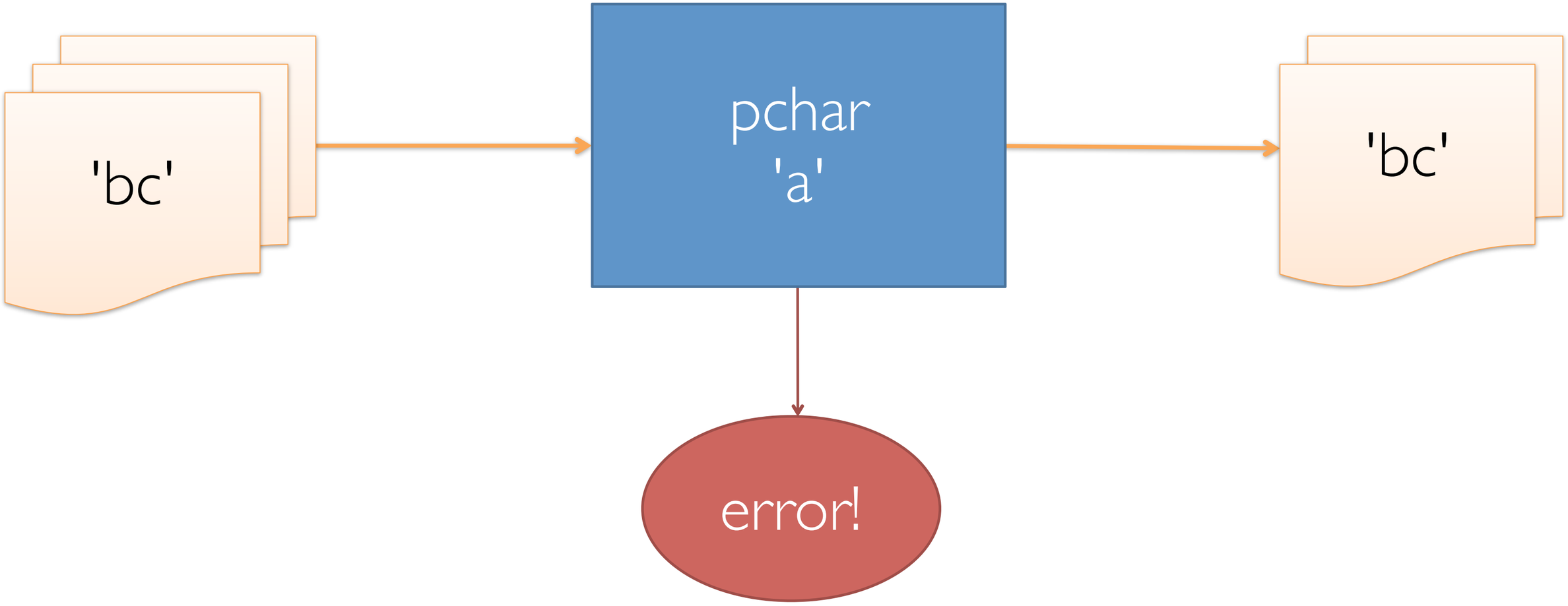
parses strings of characters

into F# objects

Read a Single Character



Read a Single Character



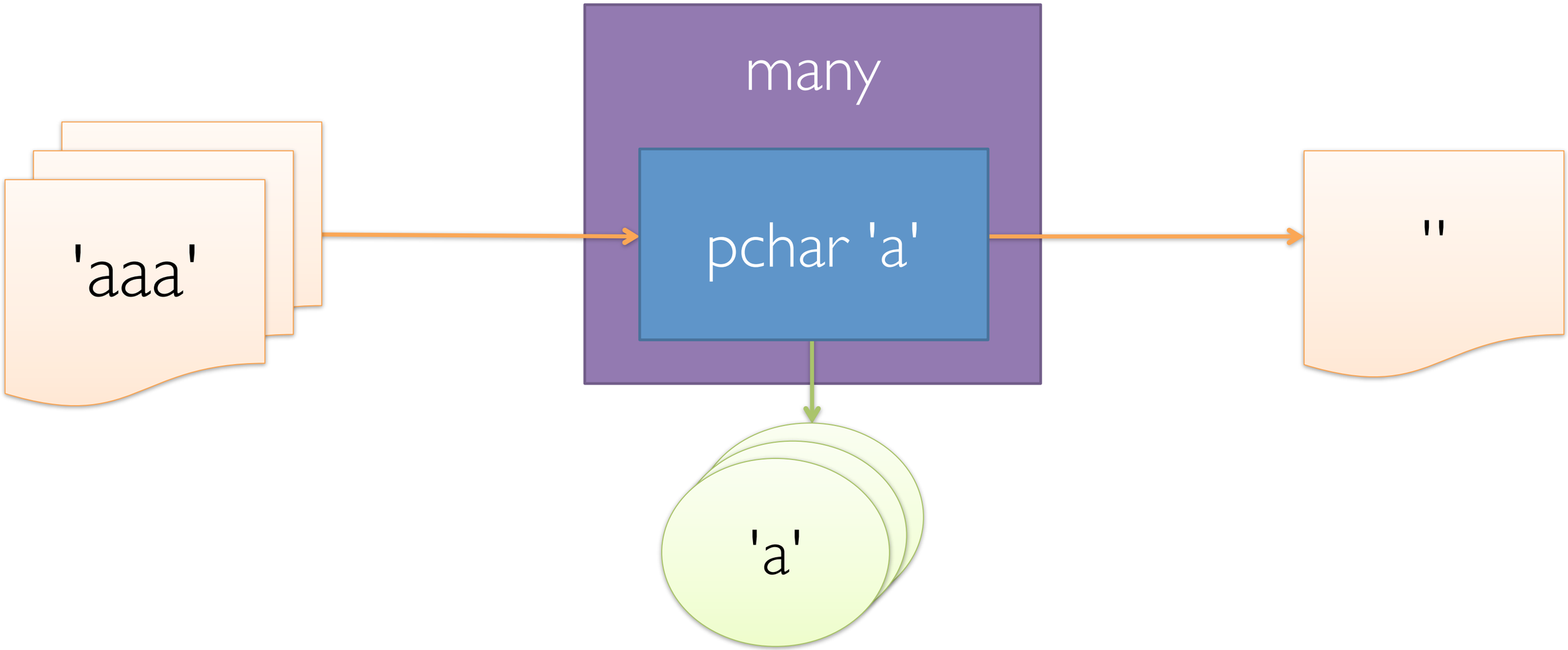
```
1.  open System
2.  open FParsec

3.  let go parser text =
4.    match run parser text with
5.    | Success (r, _, _) -> sprintf "Success! Parsed '%O'\n" r
6.    | Failure (m, _, _) -> sprintf "%O\n" m

7.  let main argv =
8.    let parser : Parser<char, unit> =
9.      pchar 'a'

10.   Console.WriteLine (go parser "abc")
11.   Console.WriteLine (go parser "bc")

12. main()
```

```
8.  let parser : Parser<char, unit> =
9.    pchar 'a'

10. Console.WriteLine (go parser "abc")
11. Console.WriteLine (go parser "bc")
```

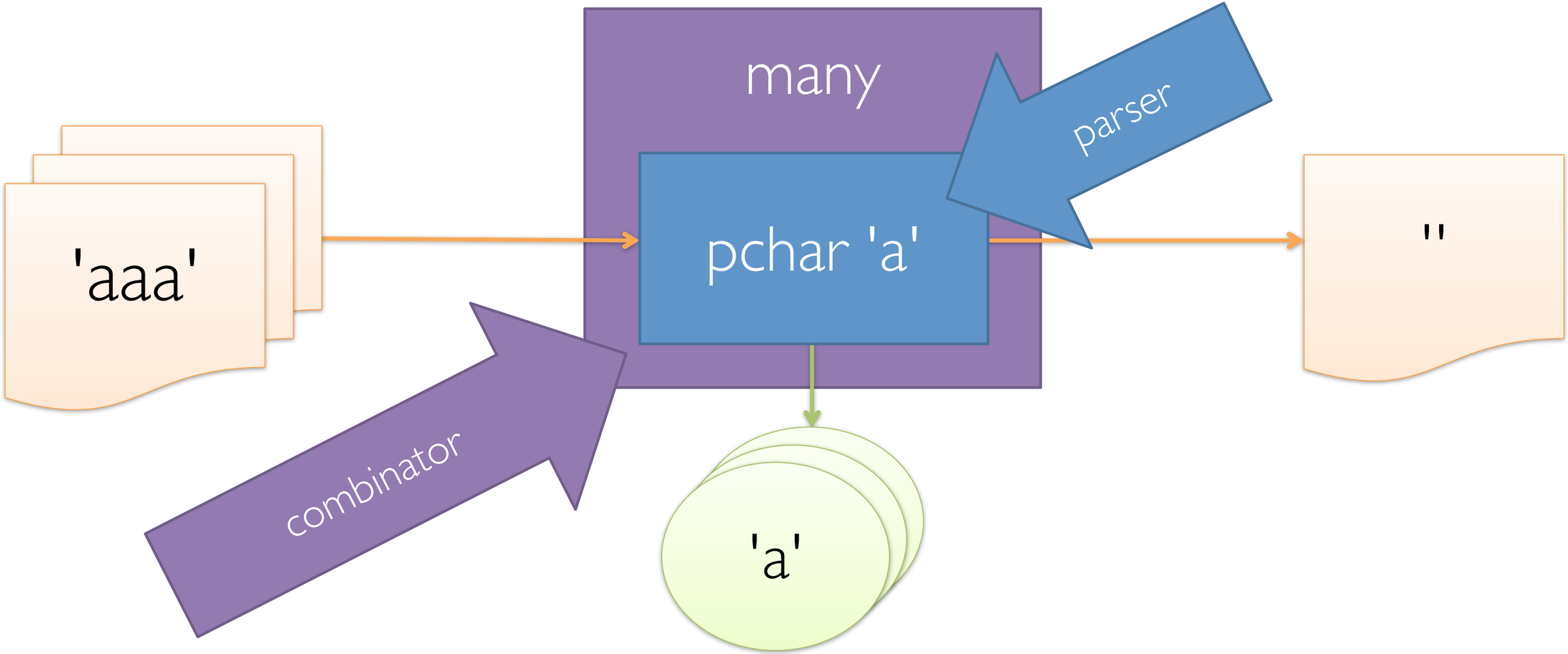
```
1.  Success! Parsed 'a'
```

```
2.  Error in Ln: 1 Col: 1
3.  bc
4.  ^
5.  Expecting: 'a'
```

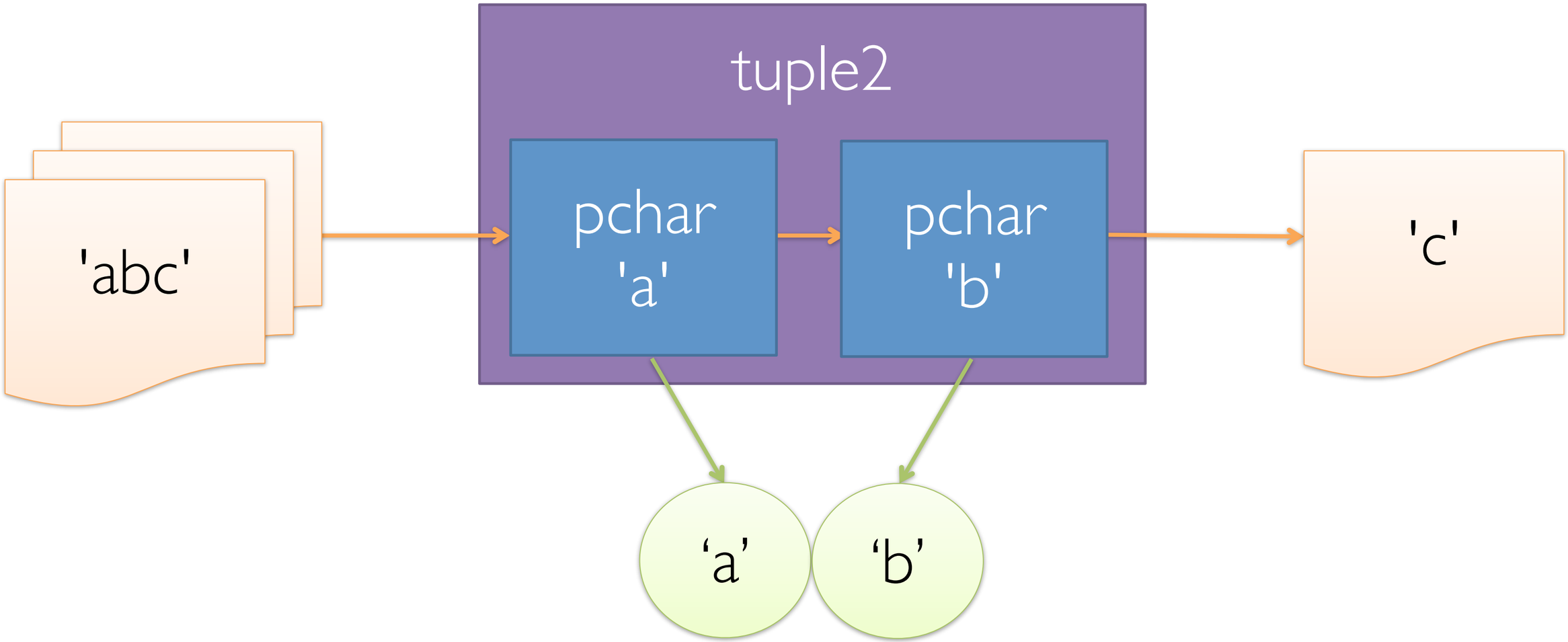
Read Multiple Characters



Read Multiple Characters



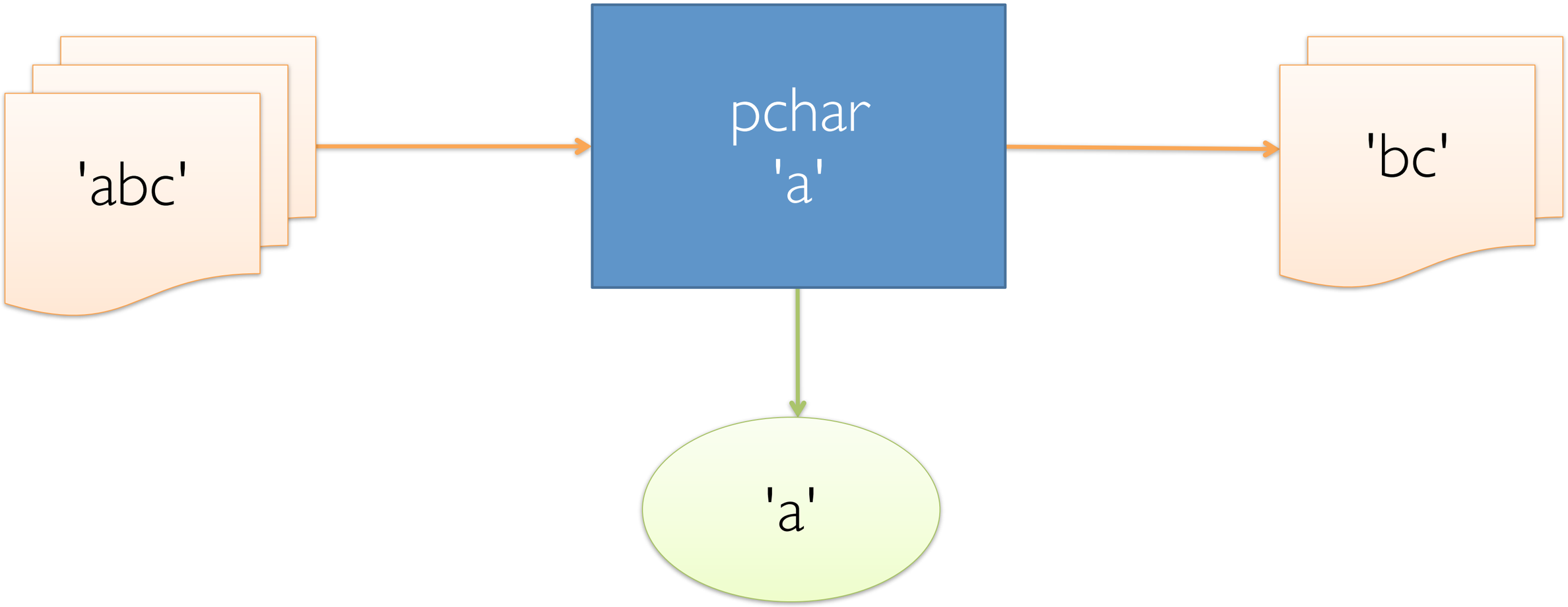
Read Multiple Characters



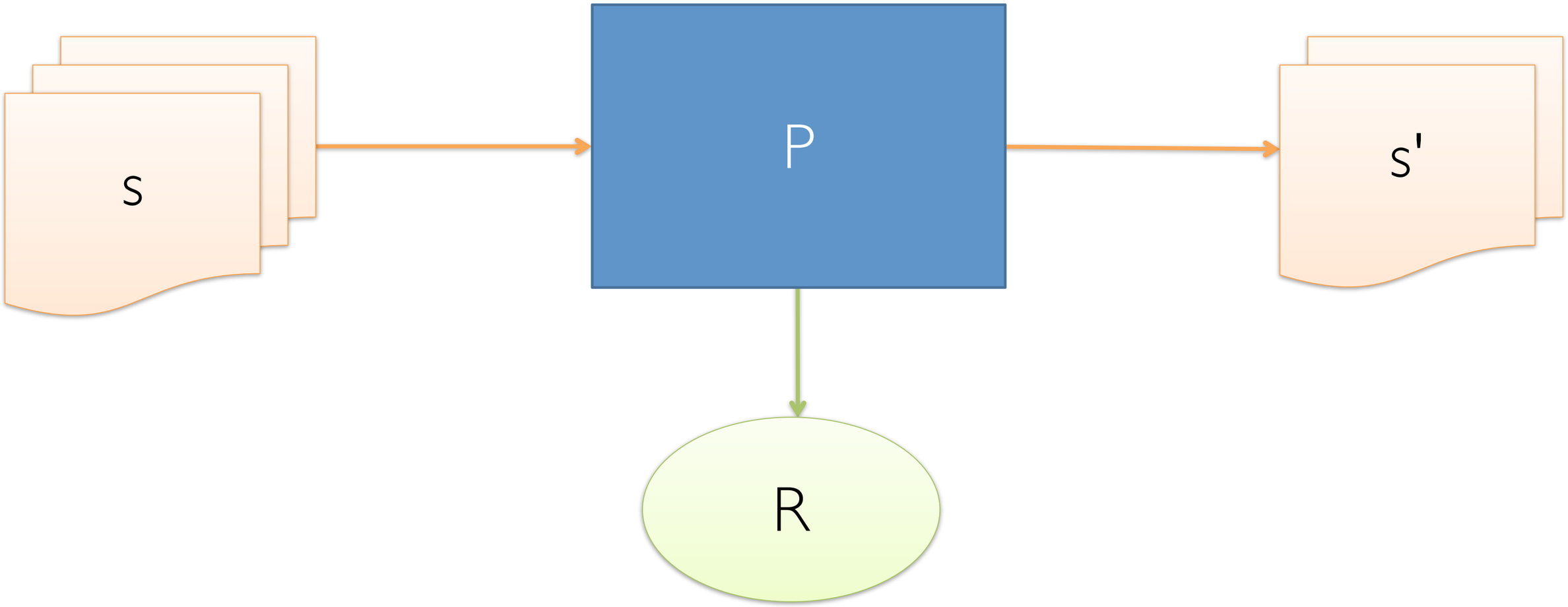
```
1.  let manyparser : Parser<char list, unit> =
2.    many (pchar 'a')

3.  let tupleparser : Parser<(char * char), unit> =
4.    (pchar 'a') .>>. (pchar 'b')

5.  Console.WriteLine (go manyparser "aaa")
6.  Console.WriteLine (go tupleparser "abc")
```

```
1.  Success! Parsed '[a; a; a]'

2.  Success! Parsed '(a, b)'
```

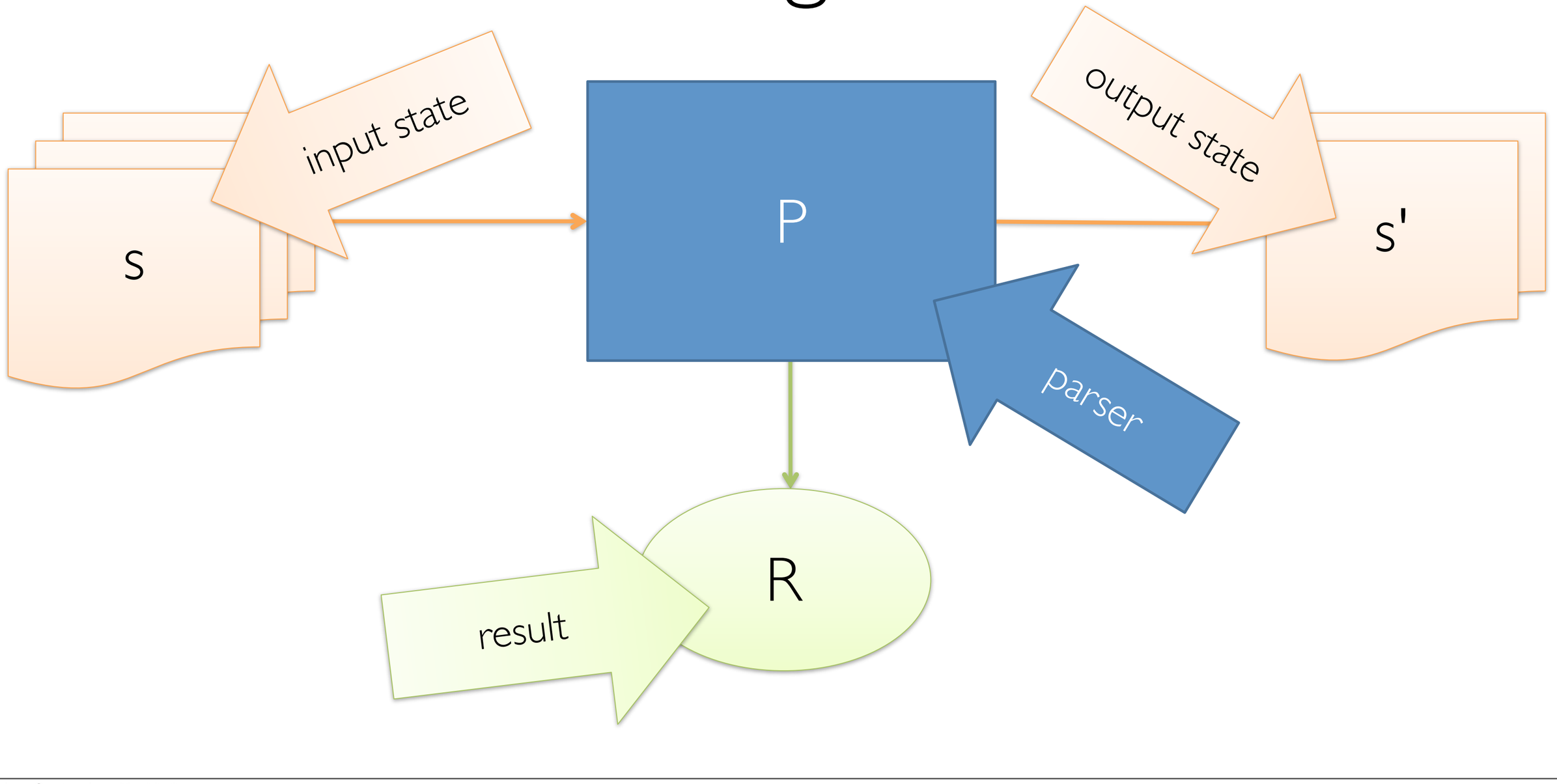
Read a Single Character



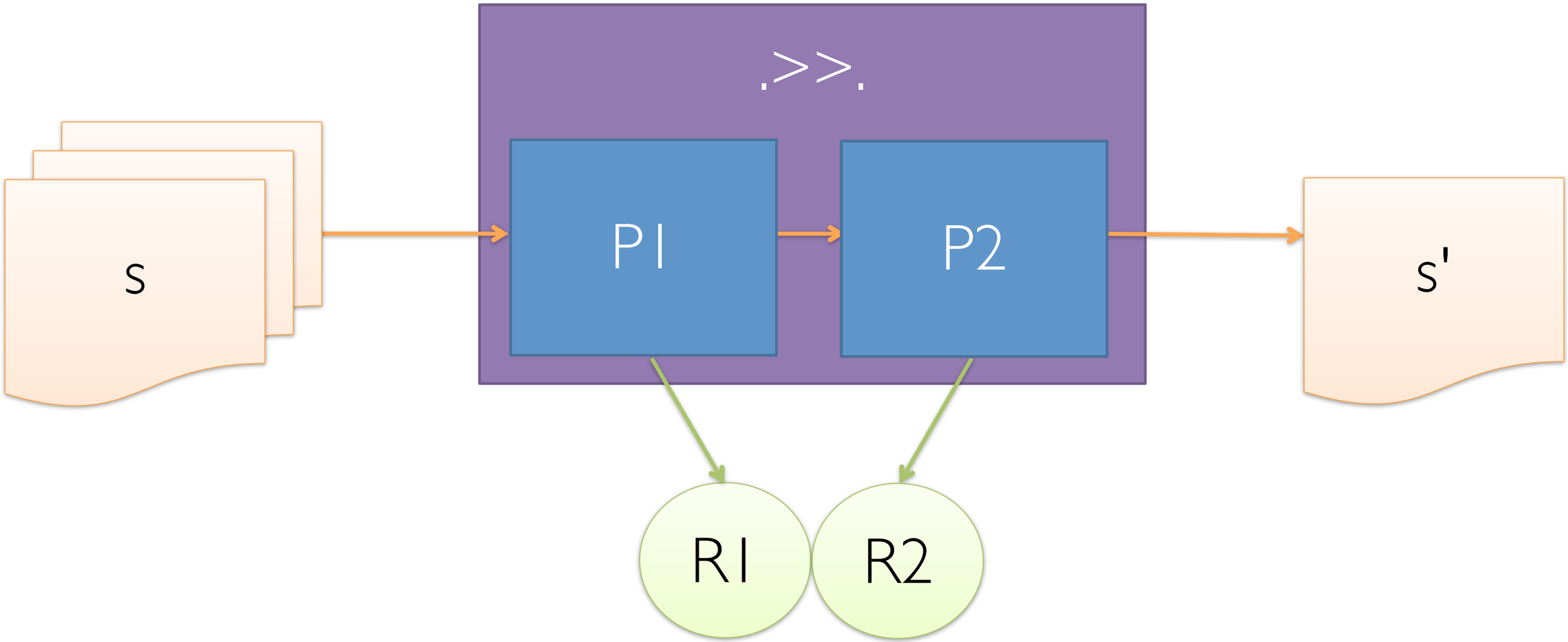
Run a Single Parser



Run a Single Parser



Run Multiple Parsers



applying a

Combinator

produces a

Parser

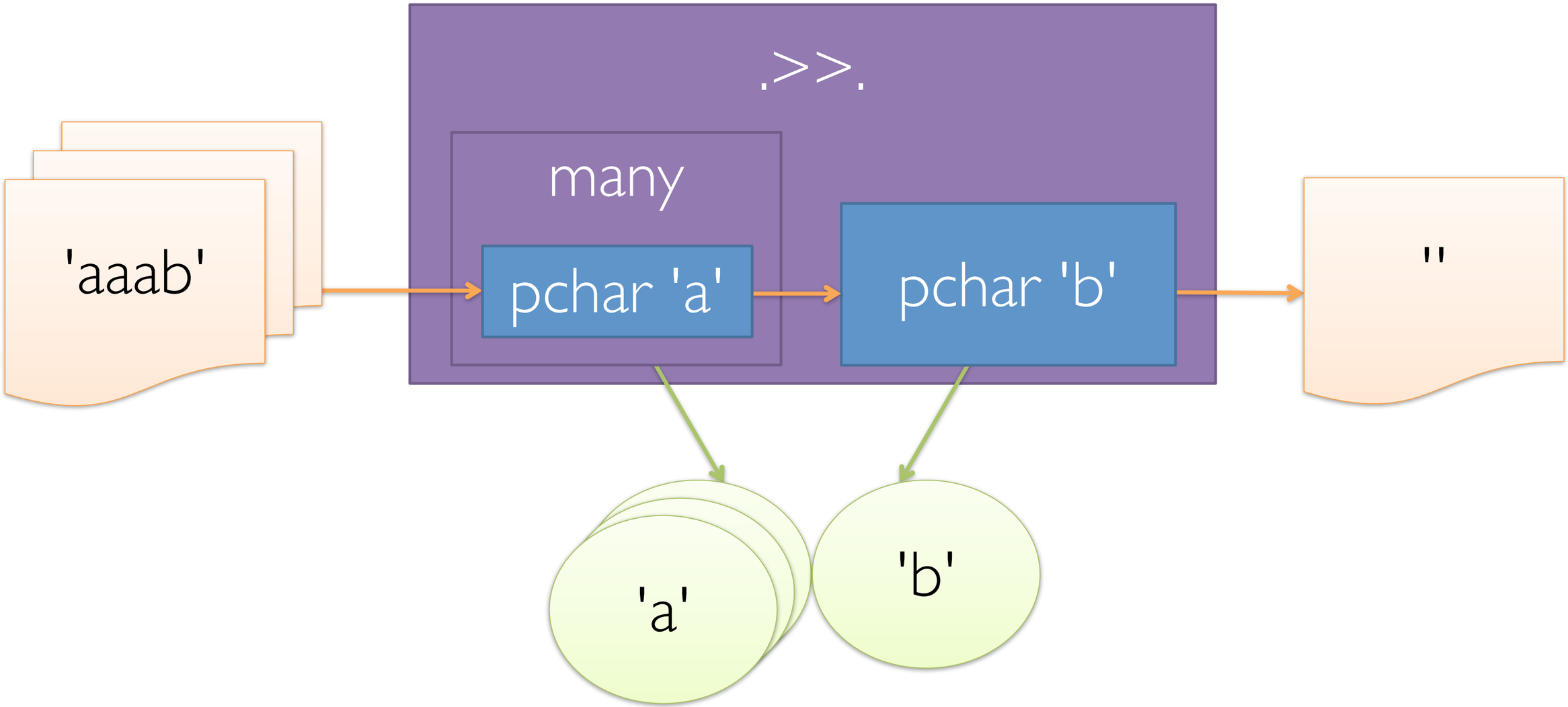
from one or more

parsers

```
1.  let manyparser : Parser<char list, unit> =
2.    many (pchar 'a')

3.  let tupleparser : Parser<(char list * char), unit> =
4.    manyparser .>>. (pchar 'b')

5.  Console.WriteLine (go tupleparser "aaab")
6.  Console.WriteLine (go tupleparser "ab")
```

```
1.  Success! Parsed '([a; a; a], b)'

2.  Success! Parsed '([a], b)'
```

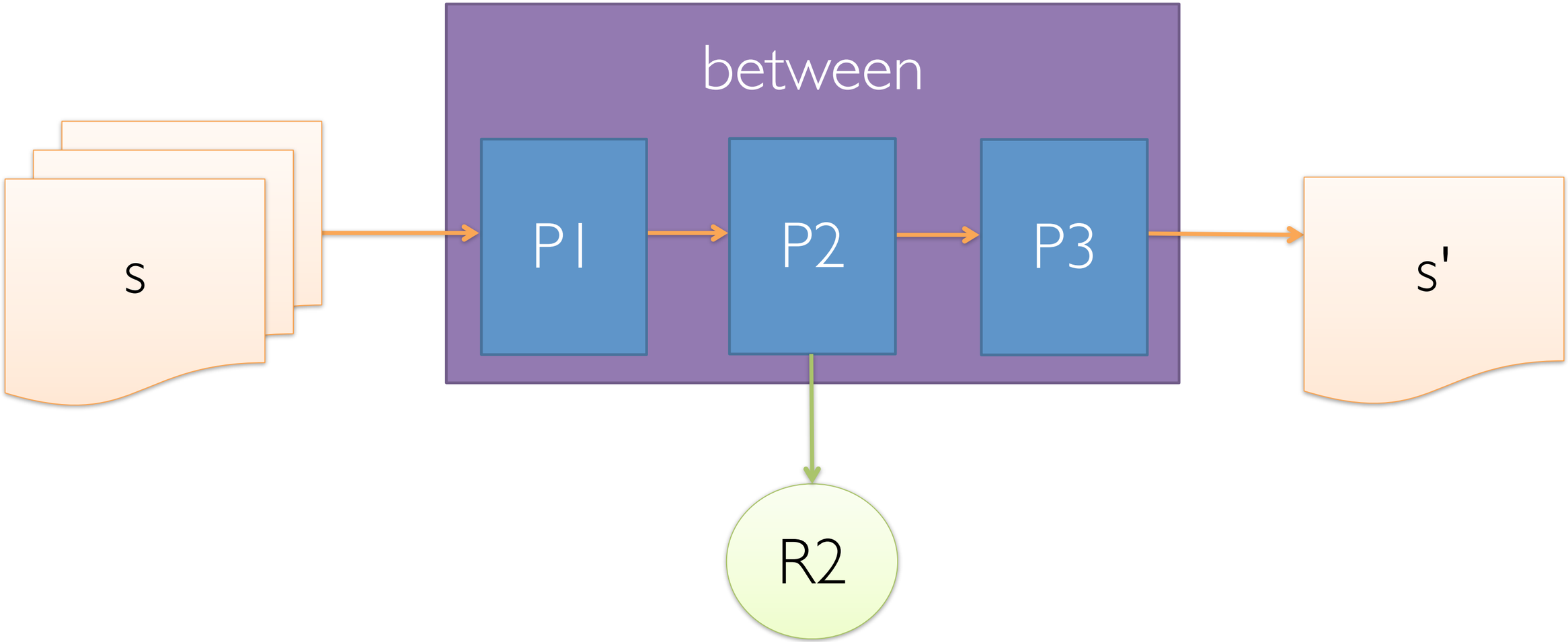
Run Multiple Parsers



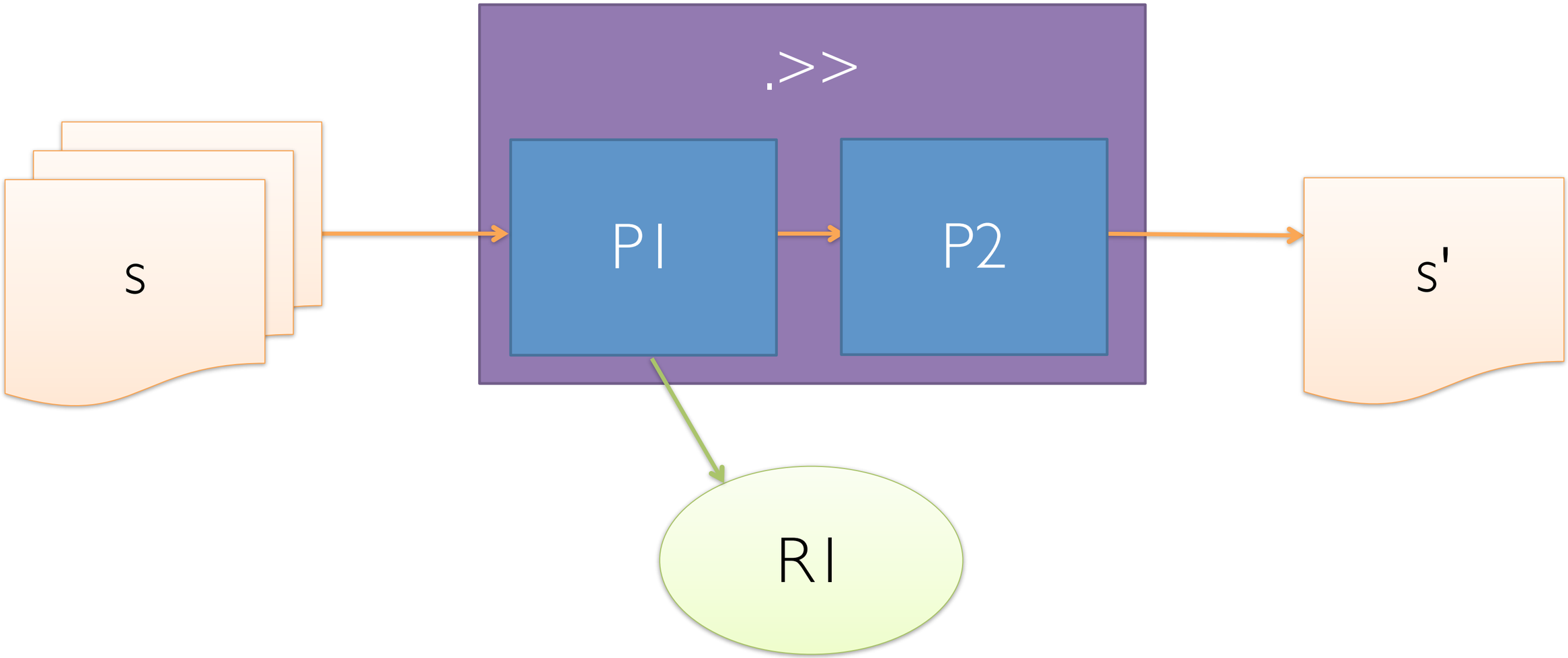
```
1.  let areaCodeParser : Parser<string, unit> =
2.    between (pchar '(') (pchar ')')
3.      (manyMinMaxSatisfy 3 3 isDigit)

4.  let localNumberParser : Parser<string * string, unit> =
5.    ((manyMinMaxSatisfy 3 3 isDigit) .>> pchar '-') .>>.
6.      (manyMinMaxSatisfy 4 4 isDigit)

7.  let phoneParser : Parser<string * (string * string), unit> =
8.    (areaCodeParser .>> pchar ' ') .>>. localNumberParser

9.  Console.WriteLine (go phoneParser "(212) 555-0134")
```

```
1.  Success! Parsed '(212, (555, 0134))'
```

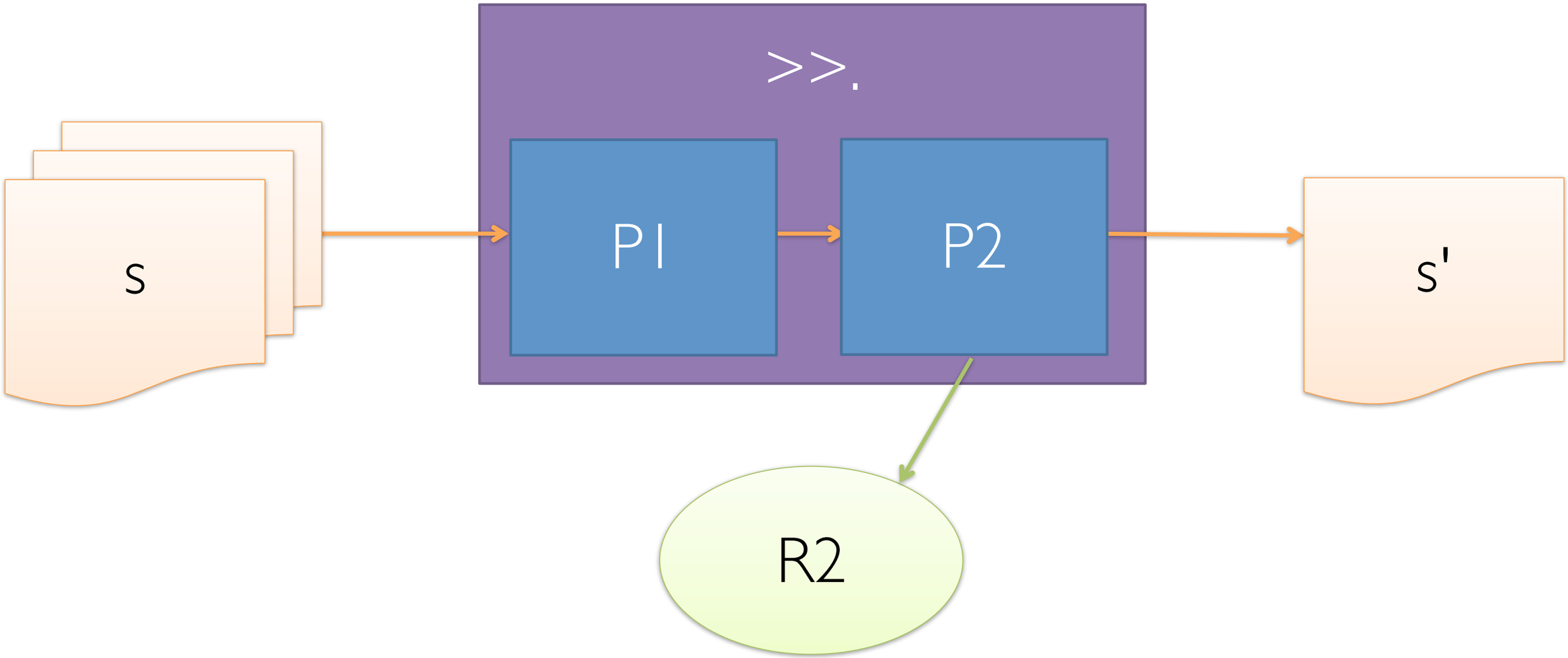
Run Multiple Parsers



Run Multiple Parsers



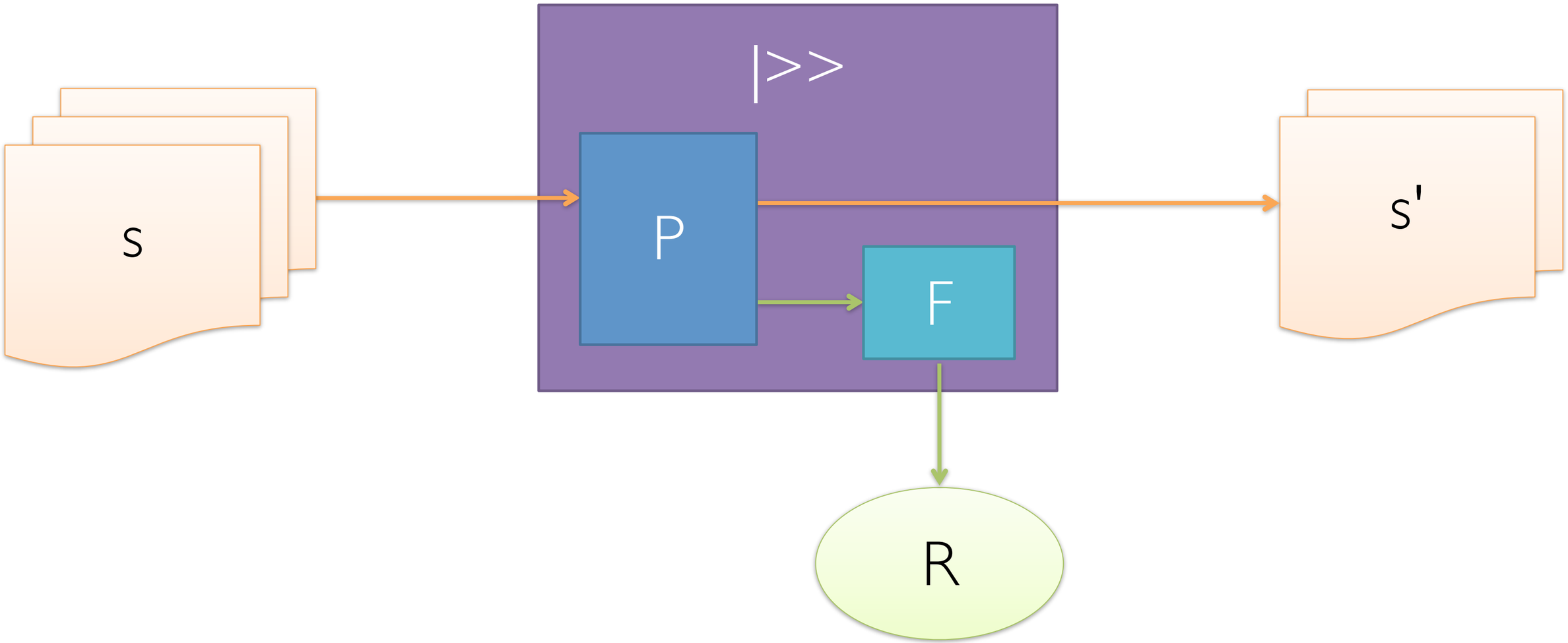
Run Multiple Parsers



```
1.  let areaCodeParser : Parser<string, unit> =
2.    between (pchar '(') (pchar ')')
3.      (manyMinMaxSatisfy 3 3 isDigit)

4.  let localNumberParser : Parser<string * string, unit> =
5.    ((manyMinMaxSatisfy 3 3 isDigit) .>> pchar '-') .>>.
6.      (manyMinMaxSatisfy 4 4 isDigit)

7.  let phoneParser : Parser<string * (string * string), unit> =
8.    (areaCodeParser .>> pchar ' ') .>>. localNumberParser

9.  Console.WriteLine (go phoneParser "(212) 555-0134")
```

```
1.  Success! Parsed '(212, (555, 0134))'
```

Parse + Apply

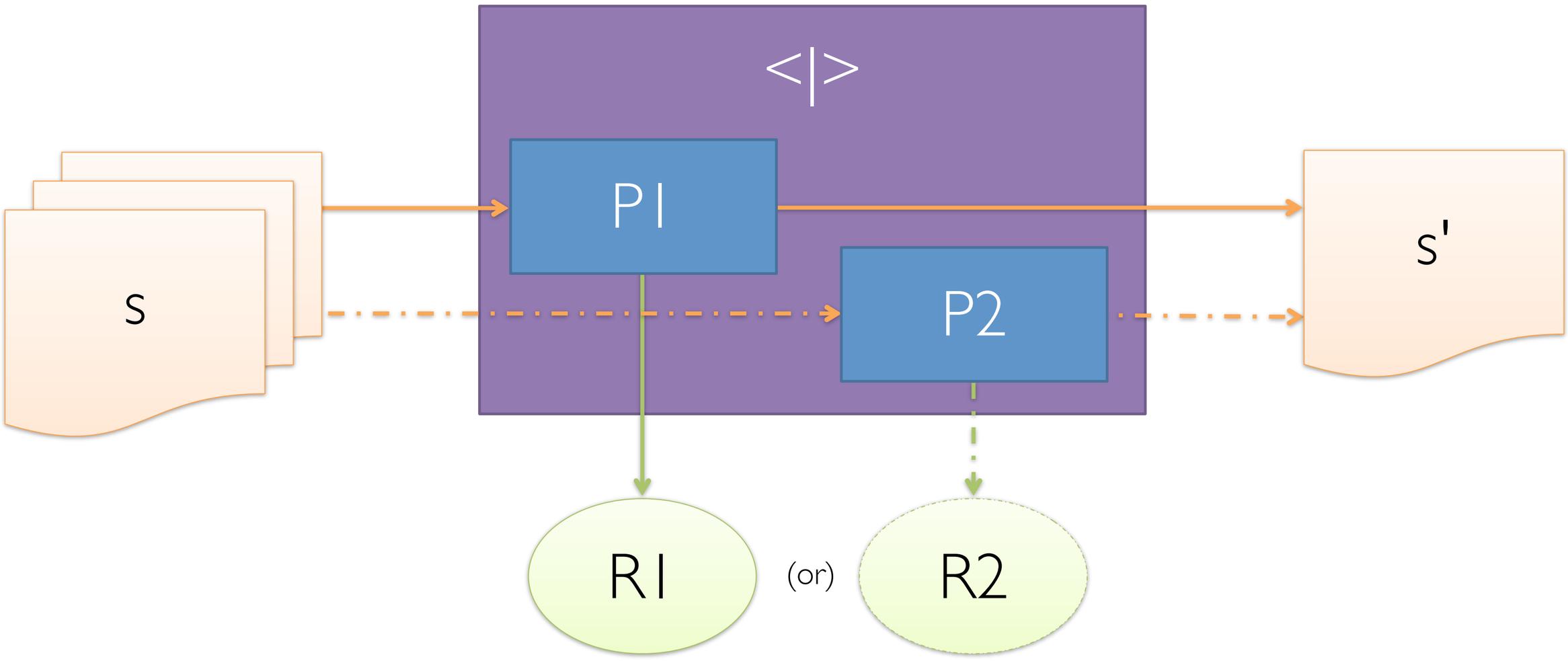


```
type AreaCode =
  { Code: string }
  with
    override this.ToString() = this.Code

type LocalNumber =
  { Exchange: string; Subscriber: string }
  with
    override this.ToString() = sprintf "%O-%O" this.Exchange this.Subscriber

type PhoneNumber =
  | Local of LocalNumber
  | Full of AreaCode * LocalNumber
  with
    override this.ToString() =
      match this with
      | Local(l) -> string l
      | Full(a, l) -> sprintf "(%O) %O" a l
```

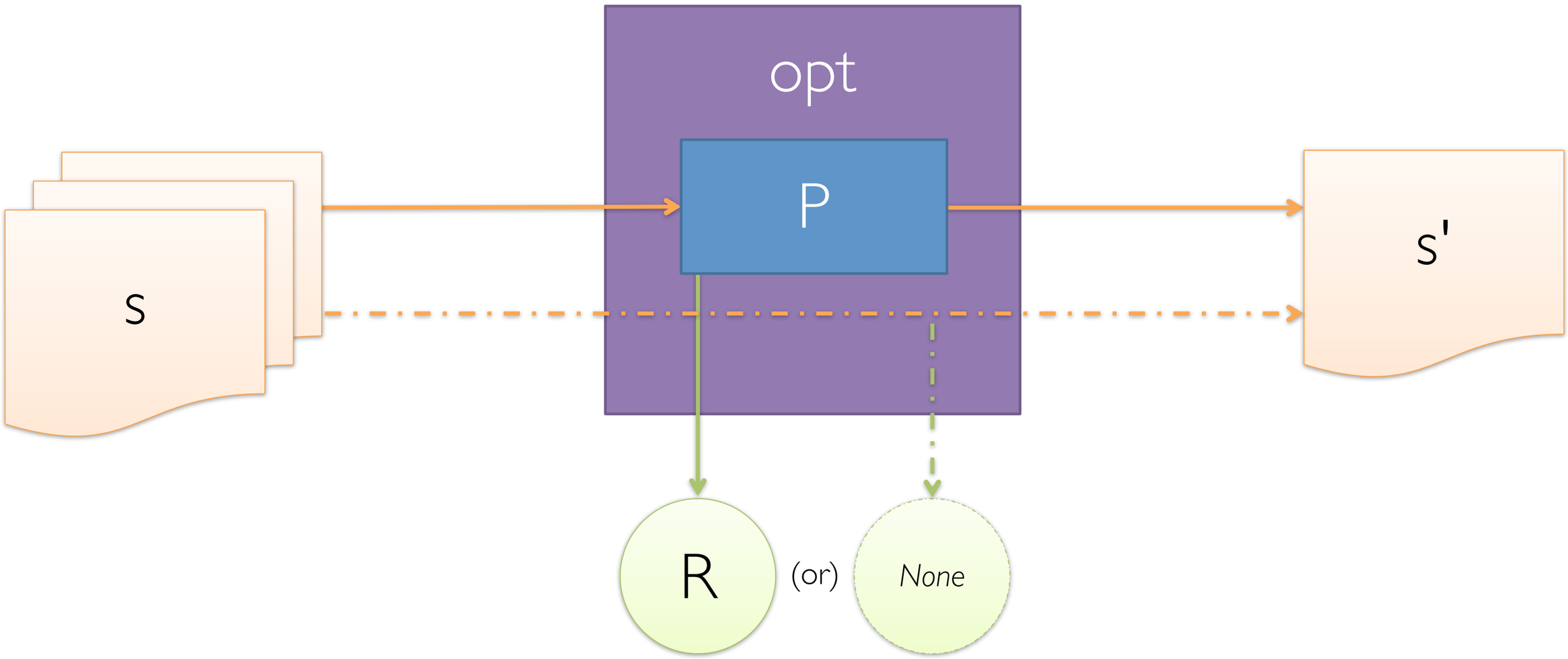
```
1.  let areaCodeParser : Parser<AreaCode, unit> =
2.    between (pchar '(') (pchar ')')
3.      (manyMinMaxSatisfy 3 3 isDigit)
4.        |>> (fun s -> { Code = s })

5.  let localNumberParser : Parser<LocalNumber, unit> =
6.    ((manyMinMaxSatisfy 3 3 isDigit) .>> pchar '-') .>>.
7.      (manyMinMaxSatisfy 4 4 isDigit)
8.        |>> (fun (e, s) -> { Exchange = e; Subscriber = s })

9.  let phoneParser : Parser<PhoneNumber, unit> =
10.   (areaCodeParser .>> pchar ' ') .>>. localNumberParser
11.     |>> Full
```

```
1.  Success! Parsed '(212) 555-0134'
```

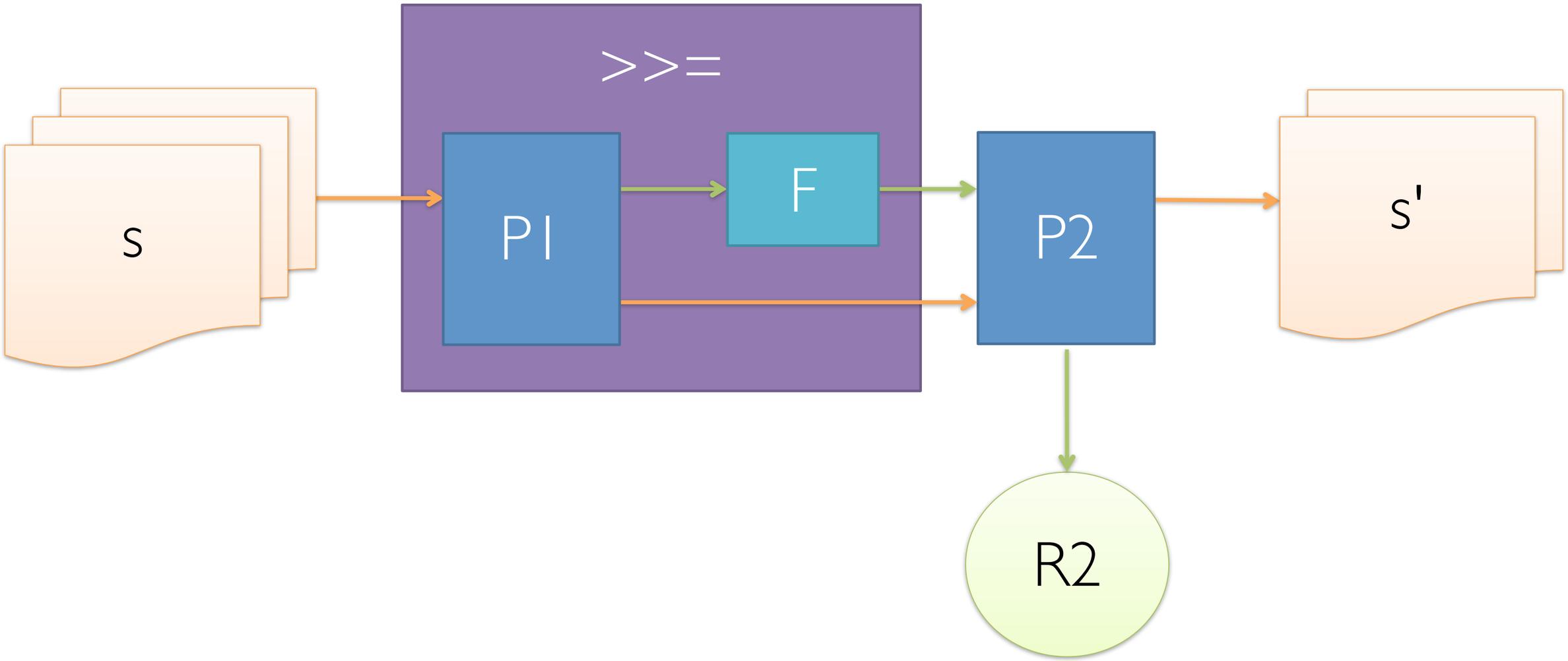
Parsing Possible Options



Parsing Possible Options



Bind



```
1.  let phoneParser : Parser<PhoneNumber, unit> =
2.    opt (areaCodeParser .>> pchar ' ') >>= (fun o ->
3.      match o with
4.      | Some a -> preturn a .>>. localNumberParser |>> Full
5.      | None -> localNumberParser |>> Local)

6.  Console.WriteLine (go phoneParser "(212) 555-0134")
7.  Console.WriteLine (go phoneParser "555-0134")
```

```
1.  Success! Parsed '(212) 555-0134'

2.  Success! Parsed '555-0134'
```

Parsing with FParsec

http://www.quanttec.com/fparsec/

http://en.wikibooks.org/wiki/F_Sharp_Programming

http://kevingessner.com/fparsec.pdf

Kevin Gessner • @kevingessner • hello@kevingessner.com