Typed Parsing and Unparsing for Untyped Regular Expression Engines

Gabriel Radanne

PEPM 2019

Some people, when confronted with a problem, think “I know, I’ll use regular expressions.”
Now they have two problems.

Jamie Zawinski

I want to search my logs to find domain names!

```
[0-9a-zA-Z.-]+|\[[0-9A-Fa-f:.]+\]
(:[0-9]+)?
```

It recognizes things like `foo.bar:8080`.

Now, I want to list domains that made a request on registered ports.

I want to search my logs to find domain names!

```
[0-9a-zA-Z.-]+|\[[0-9A-Fa-f:.]+\]
(:[0-9]+)?
```

It recognizes things like `foo.bar:8080`.

Now, I want to list domains that made a request on registered ports.

```
[0-9a-zA-Z.-]+|\[[0-9A-Fa-f:.]+\]
(:[0-9]+)?
```

```
([0-9a-zA-Z.-]+|\[[0-9A-Fa-f:.]+\])
(:([0-9]+))?
```

I add capture groups

```
([0-9a-zA-Z.-]+|\[[0-9A-Fa-f:.]+\])
(:([0-9]+))?
```

I add capture groups

And then I write a small program:

```
result = match(regex,s)
domain = result[1]
port = int(result[3])
if port < 49152:
  print(domain)
```

Now, I want to improve my program to also give me scheme and path.

```
([0-9a-zA-Z.-]+|\[[0-9A-Fa-f:.]+\])
(:([0-9]+))?
```

```
domain = result[1]
port = int(result[3])
```

Now, I want to improve my program to also give me scheme and path.

```
(([a-zA-Z]*)://)?
([0-9a-zA-Z.-]+|\[[0-9A-Fa-f:.]+\])
(:([0-9]+))?
((/[^/?]+)*)
```

```
domain = result[1]
port = int(result[3])
```

Now, I want to improve my program to also give me scheme and path.

```
(([a-zA-Z]*)://)?
([0-9a-zA-Z.-]+|\[[0-9A-Fa-f:.]+\])
(:([0-9]+))?
((/[^/?]+)*)
```

```
scheme = result[2]
domain = result[3]
port = int(result[5])
path = result[7].split("/")
```

Now, I want to improve my program to also give me scheme and path.

```
(([a-zA-Z]*)://)?
([0-9a-zA-Z.-]+|\[[0-9A-Fa-f:.]+\])
(:([0-9]+))?
((/[^/?]+)*)
```

```
scheme = result[2]
domain = result[3]
port = int(result[5])
path = result[7].split("/")
```

What if I want to differentiate domain names and IP addresses ?

What have we learned?

Pros:

- Composition of *recognition* is good(-ish)
- Linear time (mostly, . . .)

Cons:

- Composition of *extraction* is completely broken
- Extracting things under star/alternative is painful

Common answer:

Meh, Just use parser combinators

Pros:

- Everything composes
- Processing/extraction integrated into the parser (Applicative,. . .)
- Star/Alternative works well (Alternative,. . .)

Cons:

- It’s slow (not linear time)

Common answer:

Meh, Just use parser combinators

Pros:

- Everything composes
- Processing/extraction integrated into the parser (Applicative,. . .)
- Star/Alternative works well (Alternative,. . .)

Cons:

- It's slow (not linear time)

Common answer:

Meh, Just use parser combinators

Pros:

- Everything composes
- Processing/extraction integrated into the parser (Applicative,. . .)
- Star/Alternative works well (Alternative,. . .)

Cons:

- It’s slow (not linear time)

Another answer:

Just use full regex parsing algorithms

Pros:

- Everything composes
- Typed interpretation of regular expressions with ADTs
- Linear time

Cons:

- Can I use Greedy and POSIX semantics?
- Does it support charsets?
- Please let me use Re2 instead. ☹

Another answer:

Just use full regex parsing algorithms

Pros:

- Everything composes
- Typed interpretation of regular expressions with ADTs
- Linear time

Cons:

- Can I use Greedy and POSIX semantics?
- Does it support charsets?
- Please let me use Re2 instead. ☹

Another answer:

Just use full regex parsing algorithms

Pros:

- Everything composes
- Typed interpretation of regular expressions with ADTs
- Linear time

Cons:

- Can I use Greedy and POSIX semantics?
- Does it support charsets?
- Please let me use Re2 instead. ☹

Another answer:

Just use full regex parsing algorithms

Pros:

- Everything composes
- Typed interpretation of regular expressions with ADTs
- Linear time

Cons:

- Can I use Greedy and POSIX semantics?
- Does it support charsets?
- Please let me use Re2 instead. ☹

Another answer:

Just use full regex parsing algorithms

Pros:

- Everything composes
- Typed interpretation of regular expressions with ADTs
- Linear time

Cons:

- Can I use Greedy and POSIX semantics?
- Does it support charsets, word boundaries?
- Please let me use Re2 instead. ☹

Another answer:

Just use full regex parsing algorithms

Pros:

- Everything composes
- Typed interpretation of regular expressions with ADTs
- Linear time

Cons:

- Can I use Greedy and POSIX semantics?
- Does it support charsets, word boundaries, lookaround operators?
- Please let me use Re2 instead. ☹

Another answer:

Just use full regex parsing algorithms

Pros:

- Everything composes
- Typed interpretation of regular expressions with ADTs
- Linear time

Cons:

- Can I use Greedy and POSIX semantics?
- Does it support charsets, word boundaries, lookaround operators, streaming?
- Please let me use Re2 instead. ☹

Another answer:

Just use full regex parsing algorithms

Pros:

- Everything composes
- Typed interpretation of regular expressions with ADTs
- Linear time

Cons:

- Can I use Greedy and POSIX semantics?
- Does it support charsets, word boundaries, lookaround operators, streaming, online/lazy determinization?
- Please let me use Re2 instead. ☹

Another answer:

Just use full regex parsing algorithms

Pros:

- Everything composes
- Typed interpretation of regular expressions with ADTs
- Linear time

Cons:

- Can I use Greedy and POSIX semantics?
- Does it support charsets, word boundaries, lookaround operators, streaming, online/lazy determinization, . . . ?
- Please let me use Re2 instead. ☹

Another answer:

Just use full regex parsing algorithms

Pros:

- Everything composes
- Typed interpretation of regular expressions with ADTs
- Linear time

Cons:

- Can I use Greedy and POSIX semantics?
- Does it support charsets, word boundaries, lookaround operators, streaming, online/lazy determinization, . . . ?
- Please let me use Re2 instead. ☹

Idea:

Retrofit regex parsing on existing engine

Tyre

A familiar API

```
type ’a t (* A regular expression that captures ’a *)

(** Applicative-like *)
val conv : (’a -> ’b) -> (’b -> ’a) -> ’a t -> ’b t
val ( *> ) : _ t -> ’a t -> ’a t
val (<&>) : ’a t -> ’b t -> (’a * ’b) t

(* Alternative-like *)
val (<|>) : ’a t -> ’b t -> [‘Left of ’a | ‘Right of ’b] t
val list : ’a t -> ’a list t
val opt : ’a t -> ’a option t
```

A familiar API

```
type ’a t

(∗ Base element ∗)
val regex : regex -> string t

let int : int t =
  conv string_of_int int_of_string (regex "[0-9]+")
```

A familiar API

```
type ’a t

(* Base element *)
val regex : regex -> string t

let int : int t =
  conv string_of_int int_of_string (regex "[0-9]+")
```

Revisiting URLs

```
let schm: string t      = regex "[^/:?#]*" <* str "://"
let host: string t      = regex "[^/:?#]+"
let port: int option t  = opt (char ’:’ *> int)
let path: string list t = list (char ’/’ *> regex "[^/?#]*")

let url : url t =
  conv to_url from_url (schm <&> host <&> port <&> path)
```

Revisiting URLs

```
let schm: string t      = regex "[^/:?#]*" <* str "://"
let host: string t      = regex "[^/:?#]+"
let port: int option t  = opt (char ’:’ *> int)
let path: string list t = list (char ’/’ *> regex "[^/?#]*")

let url : url t =
  conv to_url from_url (schm <&> host <&> port <&> path)
```

Revisiting URLs – syntax extension

```
let schm: string t      = [%tyre "(?<schm>:[^/:?#]*)://"]
let host: string t      = [%tyre "[^/:?#]+" ]
let port: int option t  = [%tyre "(:(?&int))?"]
let path: string list t = [%tyre "(/(?<p>:[^/?#]*))*"]

let url =
  [%tyre "(?&schm)(?&host)(?&port)(?&path)"]
```

Using typed regular expressions

```
# let c = compile url
# exec c "http://foo.com:80/some/path"
- : (url, url error) result =
 Result.Ok { scheme = "http" ; host = "foo.com";
             port = Some 80  ; path = ["some"; "path"] }

# let myurl =  { scheme = "ftp" ; host = "myserver.net" ;
                 port = None ; path = []}
# eval url myurl ;;
- : string = "ftp://myserver.net"
```

Using typed regular expressions

```
# let c = compile url
# exec c "http://foo.com:80/some/path"
- : (url, url error) result =
 Result.Ok { scheme = "http" ; host = "foo.com";
             port = Some 80  ; path = ["some"; "path"] }

# let myurl =  { scheme = "ftp" ; host = "myserver.net" ;
                 port = None ; path = []}
# eval url myurl ;;
- : string = "ftp://myserver.net"
```

Using typed regular expressions

```
# let c = compile url
# exec c "http://foo.com:80/some/path"
- : (url, url error) result =
 Result.Ok { scheme = "http" ; host = "foo.com";
             port = Some 80  ; path = ["some"; "path"] }

# let myurl =  { scheme = "ftp" ; host = "myserver.net" ;
                 port = None ; path = []}
# eval url myurl ;;
- : string = "ftp://myserver.net"
```

Using typed regular expressions

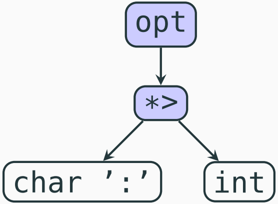
```
# let c = compile url
# exec c "http://foo.com:80/some/path"
- : (url, url error) result =
 Result.Ok { scheme = "http" ; host = "foo.com";
             port = Some 80  ; path = ["some"; "path"] }

# let myurl =  { scheme = "ftp" ; host = "myserver.net" ;
                 port = None ; path = []}
# eval url myurl ;;
- : string = "ftp://myserver.net"
```

Using typed regular expressions

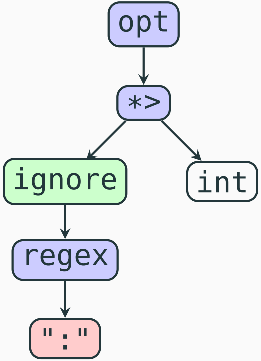
```
# let c = compile url
# exec c "http://foo.com:80/some/path"
- : (url, url error) result =
 Result.Ok { scheme = "http" ; host = "foo.com";
             port = Some 80  ; path = ["some"; "path"] }

# let myurl =  { scheme = "ftp" ; host = "myserver.net" ;
                 port = None ; path = []}
# eval url myurl ;;
- : string = "ftp://myserver.net"
```

Internals

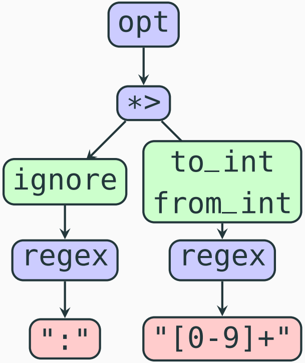
Internals



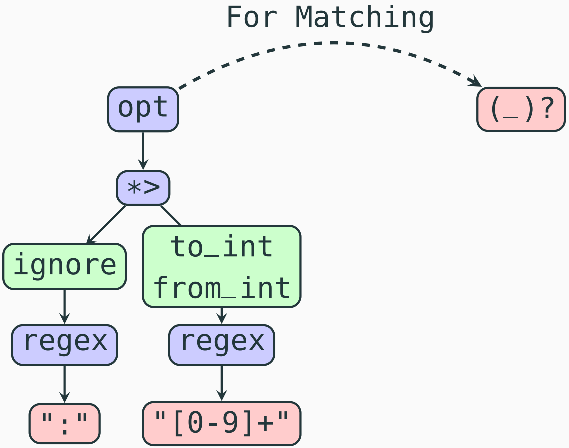
Internals



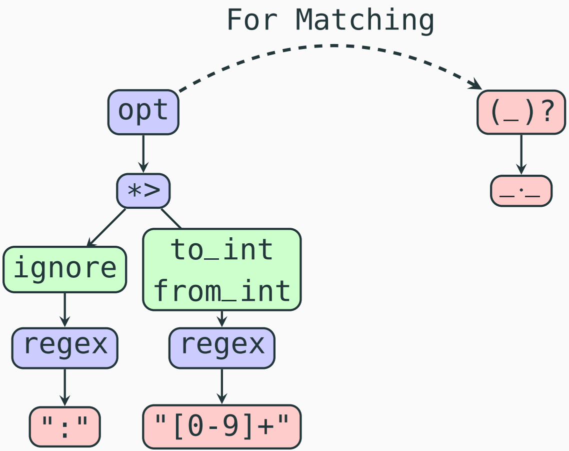
Internals



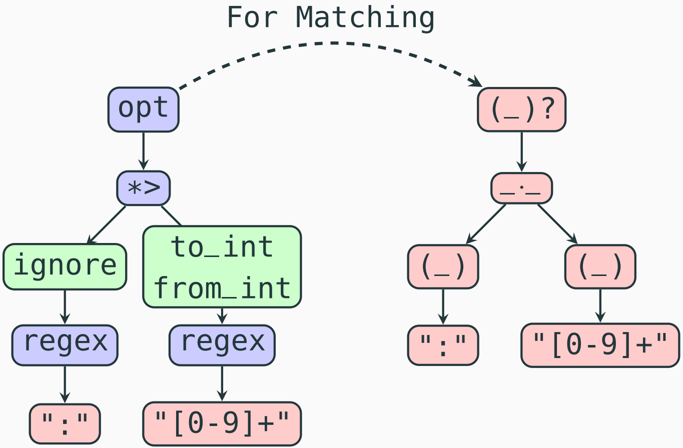
Internals



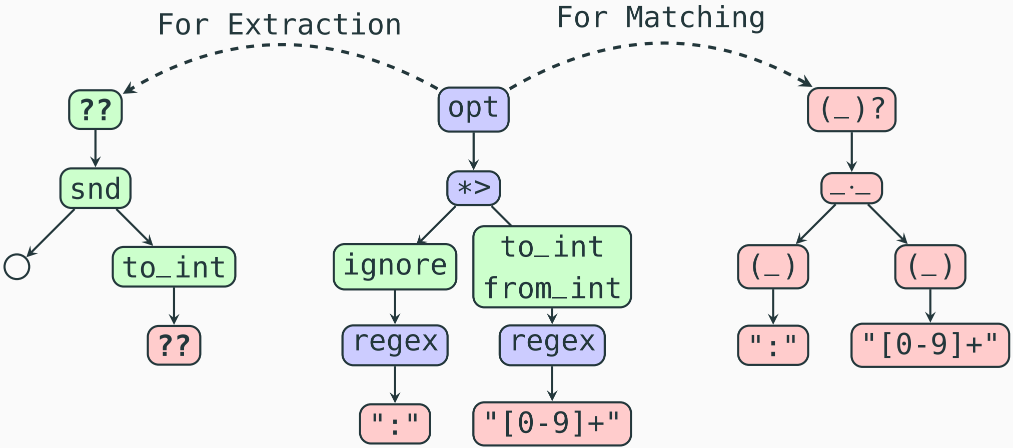
Internals



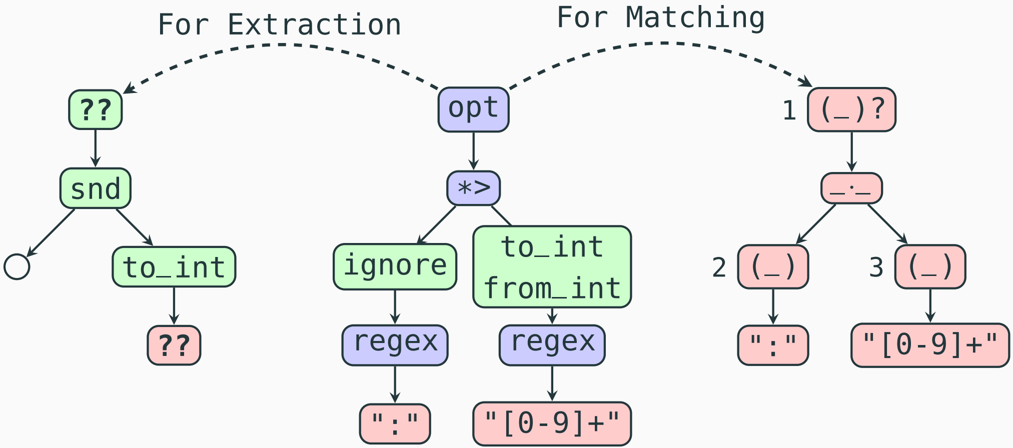
Internals



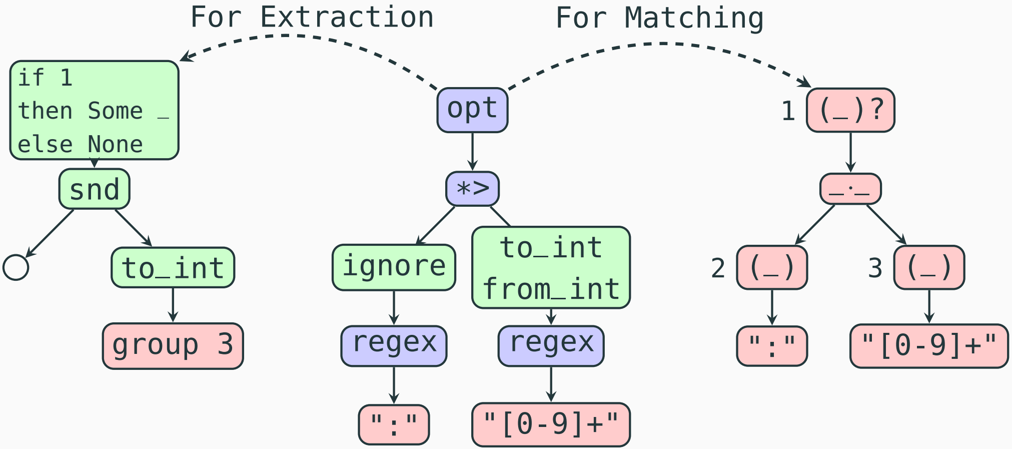
Internals


For Extraction
For Matching
??
snd
to_int
??
opt
*>
ignore
regex
":"
to_int
from_int
regex
"[0-9]+"
(_)?
_·_
(_)
":"
(_)
"[0-9]+"

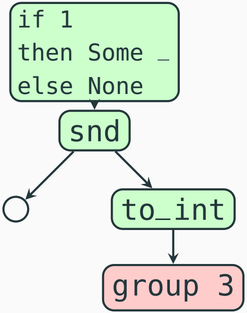
Internals


For Extraction
For Matching
??
snd
○
to_int
??
opt
*>
ignore
regex
":"
to_int
from_int
regex
"[0-9]+"
1
(_)?
_·_
2
(_)
":"
3
(_)
"[0-9]+"

Internals


For Extraction
For Matching
if 1
then Some _
else None
snd
○
to_int
group 3
opt
*>
ignore
regex
":"
to_int
from_int
regex
"[0-9]+"
1
(_)?
_·_
2
(_)
":"
3
(_)
"[0-9]+"

Internals

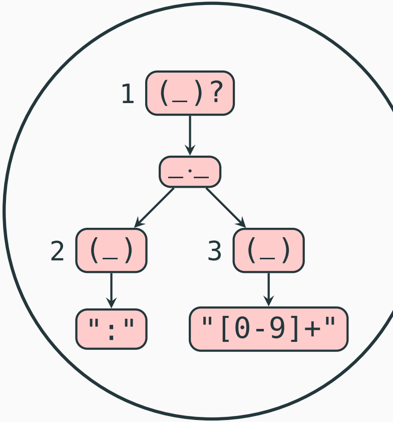


```
if 1
then Some _
else None
```

snd

to_int

group 3



1 (_)?

_·_

2 (_)

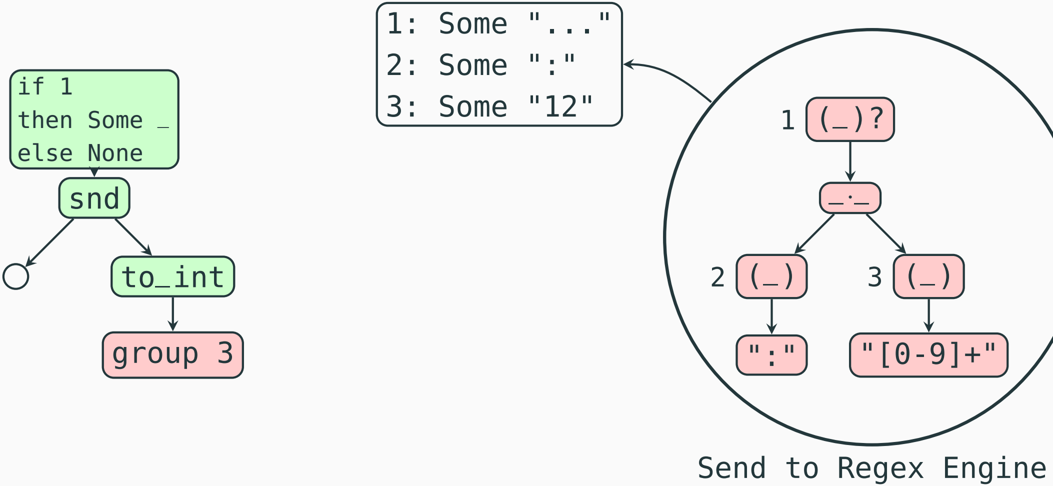
3 (_)

":"

"[0-9]+"

Send to Regex Engine

Internals



if 1
then Some _
else None

snd

to_int

group 3

```
1: Some "..."
2: Some ":"
3: Some "12"
```

1 (_)?

_:_

2 (_)

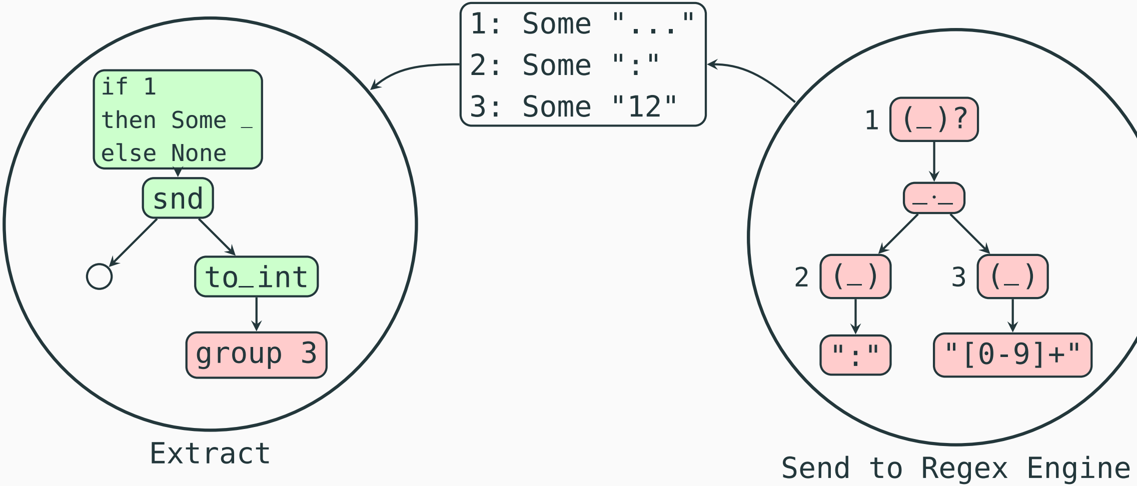
3 (_)

":"

"[0-9]+"

Send to Regex Engine

Internals



```
if 1
then Some _
else None
```

snd

to_int

group 3

Extract

```
1: Some "..."
2: Some ":"
3: Some "12"
```

1 (_)?

_·_

2 (_)

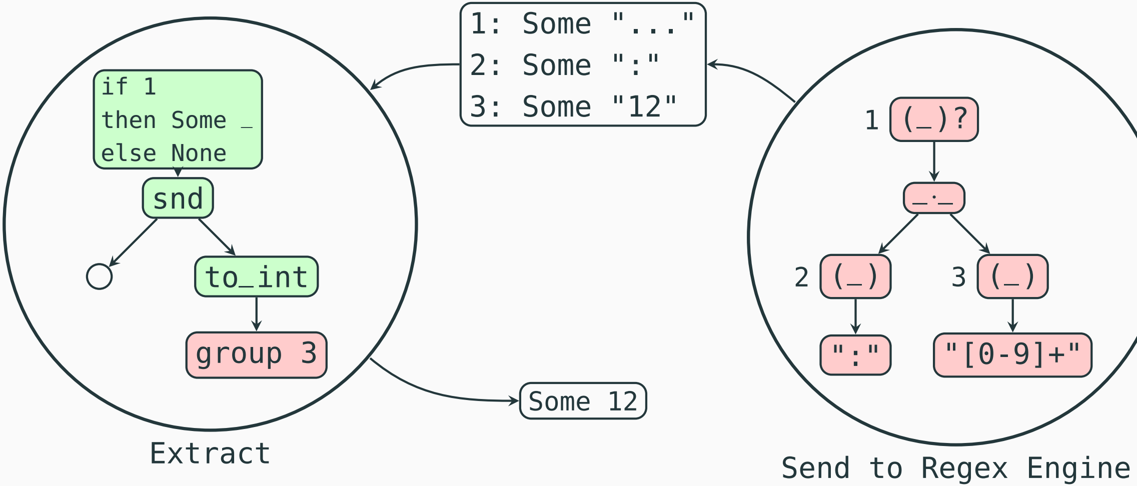
3 (_)

":"

"[0-9]+"

Send to Regex Engine

Internals



1: Some "..."
2: Some ":"
3: Some "12"

if 1
then Some _
else None

snd

to_int

group 3

Some 12

Extract

1 (_)?

_·_

2 (_)

3 (_)

":"

"[0-9]+"

Send to Regex Engine

Two thorny issues remains:

- Alternatives
- Repetitions

Two thorny issues remains:

- Alternatives

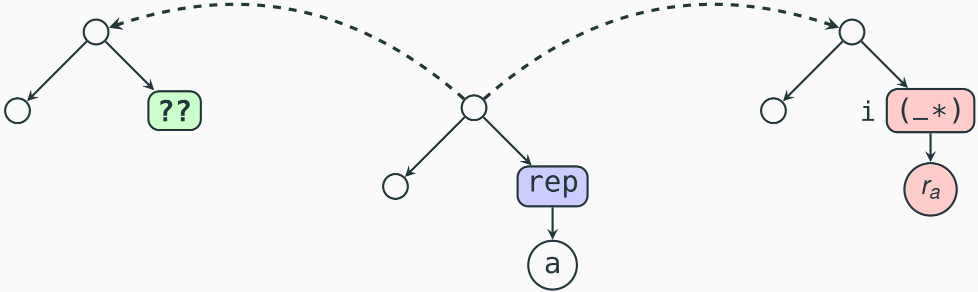
  ⇒: Similar to option: abuse groups for branching
- Repetitions

Repetitions

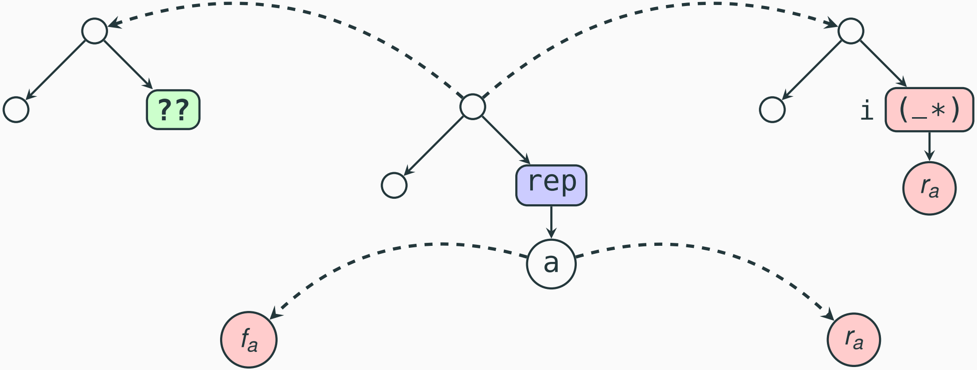
Let’s take a concrete example:

```
let r = str "numbers:" *> rep (int <* char ’;’)
let cr = compile r
exec cr "numbers:1;2;345;6;"
> Result.Ok [1; 2; 345; 6]
```

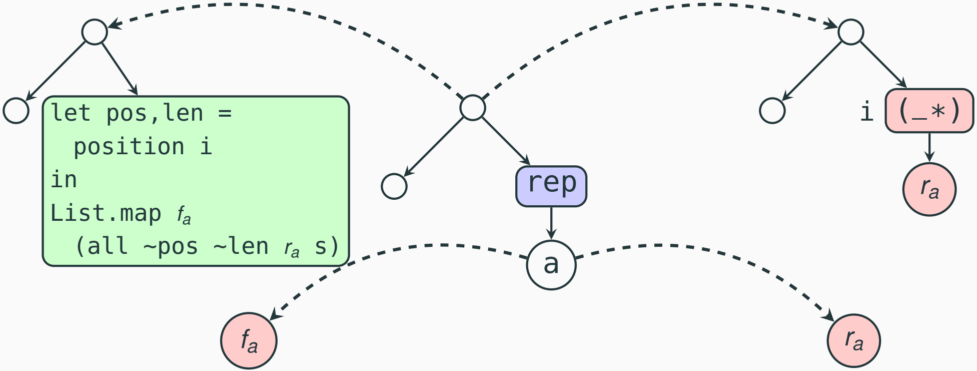
Repetitions



Repetitions



Repetitions



```
let pos,len =
  position i
in
List.map f_a
  (all ~pos ~len r_a s)
```

Repetitions

- Pay a linear cost (proportional to the star height)
- Only problematic in the typed part! `...(regex "abc+")...` is fine.
- Top-level repetitions are not costly

Experimentations

Experimentations:

- Implemented a spec-compliant URI parser.
  ⇒ faster and safer than original `ocaml-uri`, passes all the tests
- Primitive HTTP parser
  ⇒ 2.5 times faster than the equivalent parser-combinator implementation
- Various uses in the wild

See the paper for details

Conclusion

Take away

- Regular expression parsing doesn’t really compose
  ⇒ You have to enrich them with extraction info
- Implementing a fast and featureful regex engine is a non-trivial undertaking
  ⇒ Try to reuse the existing work as much as possible
- Parsing combinators provide a nice API, but sometimes you want a tagged representation
- Syntax extensions really help adoption (see the paper)

Conclusion

I presented a method to have typed regex parsing on top of untyped engines

- Work on top of many engines

  ⇒ Can be used with various regex languages (but not backrefences . . .)
- Various optimisations in the paper:

  ⇒ Use marks to avoid groups in alternatives

  ⇒ Extraction code can be staged too!
- Implement alternatives and repetitions
- Not perfect, but sufficient in practice

Implemented in OCaml and distributed:

- Library: `tyre` in opam
- Syntax extension: `ppx_tyre` in opam

Conclusion

I presented a method to have typed regex parsing on top of untyped engines

- Work on top of many engines

  ⇒ Can be used with various regex languages (but not backrefences . . .)
- Various optimisations in the paper:

  ⇒ Use marks to avoid groups in alternatives

  ⇒ Extraction code can be staged too!
- Implement alternatives and repetitions
- Not perfect, but sufficient in practice

Implemented in OCaml and distributed:

- Library: `tyre` in opam
- Syntax extension: `ppx_tyre` in opam

Future Work and questions

- Better scheme for repetitions ?
- Make sure exactly which extensions of regexes are compatible.
- Compatibility with the Javascript Regex API . . .

Questions?

Using typed regular expressions

```
type ’a re
(** A compiled typed regular expression of type ’a *)

val compile : ’a t -> ’a re
val exec : ’a re -> string -> (’a, error) result

(* Unparsing/Printing a value using a regex *)
val eval : ’a t -> ’a -> string

(* Routing: pattern matching for regexs *)
val route : ’a route list -> ’a re
```

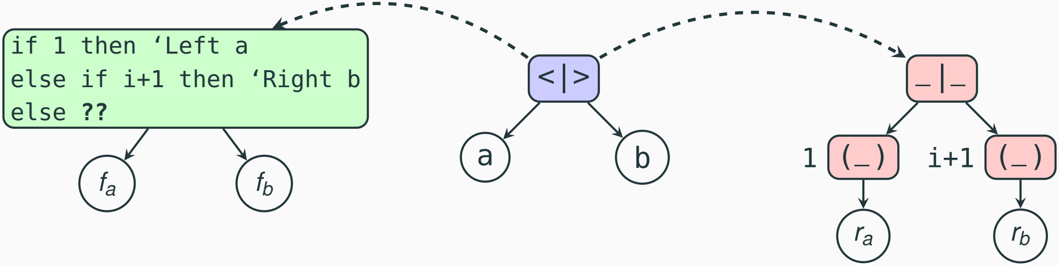
Using typed regular expressions

```
type ’a re
(∗∗ A compiled typed regular expression of type ’a ∗)

val compile : ’a t -> ’a re
val exec : ’a re -> string -> (’a, error) result

(∗ Unparsing/Printing a value using a regex ∗)
val eval : ’a t -> ’a -> string

(∗ Routing: pattern matching for regexs ∗)
val route : ’a route list -> ’a re
```

Using typed regular expressions

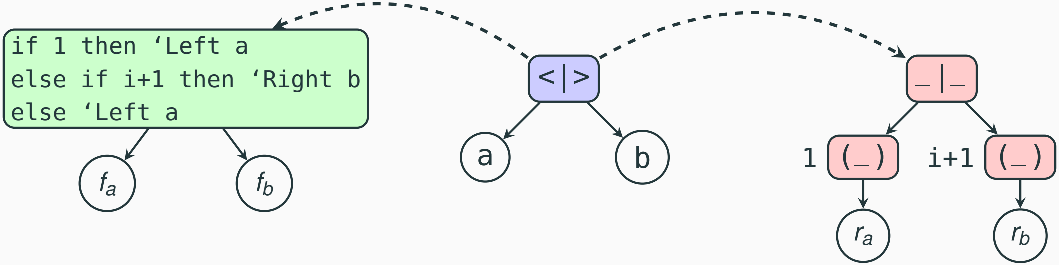
```
type ’a re
(** A compiled typed regular expression of type ’a *)

val compile : ’a t -> ’a re
val exec : ’a re -> string -> (’a, error) result

(* Unparsing/Printing a value using a regex *)
val eval : ’a t -> ’a -> string

(* Routing: pattern matching for regexs *)
val route : ’a route list -> ’a re
```

Alternatives



```
if 1 then ‘Left a
else if i+1 then ‘Right b
else ??
```

Alternatives



```
if 1 then ‘Left a
else if i+1 then ‘Right b
else ‘Left a
```

Alternatives

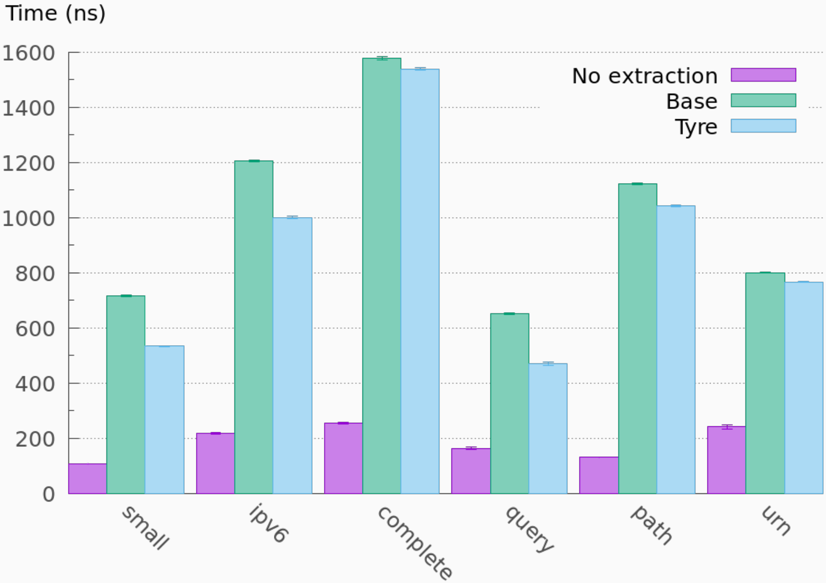
- Need to insert many additional groups
- Can be improved by using marks (see the paper)

Comparison with parser combinators

Angstrom	Tyre	Tyre, test-only
$28.3 \pm 1.8 ms$	$11.6 \pm 0.13 ms$	$7.6 \pm 0.013 ms$

Figure 1: Parsing 100 HTTP requests with various parsers

Performances of URI parsing



Definition of URIs

small:	`http://foo.com`
ipv6:	`http://%5Bdead%3Abeef%3A%3Adead%3A0%3Abeaf%5D`
complete:	`https://user:pass@foo.com:123/a/b/c?foo=1&bar=5#5`
query:	`//domain?f+1=bar&+f2=bar%212`
path:	`http://a/b/c/g;x?y#s`
urn:	`urn:uuid:f81d4fae-7dec-11d0-a765-00a0c91e6bf6`