Functional Parsing for Novel Markup Languages

What it is

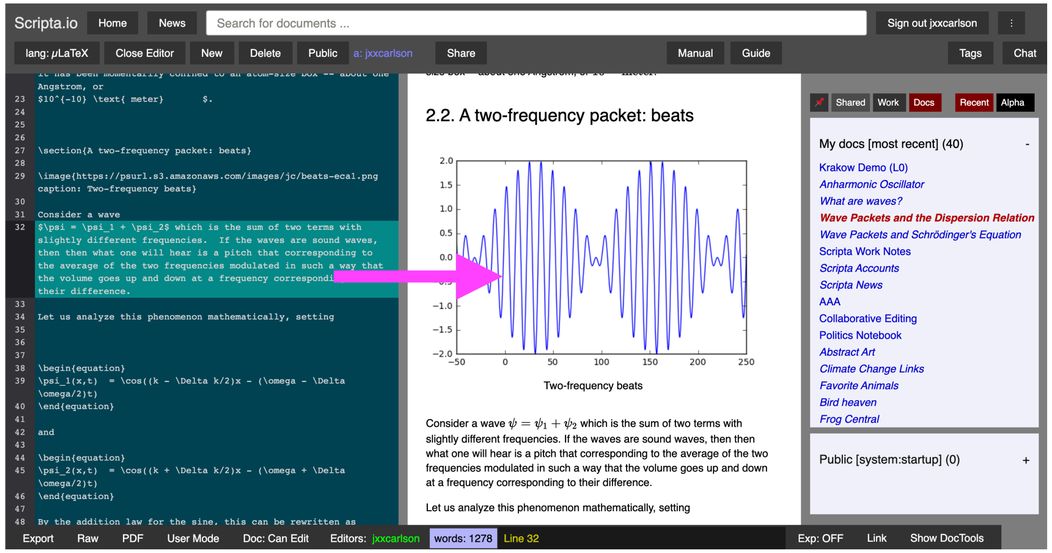
Why we need it

James Carlson Lambda Days (Krakow) July 28, 2022 scripta.io jxxcarlson@gmail.com

How I got started …

Web app for editing and publishing math docs

Partial support: Simons Foundation



Requirements

- Instantaneous rendering
- In-place, real-time error handling
  - All text processed
  - Rendered text not messed up …
  - Note error in rendered text

scripta.io

Elm, Lamdera

Fault-tolerant parser

Prior and related work

- Matt Griffith (Elm Markup)

- Rob Simmons & Team (brilliant.org)

Markup Languages

MicroLaTeX

XMarkdown

L0

L0

| title
Krakow Example

This is a [i [blue real]] test.
Ho ho ho!

| theorem
There are infinitely many primes
$p \equiv 1 \modulo p$.

|| image width:300
https://images.io/robin.jpg

Krakow Example

This is a *real* test. Ho ho ho!

Theorem 1

There are infinitely many primes $p \equiv 1 \bmod 4$.

Demo!

Parser I

Syntax Tree

```
type Expr
    = Fun String (List Expr) Meta
    | Text String Meta
    | Verbatim String String Meta
```

This is [i strong] stuff. ⟶

[**Text** “This is”, **Fun** “i” [**Text** “strong”], **Text** “stuff.”]

Math: $f'(x) = 3*x^2$ 

[**Text** “Math:”, **Verbatim** “math” [**Text** “f'(x) = 3*x^2”],]

Parser data flow

```
[i bright [blue flowers]]
```

tokenize

```
[LB, S “i”, S “bright”, LB, S “blue”
            , S “flowers”, RB, RB]
```

parse

```
[Fun “i” [Text “bright”,
              Fun “blue” [Text “flowers”]]
```

Parser II

Shift-reduce algorithm

Functional loop

Functional Loops

- ```
  type Step state a = Loop state | Done a
  ```

- ```
  loop : state -> (state -> Step state a) -> a
  loop s nextStep =
    case nextStep s of
      Loop t -> loop t nextStep
      Done b -> b
  ```

- ```
  type alias State = { tokens : List Token, tokenIndex : Int
                     , stack : List Token, committed : List Expr}
  ```
  ```

Parser: nextStep function

```
nextStep : State -> Step State State
nextStep state =
    case getToken state of
        Nothing ->
            if stackIsEmpty state
            then Done state
            else recoverFromError state

        Just token ->
            state
                |> advanceTokenIndex
                |> pushOrCommit token
                |> applyIf isReducible reduce
                |> Loop
```

SHIFT

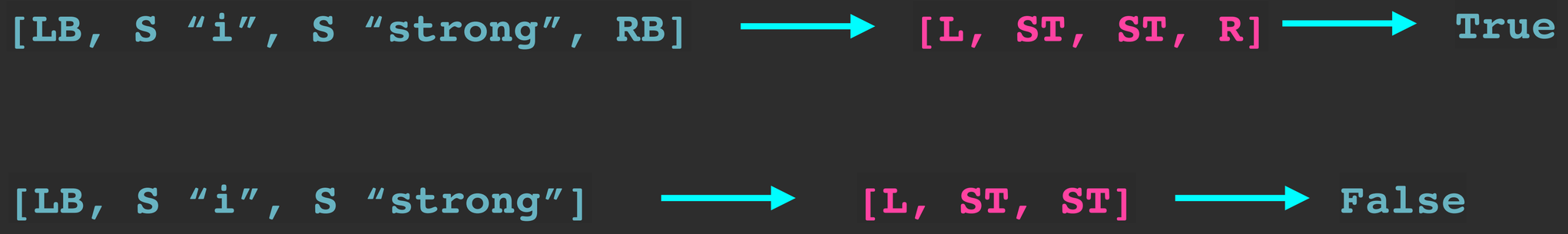
REDUCE

- STACK -> EXPR -> COMITTED
- CLEAR STACK

Parser III

reducibility

isReducible



Implementation: two mutually recursive functions

```
isReducible : List Symbol -> Bool
```

```
hasReducibleArgs : List Symbol -> Bool
```

```
[L, ST, ST, R]        [L, ST, L, ST, ST, R, ST, R]
```

```
isReducible : List Symbol -> Bool
isReducible symbols =
    case symbols of
        L :: ST :: rest ->   [L, ST, ST, R]    [L, ST, L, ST, ST, R, ST, R]
            case List.Extra.last rest of

                Just R -> hasReducibleArgs (dropLast rest) [ST]   [L, ST, ST, R, ST]

                _ -> False

        _ -> False
```

```
hasReducibleArgs [ST] = True

hasReducibleArgs [L, ST, ST, R, ST] = True
```

`hasReducibleArgs [ST]`

`hasReducibleArgs [L, ST, ST, R, ST]`

```
hasReducibleArgs : List Symbol -> Bool
hasReducibleArgs symbols =
    case symbols of
        [] ->
            True

        ST :: rest ->
            hasReducibleArgs rest

        L :: _ ->
            case split symbols of
                Nothing -> False

                Just ( prefix, suffix ) ->
                    isReducible prefix && hasReducibleArgs suffix

        _ -> False
```

ST :: [ ]
[ ]

[L, ST, ST, R, ST]

([L, ST, ST, R], [ST])

[L, ST, ST, R]

[ST]

`hasReducibleArgs [ST] = True`

`hasReducibleArgs [L, ST, ST, R, ST] = True`

Parser IV

worked examples

Parse: This is [i strong] stuff.

```
Tokens: S “This is ”, LB, S “i ”, S “strong ”, RB , S “stuff.”
                0          1    2          3          4        5

0:  c = [ ], s = [ ], p = 0

1:  c = [Text “This is “], s = [ ], p = 1  (Commit immediately)

2:  c = [Text “This is “], s = [LB], p = 2 (Shift)

3:  c = [Text “This is “], s = [S “i”, LB], p = 3 (Shift)

4:  c = [Text “This is “],
    s = [S “strong”, S “i”, LB], p = 4 (Shift)
```

Parse: This is [i strong] stuff.

Tokens: S “This is ”, LB, S “i ”, S “strong ”, RB , S “stuff.”
0 1 2 3 4 5

```
4:  c = [Text “This is “],
    s = [S “strong”, S “i”, LB], p = 4 (Shift)
```

```
5:  c = [Text “This is “],
    s = [RB, S “strong”, S “i”, LB], p = 5 (Shift)

6:  c = [Text “This is “, Fun “i” [Text “strong”]],
    s = [], p = 5 (REDUCE)

7:  c = [Text “This is “, Fun “i” [Text “strong”], Text “stuff”],
    s = [], p = 6 (Commit immediately)

DONE: This is strong stuff
```

What happens if there is an error?

This is [i strong stuff.

This is [i strong stuff.

Parse error text: This is [i strong stuff.

```
Tokens: S “This is ”, LB, S “i ”, S “strong stuff.”
              0         1    2              3

6:  c = [Text “This is “],

    s = [S “strong stuff”, S “i”, LB], p = 4 (ERROR)


7:  c = [Text “This is “, Fun “pink” [Text “[i”]],

    s = [], p = 3 (Run parser again)

8:  c = [  Text “This is “, Fun “pink” [Text “[i”]
         , Text “strong stuff”], s = [], p = 4 (DONE)
```

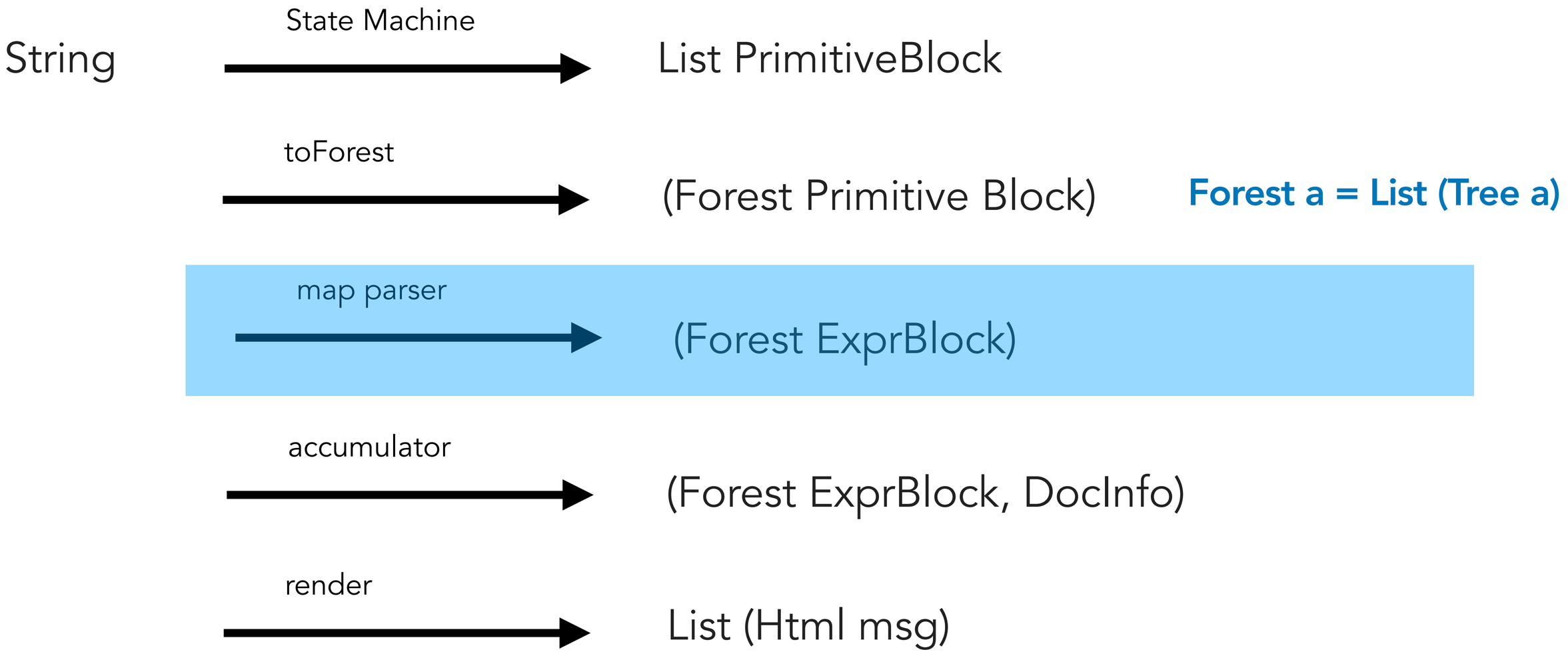
This is [i strong stuff.

Error recovery algorithm:

```
recoverFromError : State -> State
recoverFromError state =
  case (List.reverse stack) of
    LB :: S name :: rest ->
      push error element(LB :: S name) onto committed
      clear stack
      restart parser at index of head(rest)
    LB :: LB :: rest -> do something else
    LB :: RB :: rest -> do something else
```

Compiler Pipeline

Overview: compiler pipeline



mapAccumulate

```
mapAccumulate : (s -> a -> ( s, b )) -> s -> Tree a -> ( s, Tree b )
```

Map a function over every note while accumulating some value.

(From zwilias/elm-rosetree)

Blocks

Blocks make certain errors impossible

blah blah blah ….

\begin{theorem}
There are infinitely many prime numbers.
\end{theorem}

blah blah blah ….

blah blah blah ….

l theorem
There are infinitely many prime numbers.

blah blah blah ….

L0

Blocks

```
| title
Krakow Example

This is a [i [blue real]] test.
Ho ho ho!

| theorem
There are infinitely many primes
$p \equiv 1 \modulo p$.

|| image width:300
https://images.io/robin.jpg
```

Elements

Krakow Example

This is a *real* test. Ho ho ho!

Theorem 1

There are infinitely many primes $p \equiv 1 \bmod 4$.

Kinds of blocks

```
| title
Krakow Example

This is a [i [blue real]] test.
Ho ho ho!

| theorem
There are infinitely many primes
$p \equiv 1 \modulo p$.

|| image width:300
https://images.io/robin.jpg
```

Krakow Example

This is a *real* test. Ho ho ho!

Theorem 1

There are infinitely many primes $p \equiv 1 \bmod 4$.

Notes

- microLaTeX, xMarkdown:
  Parse to a common Syntax Tree (L0)

- Speed?
  Differential parsing

- Tests
  Round trip: (a) parse, (b) parse |> print |> parse, (c) compare

Thank you!