Teaspoon: Parser Combinators in Languages without User-Defined Operators

Eugene Lin

Monday 27th June, 2022

Imperial College London

Parser Combinators

What even are parser combinators?

What even are **parser** combinators?

- A parser is just a function in *your* favourite programming language!

What even are parser combinators?

- A parser is just a function in *your* favourite programming language!
- Simple functions take an input and obtain a result

What even are parser **combinators**?

- A parser is just a function in *your* favourite programming language!
- Simple functions take an input and obtain a result
- Combinators combine these parsers and create more complex parsers

But why?

Example: Calculator

```
nat  : [0-9]+;
expr : expr '+' term
     | expr '-' term
     | term;
term : term '*' fact
     | term '/' fact
     | fact;
fact : nat
     | '(' expr ')';
```

Example: Calculator

```
val  nat  = parseInt <$> /[0-9]+/;
lazy expr = expr <***> ('+' $> add) <*> term <|>
            expr <***> ('-' $> sub) <*> term <|>
            term;
lazy term = term <***> ('*' $> mul) <*> fact <|>
            term <***> ('/' $> div) <*> fact <|>
            fact;
lazy fact = nat <|>
            '(' *> expr <* ')';
```

Example: Arrays and Pairs

```
array  : '[' exprs ']';
exprs  : exprs ',' expr
       | expr;
object : '{' pairs '}';
pairs  : pairs ',' pair
       | pair;
```

Example: Arrays and Pairs

```
val exprs  = expr <:> many(',' *> expr);
val pairs  = pair <:> many(',' *> pair);
val array  = '[' *> exprs <* ']';
val object = '{' *> pairs <* '}';
```

Example: Arrays and Pairs

```
val exprs  = expr <:> many(',' *> expr);
val pairs  = pair <:> many(',' *> pair);
val array  = '[' *> exprs <* ']';
val object = '{' *> pairs <* '}';
```

-10,000 too much
code duplication

Example: Arrays and Pairs

```
function commaSep<A>(p: Parser<A>): Parser<A[]> {
    return p <:> many(',' *> p);
}
val array  = '[' *> commaSep(expr) <* ']';
val object = '{' *> commaSep(pair) <* '}';
```

Why Parser Combinators?

How are they better than parser generators?

Why Parser Combinators?

How are they better than parser generators?

- Easily obtain the desired result

Why Parser Combinators?

How are they better than parser generators?

- Easily obtain the desired result
- Lexing, parsing, and building AST in a single pass

Why Parser Combinators?

How are they better than parser generators?

- Easily obtain the desired result
- Lexing, parsing, and building AST in a single pass
- Host abstraction: *you* can build anything that's missing!

Why Parser Combinators?

How are they better than parser generators?

- Easily obtain the desired result
- Lexing, parsing, and building AST in a single pass
- Host abstraction: *you* can build anything that's missing!
- Parser combinators embody software engineering principles

Demo

Something's wrong...

Left Recursion

```
val  nat  = parseInt <$> /[0-9]+/;
lazy expr = expr <***> ('+' $> add) <*> term <|>
            expr <***> ('-' $> sub) <*> term <|>
            term;
lazy term = term <***> ('*' $> mul) <*> fact <|>
            term <***> ('/' $> div) <*> fact <|>
            fact;
lazy fact = nat <|>
            '(' *> expr <* ')';
```

Left Recursion

```
val  nat  = parseInt <$> /[0-9]+/;
lazy expr = chainl1(term,
    ('+' $> add) <|> ('-' $> sub)
);
lazy term = chainl1(fact,
    ('*' $> mul) <|> ('/' $> div)
);
lazy fact = nat <|>
            '(' *> expr <* ')';
```

A Real Problem

April 23rd, 2021

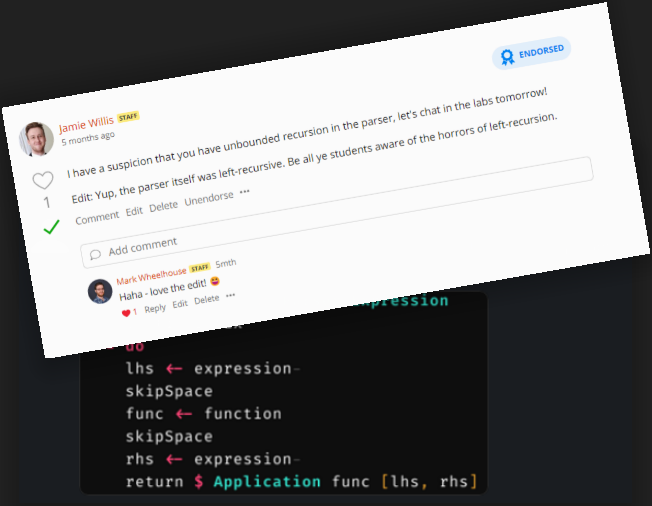
1:55 PM

can anyone help me try to solve the dreaded left-recursion in my parser?

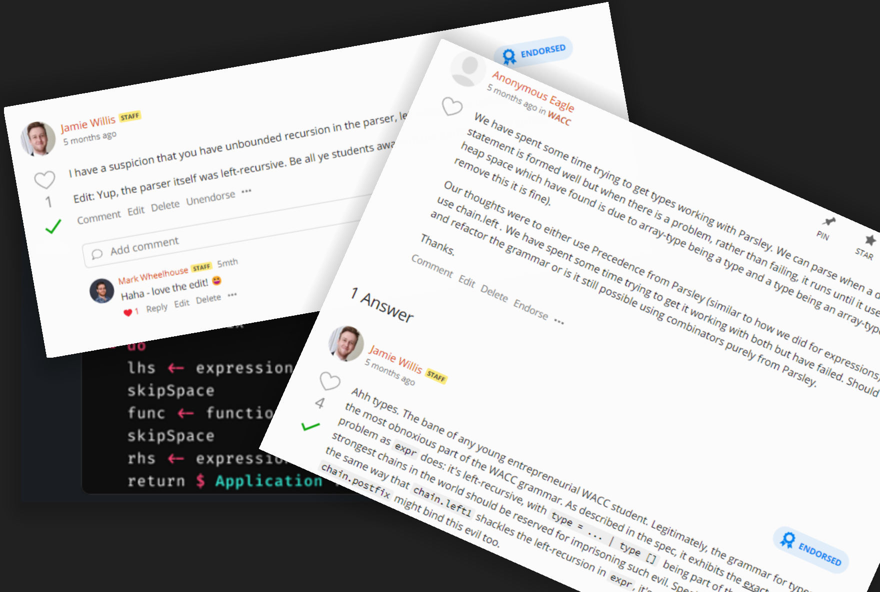
image.png

```haskell
applicationInfix :: Parser Expression
applicationInfix
  = do
    lhs ← expression
    skipSpace
    func ← function
    skipSpace
    rhs ← expression
    return $ Application func [lhs, rhs]
```

A Real Problem

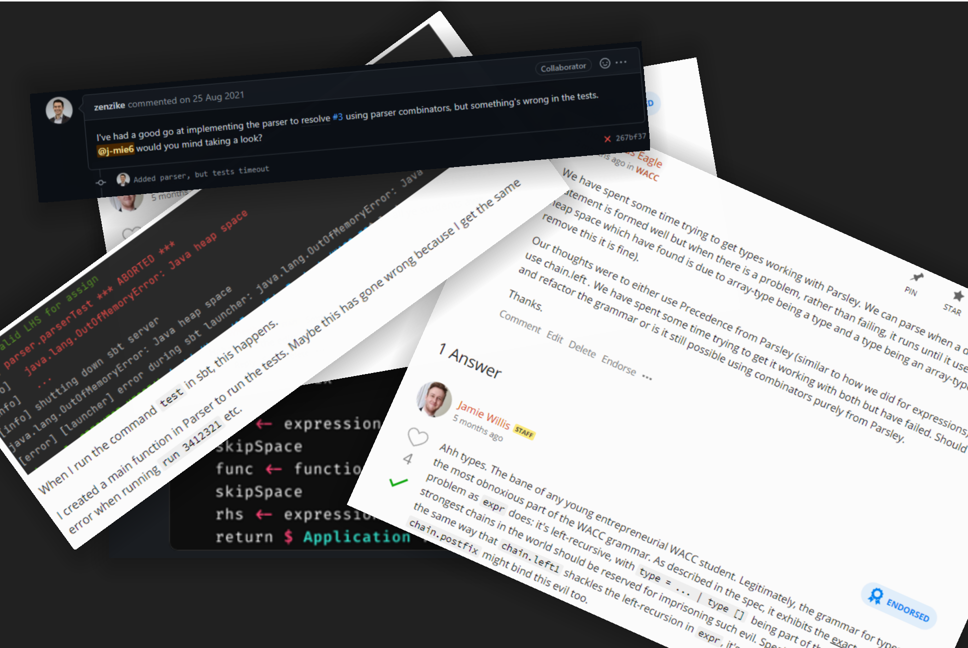


A Real Problem

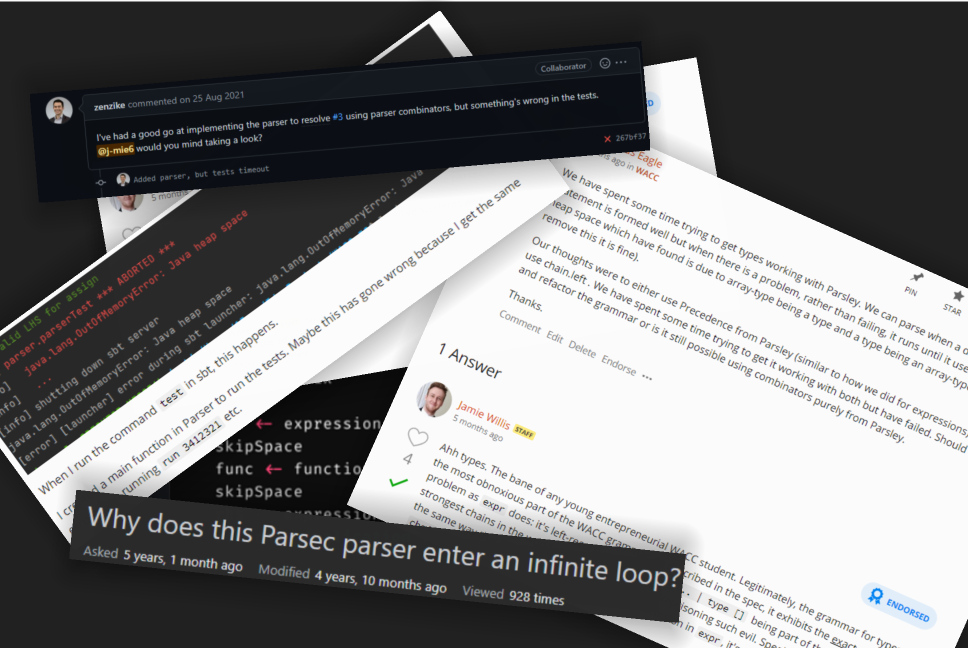


A Real Problem

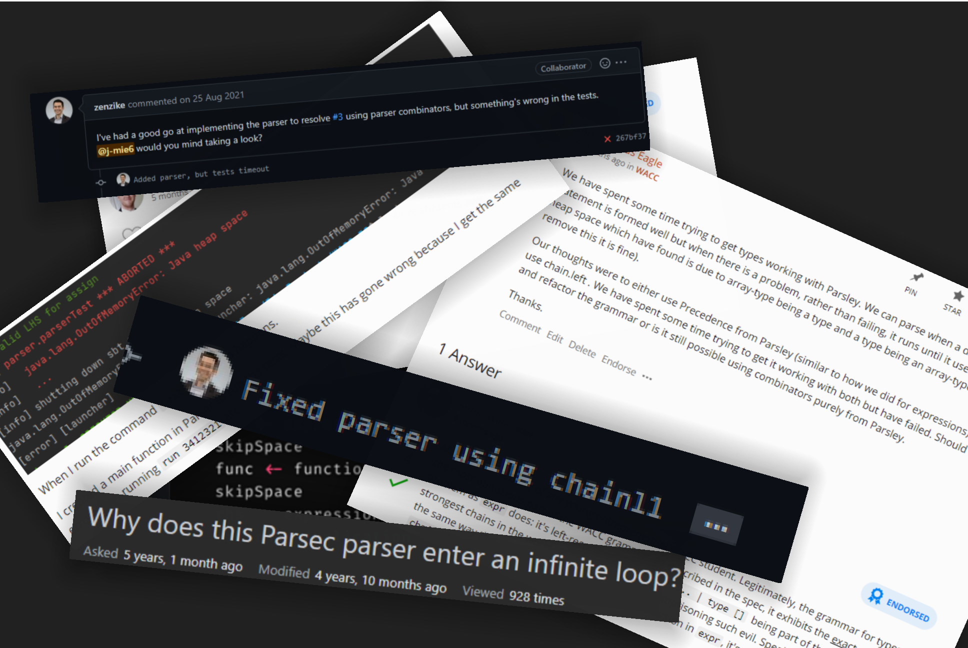
A Real Problem



A Real Problem



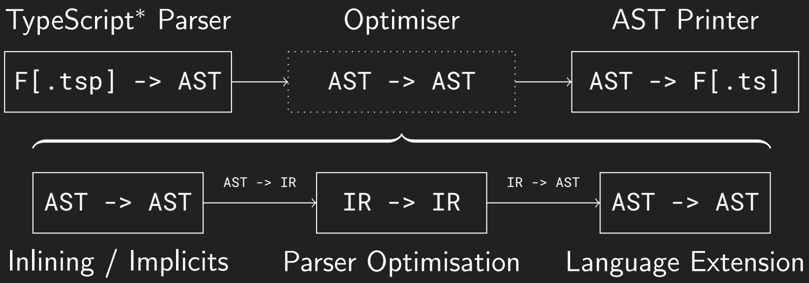
A Real Problem



Demo

The Preprocessor

Pipeline


TypeScript* Parser
Optimiser
AST Printer
F[.tsp] -> AST
AST -> AST
AST -> F[.ts]
AST -> AST
AST -> IR
IR -> IR
IR -> AST
AST -> AST
Inlining / Implicits
Parser Optimisation
Language Extension

Extending TypeScript

Main differences from TypeScript:

Extending TypeScript

Main differences from TypeScript:

- **New declarations and laziness**

```
lazy p = p;
val q = x;
```

Extending TypeScript

Main differences from TypeScript:

- **New declarations and laziness**

```
const p = lazy(() => p);
const q = x;
```

Extending TypeScript

Main differences from TypeScript:

- New declarations and laziness
- **User-defined operators**

```
let p = a <~> b;
let q = a *> b *> c;
let r = a <*> b <|> c <*> d;
let s = a <***> (pf <* b) <*> c;
let t = a <xyz> b;
```

Extending TypeScript

Main differences from TypeScript:

- New declarations and laziness
- **User-defined operators**

```
let p = mult(q, r);
let q = apR(apR(a, b), c);
let r = choice(ap(a, b), ap(c, d));
let s = ap(pa(a, apL(pf, b)), c);
let t = xyz(a, b);
```

Extending TypeScript

Main differences from TypeScript:

- New declarations and laziness
- User-defined operators
- ‘Macros’

```
inline paap(x, y, z) = x <***> y <**> z;
inline o = add <$ chr('+');
inline add = (x: number) => (y: number) => x + y;
let r = paap(a, o, b);
```

Extending TypeScript

Main differences from TypeScript:

- New declarations and laziness
- User-defined operators
- ‘**Macros**’

```
let r = ap(pa(a,
  constFmapL(
    (x: number) => (y: number) => x + y,
    chr('+')
)), b);
```

Removing Left Recursion

```
val nat = parseInt <$> /[0-9]+/;
lazy ex = (x => y => x+y) <$> (ex <* '+') <*> nat <|>
          ex <**> ((x => y => x-y) <$ '-') <*> nat <|>
          nat;
```

Removing Left Recursion

```
const nat = fmap(parseInt, re(/[0-9]+/));
const ex  = lazy(() => postfix(nat,
    choice(
        pamf(apR(chr('-'), nat), (y) => (x) => x - y),
        pamf(apR(chr('+'), nat), (y) => (x) => x + y)
    )
));
```

Rewriting

Rewriting is done in three steps:

Rewriting

Rewriting is done in three steps:

1. **Normalisation**

```
lazy e = (x => y => x + y) <$> (e <* '+') <*> n <|> n;
```

Rewriting

Rewriting is done in three steps:

1. **Normalisation**

```
lazy e = lift2(u(x => y => x + y), e <* '+', n) <|> n;
```

Rewriting

Rewriting is done in three steps:

1. Normalisation
2. **Reduction**

```
lazy e = lift2(u(x => y => x + y), e, '+' *> n) <|> n;
```

Rewriting

Rewriting is done in three steps:

1. Normalisation
2. Reduction
3. **Postfix rewrite**

```
lazy e = postfix(n,
    f(c(u(x => y => x + y))) <$> ('+' *> n)
);
```

Rewriting

Rewriting is done in three steps:

1. Normalisation
2. Reduction
3. **Postfix rewrite**

```
lazy e = postfix(n,
    (y => x => x + y) <$> ('+' *> n)
);
```

Conclusion

Contributions

Contributions:

Contributions

Contributions:

- A lightweight parser combinator library for TypeScript

Contributions

Contributions:

- A lightweight parser combinator library for TypeScript
- A pipeline for analysing and extending TypeScript

Contributions

Contributions:

- A lightweight parser combinator library for TypeScript
- A pipeline for analysing and extending TypeScript
- Techniques for detecting and refactoring left-recursion and general parser optimisations such as backtracking removal

Contributions

Contributions:

- A lightweight parser combinator library for TypeScript
- A pipeline for analysing and extending TypeScript
- Techniques for detecting and refactoring left-recursion and general parser optimisations such as backtracking removal
    - Correctness of these transformations have been reasoned about in the report

Related Works

This isn't a new problem:

Related Works

This isn't a new problem:

- *ANTLR4*: a parser generator that rewrites left-recursion

This isn't a new problem:

- *ANTLR4*: a parser generator that rewrites left-recursion
- Other TypeScript parser combinator libraries (*ts-parsec*, *parsimmon*)

Related Works

This isn't a new problem:

- *ANTLR4*: a parser generator that rewrites left-recursion
- Other TypeScript parser combinator libraries (*ts-parsec*, *parsimmon*)
- Baars & Swierstra's self inspecting parsers

Thanks for Listening!

What have we seen?

Thanks for Listening!

What have we seen?

- Parser combinators are an effective way of performing real parsing

Thanks for Listening!

What have we seen?

- Parser combinators are an effective way of performing real parsing
- TypeScript doesn't support it well

Thanks for Listening!

What have we seen?

- Parser combinators are an effective way of performing real parsing
- TypeScript doesn't support it well
- Left-recursion is a real problem that complicates parser combinators

Thanks for Listening!

What have we seen?

- Parser combinators are an effective way of performing real parsing
- TypeScript doesn't support it well
- Left-recursion is a real problem that complicates parser combinators
- These problems, and more, can be solved with a preprocessor