

# Teaspoon: Parser Combinators in Languages without User-Defined Operators

---

Eugene Lin

Monday 27<sup>th</sup> June, 2022

Imperial College London

# Parser Combinators

What even are parser combinators?

# Parser Combinators

What even are **parser** combinators?

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

# Parser Combinators

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

# Parser Combinators

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 ▾

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

# A Real Problem


**Jamie Willis** STAFF  
5 months ago


  
1


I have a suspicion that you have unbounded recursion in the parser, let's chat in the labs tomorrow!


Edit: Yup, the parser itself was left-recursive. Be all ye students aware of the horrors of left-recursion.

[Comment](#) [Edit](#) [Delete](#) [Unendorse](#) ...

 Add comment



**Mark Wheelhouse** STAFF Smith  
Haha - love the edit! 🤔

 [Reply](#) [Edit](#) [Delete](#) ...

```
lhs ← expression-  
skipSpace  
func ← function  
skipSpace  
rhs ← expression-  
return $ Application func [lhs, rhs]
```

## A Real Problem

[illegible]



# A Real Problem

**Terminal Screenshot:**

```
... build LMS for session
... parser.parseTest *** ABORTED ***
... java.lang.OutOfMemoryError: Java heap space
[info] shutting down sbt server
[info] java.lang.OutOfMemoryError: Java heap space
[error] [launcher] error during sbt launcher: java.lang.OutOfMemoryError: Java heap space
```

When I run the command `test` in sbt, this happens.  
I created a main function in Parser to run the tests. Maybe this has gone wrong because I get the same error when running `run 3412321` etc.

**Code Snippet:**

```
← expression
skipSpace
func ← function
skipSpace
rhs ← expression
return $ Application
```

**GitHub Comment:**

**Anonymous Eagle** 5 months ago in WACC  
We have spent some time trying to get types working with Parsley. We can parse when a `type` argument is formed well but when there is a problem, rather than failing, it runs until it uses heap space which have found is due to array-type being a type and a type being an array-type. remove this it is fine).

Our thoughts were to either use Precedence from Parsley (similar to how we did for expressions, use `chain.left`). We have spent some time trying to get it working with both but have failed. Should and refactor the grammar or is it still possible using combinators purely from Parsley.

Thanks.

Comment Edit Delete Endorse ...

**1 Answer**

**Jamie Willis** 5 months ago **STAFF**

4  
✓

Ahh types. The bane of any young entrepreneurial WACC student. Legitimately, the grammar for types is the most obnoxious part of the WACC grammar. As described in the spec, it exhibits the `expr` problem as `expr` does: it's left-recursive, with `type = ... | type [ ]` being part of the strongest chains in the world should be reserved for imprisoning such evil. Speaking of the same way that `chain.left` shackles the left-recursion in `expr`, it's `chain.postfix` might bind this evil too.

**ENDORSED**

# A Real Problem

The collage consists of several overlapping screenshots:

- GitHub Comment:** A comment by user `zenzike` from August 25, 2021. The text reads: "I've had a good go at implementing the parser to resolve #3 using parser combinators, but something's wrong in the tests. @j-mie6 would you mind taking a look?". Below the comment, a smaller snippet shows the text "Added parser, but tests timeout".
- Terminal Output:** A screenshot of a terminal window showing a series of error messages. The visible text includes: "ABORTED \*\*\*", "Java.Lang.OutOfMemoryError: Java heap space", "shutting down sbt server", "error during sbt launcher: java.lang.OutOfMemoryError: Java heap space", and "[launcher] error during sbt launcher: java.lang.OutOfMemoryError: Java heap space".
- Stack Overflow Answer:** An answer by user `Jamie Willis` (marked as a staff member) from 5 months ago. The answer has 4 votes and is marked as "ENDORSED". The text of the answer discusses the difficulty of implementing a parser for the WACC grammar, mentioning that the grammar is left-recursive and that the parser combinators used are purely from Parsley. It also mentions that the parser is implemented using a stack and that the grammar is not strictly left-recursive.

# A Real Problem

The collage consists of several overlapping elements:

- Top Left:** A GitHub comment from user `zenzike` dated 25 Aug 2021. The text reads: "I've had a good go at implementing the parser to resolve #3 using parser combinators, but something's wrong in the tests. @j-mie6 would you mind taking a look?". Below the comment, it says "Added parser, but tests timeout".
- Middle Left:** A screenshot of a terminal window showing error messages. The visible text includes: "ABORTED \*\*\*", "Java.Lang.OutOfMemoryError: Java heap space", and "error during sbt launcher: java.lang.OutOfMemoryError: Java heap space".
- Middle Right:** A Stack Overflow question titled "Why does this Parsec parser enter an infinite loop?". The question was asked 5 years, 1 month ago and modified 4 years, 10 months ago, with 928 views. The visible part of the question text says: "When I run the command `test` in sbt, this happens. I tried a main function in Parsec to run the tests. Maybe this has gone wrong because I get the same".
- Bottom Right:** A Stack Overflow answer by user `Jamie Willis` (marked as a staff member), dated 5 months ago. The answer has 4 votes and 1 answer. The visible text includes: "Our thoughts were to either use Precedence from Parsley. We can parse when a o...", "use chain.left. We have spent some time trying to get it working with both but have failed. Should and refactor the grammar or is it still possible using combinators purely from Parsley.", and "Ahh types. The bane of any young entrepreneurial WACC student. Legitimately, the grammar for types...".

# A Real Problem



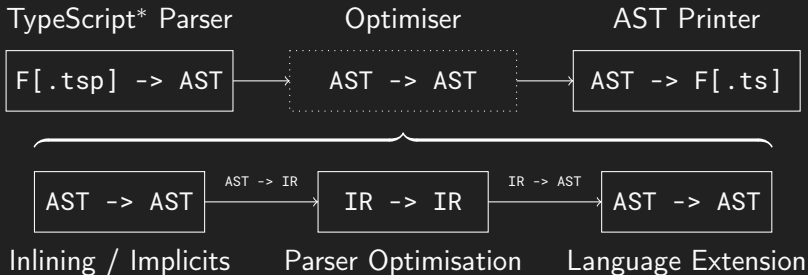
Demo

---

# The Preprocessor

---

# Pipeline



## 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

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*)

This isn't a new problem:

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