FIT2102 PASS - Week 11

Nicholas Cheng

October 25, 2020

What is a parser?

A parser is simply a function which takes a string as input and produces some structure or computation as output. — *Tim Dwyer*

What is a parser combinator?

A parser combinator is a higher-order function that accepts parsers as input and combines them somehow into a new parser. — Tim Dwyer

Let's take a step back

Think of languages as being a collection of smaller and smaller things.

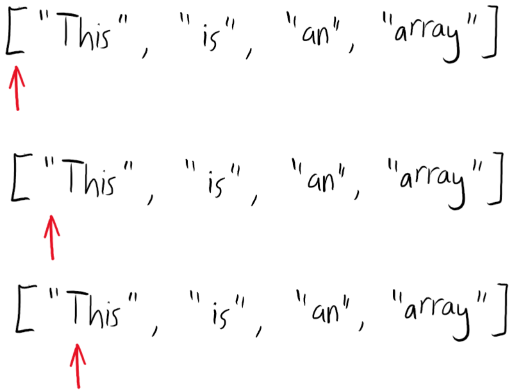
Letters $\rightarrow$ Words $\rightarrow$ Sentences $\rightarrow$ Paragraphs $\rightarrow \ldots$

How do we recognise words?

1. Imagine that you're just learning how to read.
2. You'd take your finger and start scanning from left to right[1].
3. You'd keep going until you hit a space, then you'd recognise that you just read a word.

[1]Or right to left, nothing against RTL languages

How about an illustration?



```
["This", "is", "an", "array"]
["This", "is", "an", "array"]
["This", "is", "an", "array"]
```

Okay, I admit, JSON isn't exactly a 'language' that you would immediately think of.

Fully parsed

["This", "is", "an", "array"]

JsonArray(JsonString , JsonString,
JsonString, JsonString)

Implementation

Let's start with just reading *one* character.

```
character :: Parser Char
character = P parser
  where
    parser "" = Error UnexpectedEof
    parser (c : s) = Result s c
```

Let’s run the character parser

```
> parse character "abc"
Result >bc< 'a'
> parse character ""
Unexpected end of stream
```

What's parse?

```
> :t parse
parse :: Parser a -> Input -> ParseResult a
```

It's a function defined on the `newtype` Parser. Use it with a parser to actually parse something.

Tutorial tip 1: const

If you find yourself doing this

```
f a = \_ -> a
```

Use `const` instead.

Tutorial tip 2: On point-free

Getting point-free code is just applying these operations over and over.

1. Converting infix operators to prefix notation
2. Operator sectioning
3. Function composition
4. Flipping an operator
5. Eta reduction

The goal is to turn them into forms where we can confidently apply our operations. Remember, **step by step**.

Tutorial tip 3: You can compose the compose operator

\a b c -> (a + b) * c

prefix \a b c -> (*) (a+b) c

eta \a b -> (*) (a +b)

operator sectioning \a b -> (*) ((a+) b)

compose \a b -> ((*) . (a+)) b

eta \a -> (*) . (a+)

prefix \a -> $\underbrace{(*)\ .}_{f}\ (\underbrace{(+)}_{g}\ \underbrace{a}_{x})$

compose \a -> [((*) .) . (+)] a

eta ((*) .) . (+)

Tutorial tip 4: Follow the types!

The type annotations tell you a lot about a function. Think in terms of how functions of different types can fit together. Also, use **typed holes**[2]!

[2]`https://wiki.haskell.org/GHC/Typed_holes`

Tutorial tip 5: Revise Monads

Some exercises need an intuition for the Monad typeclass and its operations. If you're confused about the Monad typeclass and its operators, ask me on Slack or now.

Assignment 2

GLHF. Just kidding! Reach out to me or the teaching staff early and often.