Parsing and Algebraic Design

Christopher R. Genovese

Department of Statistics & Data Science

Thu 20 Nov 2025
Session #23

Plan

Parsing

Plan

Parsing

What is a Parser?

A **parser** takes unstructured data as input and returns a structured representation of the data as output.

Classic examples:

- a programming language as text is converted to a *syntax tree* describing the code
- a text file in CSV format converted to a data frame,
- a series of network packets containing JSON converted to a nested record/dict.

Parsing has wide and frequent application in practice, from the small to the large.

What is a Parser?

Parsers and the operations that act on them give *meaning* to structured representations of data.

Here, we will use parsers as a mechanism to think about programming approaches, patterns, and design.

A key idea is to think of the **algebra** of our design: the building blocks and essential operations of our problem domain and the *laws* that govern their interaction. We want to design at the proper level of abstraction to achieve correctness, efficiency, and reusability.

To this end, we will build context-sensitive parsers out of smaller modular units called ***combinators***.

The design of our parser combinators will be motivated by an **algebra** of operations: a collection of functions that operate on specified types and associated *laws* that govern their operation.

Initial Explorations

We start by reasoning about our design's building blocks so we can appreciate their properties and relationships.

We hold off on writing code until our understanding is deeper, but we try to identify types, concepts, and entities that can refine our abstraction.

Initial Explorations

As a basic example, let us consider parsing a language for a **stack machine**. Some examples:

Input	Result
`3 4 +`	7
`10 10 10 * +`	110
`4 5 -`	-1
`4 5 swap -`	1
`4 dup dup * *`	64
`4 $a dict nil`	nil
`4 $a dict $a 2 *`	8
`"abc" "def" sconcat`	"defabc"

Do you see how it works? The result is the *value* left on top of the stack.

We have values, operators, functions, and literals. As they are read, they are pushed on a stack or executed with the stack.

Initial Explorations

Basic elements:

- nil value
- natural numbers
- Booleans `#true`, `#false`
- strings
- floating point numbers
- Symbols `$a`
- Procedures (i.e., "Quoted" code) such as `{swap}` or `{4 4 +}`
- dictionaries (produced with `dict`)
- typed operators
  arithmetic `+`, `*`, `-`, `/`, `^`, Logical `<`, `>`, `=`, `...`, —Boolean `and`, `or`, `not`, String, ...
- stack functions `dup`, `swap`, `pop`, `roll`, `rotate`, `...`
- other functions `def`, `call`, `begin`, `end`, `if`, `while`, `...`
- ...

There is a separate dictionary stack for managing scope via `scope` and `descope`.

Initial Questions

1. What does a parser act on?

Initial Questions

1. What does a parser act on?

   It takes some input as low level "tokens" and synthesizes the input into a structured representation via a set of syntactic rules.

   For our stack machine, the tokens are characters obtained from some *source*.

Initial Questions

2. What does a parser produce?

Initial Questions

2. What does a parser produce?

   If the input is "properly formed" according to the syntactic rules, it produces *at least* a structured representation of the input. The rest depends on our goals and use case.

   Do we require that the entire input be processed (total), or do we allow valid parses that leave some input remaining (partial)? Do we expect only a single valid parse of the input (deterministic), or do we want to see all possible parses according to the rules (non-deterministic)?

Initial Questions

2. What does a parser produce?

   If the input is "properly formed" according to the syntactic rules, it produces *at least* a structured representation of the input. The rest depends on our goals and use case.

   Do we require that the entire input be processed (total), or do we allow valid parses that leave some input remaining (partial)? Do we expect only a single valid parse of the input (deterministic), or do we want to see all possible parses according to the rules (non-deterministic)?

   For now, we will consider simple tokens, but this can be extended.

```
type Token = Char       -- For now, alternatives are possible
type Tokens = String    -- Tokens is "made of" Token
```

   Note that the `Tokens` type does not specify the *source* of the tokens.

   Our parser type will be a computational context by which we turn `Tokens` into a value. So our basic type might look like one of

```
Parser a        -- a is the type of the produced value
Parser t a      -- t is the token type
```

   Here, we will use the first in practice as we are restricting tokens as above.

Initial Questions

3. Can our inputs be decomposed into non-interacting parts?

Initial Questions

3. Can our inputs be decomposed into non-interacting parts?

   In some applications, this is possible, e.g., command-line options.

   But in our stack machine, sequencing matters, so in general it cannot be easily decomposed.

   This suggests that our parser context be a **monad** rather than just an applicative functor.

   The monad (and thus applicative and thus functor) gives us some initial combinators automatically:

```
map  : (a -> b) -> Parser a -> Parser b
pure : a -> Parser a
map2 : (a -> b -> c) -> Parser a -> Parser b -> Parser c
bind : Parser a -> (a -> Parser b) -> Parser b
```

   Here, `bind` glues two "programs" together.

Primitive Operations

With our stack machine, consider (separate) inputs +, 248 and ▢, and assume the latter is an invalid token.

We need a way to read a token from the input, and we need a way to *fail*. This suggests primitives

```
single : Parser Token
fail   : Parser a
```

The first reads a token and returns it; the second always fails. (Note that `pure t` always succeeds and returns `t`, without consuming any tokens.)

Our parser also needs to be able to distinguish its processing of + and 248. We need to be able to *choose*. This suggests another primitive:

```
choice : Parser a -> Parser a -> Parser a
```

This constructs a parser from two other parsers. It can try one first then the other only if the first fails (left biased) or try both in parallel (fair), as we decide.

We might also want a primitive `peek : Int -> Parser Tokens` that gives look-ahead at the next (several) token(s).

Some Combinators

How can we use the combinators so far to parse a natural number? Try it, just a sketch. Create combinators:

```
tokenSatisfies : (Token -> Bool) -> Parser Token
many : Parser a -> Parser (List a)
some : Parser a -> Parser (NonEmptyList a)
seq : Parser a -> Parser b -> Parser (a, b)
digit : Parser Natural
natural : Parser Natural
char : Token -> Parser Token          -- looks for a specific character
charIn : Set Token -> Parser Token    -- looks for a char in specified set
```

Note that we don't yet have a way method yet to "run" a parser. That's ok!

Some Combinators

Here's a sketch in TL1, but it's just as easy, if less succinct, in Python or R.

```
tokenSatisfies p = single `bind` (\t -> if p t then pure c else fail)

many p = (some p) `choice` pure []
some p = map2 (::) p (many p)

seq p q = p `bind` (\a -> q `bind` (\b -> pure (a, b)))     -- blech
seq p q = do                                                -- better!
  a <- p
  b <- q
  pure (a, b)

digit = map (\c -> pure (ord c - ord '0')) (tokenSatisfies isDigit)
natural = map (fold (\n d -> 10 * n + d) 0) (some digit)
char c = tokenSatisfies (== c)
```

First Try: Stack Machine Parser

Let's start with a simple version of our stack machine. Items in the input/on the stack can be:

- natural numbers
- operators `+`, `*`, `-`, `/`, `^`, `mod`, `and`, `not`, `or`
- Booleans `#true`, `#false`
- functions `def`, `rotate`, `dup`, `swap`, `pop`, `if`, `dict`
- Symbols `$a`
- Procedures `{swap}` or `{4 4 +}`
- ...

In your language, write a parser that handles at least the first two. Write additional combinators as needed. You can use the provided stubs in documents/parsing if you like, but best for now to try something.

What type of value should your parser return? `Parser ???`

Interpretations (Total, Non-Deterministic, Parallel)

Remember:

```
data Program : (Type -> Type) -> Type -> Type where
  Done : result -> Program instr result
  Then : instr result
         -> (result -> Program instr result')
         -> Program instr result'
```

This suggests (for total, non-deterministic parsing):

```
data ParseInstruction a where
  Single : ParseInstruction Token
  Fail   : ParseInstruction a
  Choice : Parser a -> Parser a -> ParseInstruction a

type Parser a = Program ParseInstruction a

interpret : Parser a -> Tokens -> List a
interpret (Done a)                ts = if empty ts then [a] else []
interpret (Single `Then` is)      ts = case ts of
                                         (c :: cs) -> interpret (is c) cs
                                         []        -> []
interpret (Fail `Then` is)        ts = []
interpret (Choice p q `Then` is) ts = interpret (p `bind` is) ts
                                      ++ interpret (q `bind` is) ts
```

Interpretations (Partial, Deterministic, Left-Biased)

In many practical cases, we may not need multiple possible parses, only one, or perhaps only one that satisfies some criterion. In such cases, a deterministic parser will be more efficient. And we may want to find a valid parse even without using up all the input.

In this case, our main parsing interpreter might look like

```
type Parser a = Tokens -> Maybe (a, Tokens)
```

or more generally

```
data ParseState = record ParseState where
                    source   : Tokens
                    position : Position

type ParseResult a = Either (Failure Info) (Success a ParseState)

type Parser a = ParseState -> ParseResult a

parse : Tokens -> Parser a -> ParseResult a
```

Next Steps

1. **Extending Our Stack Machine Parser**

   Extend your stack machine parsers. Show that you can parse simple programs.

   Pick a style and implement an interpreter.

2. **Try implementing some combinators**

   See `combinators.py` and `combinators.r`, stub versions.

   Demo from full combinators code. See Exercises within.

3. In non-det parser, the parser is parallel but still depth first, which has memory and other implications. Consider the equation (a "distributive law"):

```
    (single `bind` is) `choice` (single `bind` js)
  =  single `bind` (\c -> (is c) `choice` (js c))
```

   This let's us fetch one token at a time and feed it to both parsers, and it works just as well for any number of parallel choices.

   Use this "fusion" equation to make `choice` breadth-first in this way.

```
dist :: Parser a -> [Parser a]
dist (Choice p q `Then` is) = dist (p `bind` is) ++ dist (q `bind` is)
dist (Fail       `Then` is) = []
dist p                      = [p]
```

The End