Parsing with Derivatives

A Functional Pearl

Matthew Might David Darais
University of Utah
might@cs.utah.edu, david.darais@gmail.com

Daniel Spiewak
University of Wisconsin, Milwaukee
dspiewak@uwm.edu

Abstract

We present a functional approach to parsing unrestricted context-free grammars based on Brzozowski's derivative of regular expressions. If we consider context-free grammars as recursive regular expressions, Brzozowski's equational theory extends without modification to context-free grammars (and it generalizes to parser combinators). The supporting actors in this story are three concepts familiar to functional programmers—laziness, memoization and fixed points; these allow Brzozowski's original equations to be transliterated into purely functional code in about 30 lines spread over three functions.

Yet, this almost impossibly brief implementation has a drawback: its performance is sour—in both theory *and* practice. The culprit? Each derivative can *double* the size of a grammar, and with it, the cost of the next derivative.

Fortunately, much of the new structure inflicted by the derivative is either dead on arrival, or it dies after the very next derivative. To eliminate it, we once again exploit laziness and memoization to transliterate an equational theory that prunes such debris into working code. Thanks to this compaction, parsing times become reasonable in practice.

We equip the functional programmer with two equational theories that, when combined, make for an abbreviated understanding and implementation of a system for parsing context-free languages.

Categories and Subject Descriptors F.4.3 [*Formal Languages*]: Operations on languages

General Terms Algorithms, Languages, Theory

Keywords formal languages, parsing, derivative, regular expressions, context-free grammar, parser combinator

1. Introduction

It is easy to lose sight of the essence of parsing in the minutiae of forbidden grammars, shift-reduce conflicts and opaque action tables. To the extent that understanding in computer science comes from implementation, a deeper appreciation of parsing often seems out of reach. Brzozowski's derivative upsets this calculus of effort and understanding to make the construction of parsing systems accessible to the common functional programmer.

The derivative of regular expressions [1], if gently tempered with laziness, memoization and fixed points, acts immediately as a pure, functional technique for generating parse forests from arbitrary context-free grammars. Despite—even because of—its simplicity, the derivative transparently handles ambiguity, left-recursion, right-recursion, ill-founded recursion or any combination thereof.

1.1 Outline

- After a review of formal languages, we introduce Brzozowski's derivative for regular languages. A brief implementation highlights its rugged elegance.
- As our implementation of the derivative engages context-free languages, non-termination emerges as a problem.
- Three small, surgical modifications to the implementation (but not the theory)—laziness, memoization and fixed points—guarantee termination. Termination means the derivative can recognize arbitrary context-free languages.
- We generalize the derivative to parsers and parser combinators through an equational theory for generating parse forests.
- We find poor performance in both theory and practice. The root cause is vestigial structure left in the grammar by earlier derivatives; this structure is malignant: though it no longer serves a purpose, it still grows in size with each derivative.
- We develop an optimization—compaction—that collapses grammars by excising this mass. Compaction, like the derivative, comes from a clean, equational theory that exploits laziness and memoization in its transliteration to working code.

In this article, we provide code in Racket, but it should adapt readily to any Lisp. All code and test cases within or referenced from this article (plus additional implementations in Haskell and Scala) are available from:

`http://www.ucombinator.org/projects/parsing/`

2. Preliminary: Formal languages

A language L is a set of strings. A string w is a sequence of characters from an alphabet A. (From the parser's perspective, a "character" might be a token/terminal.)

Two atomic languages arise often in formal languages: the empty language and the null (or empty-string) language:

- The empty language $\emptyset$ contains no strings at all:
$$\emptyset = \{\}.$$
- The null language ϵ contains only the length-zero "null" string:
$$\epsilon = \{w\} \text{ where } \mathit{length}(w) = 0.$$

Or, using C notation for strings, $\epsilon = \{\texttt{""}\}$. For convenience, we may use the symbol ϵ to refer to both the null language and the null string.

Given an alphabet A, there is a singleton language for every character c in that alphabet. Where it is clear from context, we use the character itself to denote that language; that is:

$$c \equiv \{c\}\,.$$

2.1 Operations on languages

Because languages are sets, set operations like union apply:

$$\{\texttt{foo}\} \cup \{\texttt{bar}, \texttt{baz}\} = \{\texttt{foo}, \texttt{bar}, \texttt{baz}\}\,.$$

Concatenation ($\circ$) appends the product of the two languages:

$$L_1 \circ L_2 = \{w_1 w_2 : w_1 \in L_1 \text{ and } w_2 \in L_2\}\,.$$

The nth power of a language is the set of strings of n consecutive words from that language:

$$L^n = \{w_1 w_2 \ldots w_n : w_i \in L \text{ for } 1 \leq i \leq n\}\,.$$

And, the non-empty repetition of a language (its Kleene star) is the infinite union of all its powers:

$$L^\star = \bigcup_{i=0}^{\infty} L^i.$$

2.2 Regular languages and context-free languages

If a language is non-recursively definable from atomic sets using only union, concatenation and repetition, that language is regular.

If we allow mutually recursive definitions, then the set of describable languages is exactly the set of context-free languages. (Even without Kleene star, the resulting set of languages is context-free.) We assume, of course, a least-fixed-point interpretation of such recursive structure. For instance, given the language L:

$$L = (\{\texttt{x}\} \circ L) \cup \epsilon.$$

The least-fixed-point interpretation of L is a set containing a finite string of every length (plus the null string). Every string contains only the character `x`. [The greatest-fixed-point interpretation of L adds an infinite string of `x`'s.]

2.3 Encoding languages

To represent the atomic and complex languages in code, there is a `struct` for each kind of language:

```
(define-struct empty {})         ; ∅
(define-struct eps   {})         ; ϵ
(define-struct char  {value})

(define-struct cat {left right}) ; left ∘ right
(define-struct alt {this that})  ; this ∪ that
(define-struct rep {lang})       ; lang⋆
```

Example In code, the language:

$$\begin{aligned} L_{ab} = {} & L_{ab} \circ \{\texttt{a}, \texttt{b}\} \\ & \cup \epsilon, \end{aligned}$$

becomes:

```
(define L (alt (cat L (alt (char 'a) (char 'b)))
               (eps)))
```

3. Brzozowski's derivative

Brzozowski defined the derivative of regular expressions in his work on the recognition of regular languages [1]. The derivative of a language L with respect to a character c is a new language that has been "filtered" and "chopped"—$D_c(L)$:

1. First, retain only the strings that start with the character c.
2. Second, chop that first character off every string.

Formally:

$$D_c(L) = \{w : cw \in L\}\,.$$

Examples

$$\begin{aligned} D_{\texttt{b}}\,\{\texttt{foo}, \texttt{bar}, \texttt{baz}\} &= \{\texttt{ar}, \texttt{az}\} \\ D_{\texttt{f}}\,\{\texttt{foo}, \texttt{bar}, \texttt{baz}\} &= \{\texttt{oo}\} \\ D_{\texttt{a}}\,\{\texttt{foo}, \texttt{bar}, \texttt{baz}\} &= \emptyset. \end{aligned}$$

3.1 Recognition with the derivative

The simplicity of the derivative's definition masks its power. If one can compute successive derivatives of a language, it is straightforward to determine the membership of a string within a language, thanks to the following property:

$$cw \in L \text{ iff } w \in D_c(L).$$

To determine membership, derive a language with respect to each character, and check if the final language contains the null string: if yes, the original string was in; if not, it wasn't.

3.2 A recursive definition of the derivative

Brzozowski noted that the derivative is closed over regular languages, and admits a recursive implementation:

- For the atomic languages:

$$\begin{aligned} D_c(\emptyset) &= \emptyset \\ D_c(\epsilon) &= \emptyset \\ D_c(c) &= \epsilon \\ D_c(c') &= \emptyset \text{ if } c \neq c'. \end{aligned}$$

- For the derivative over union:

$$D_c(L_1 \cup L_2) = D_c(L_1) \cup D_c(L_2).$$

- The derivative over Kleene star peels off a copy of the language:

$$D_c(L^\star) = D_c(L) \circ L^\star.$$

- For the derivative of concatenation, we must consider the possibility that the first language could be null:

$$\begin{aligned} D_c(L_1 \circ L_2) &= D_c(L_1) \circ L_2 \text{ if } \epsilon \notin L_1 \\ D_c(L_1 \circ L_2) &= (D_c(L_1) \circ L_2) \cup D_c(L_2) \text{ if } \epsilon \in L_1 \end{aligned}$$

  We can express concatenation without a conditional through the nullability function: δ. This function returns the null language if its input language contains the null string, and the empty set otherwise:

$$\begin{aligned} \delta(L) &= \emptyset \text{ if } \epsilon \notin L \\ \delta(L) &= \epsilon \text{ if } \epsilon \in L. \end{aligned}$$

  Thus, we can equivalently define concatenation:

$$D_c(L_1 \circ L_2) = (D_c(L_1) \circ L_2) \cup (\delta(L_1) \circ D_c(L_2)).$$

3.3 Nullability of regular languages

Conveniently, nullability may also be computed using structural recursion on regular languages:

$$\begin{aligned}
\delta(\emptyset) &= \emptyset \\
\delta(\epsilon) &= \epsilon \\
\delta(c) &= \emptyset \\
\delta(L_1 \cup L_2) &= \delta(L_1) \cup \delta(L_2) \\
\delta(L_1 \circ L_2) &= \delta(L_1) \circ \delta(L_2) \\
\delta(L^\star) &= \epsilon.
\end{aligned}$$

A recursive implementation of the Boolean variant of the nullability function is straightforward:

```
(define (δ L)
  (match L
    [(empty)       #f]
    [(eps)         #t]
    [(char _)      #f]

    [(rep _)       #t]
    [(alt L1 L2)   (or  (δ L1) (δ L2))]
    [(cat L1 L2)   (and (δ L1) (δ L2))]))
```

Examples A couple examples illustrate the derivative on regular languages:

$$D_{\mathtt{f}}\,\{\mathtt{foo},\mathtt{bar}\}^\star = \{\mathtt{oo}\} \circ \{\mathtt{foo},\mathtt{bar}\}^\star$$
$$D_{\mathtt{f}}\,\{\mathtt{foo},\mathtt{bar}\}^\star \circ \{\mathtt{frak}\} = \{\mathtt{oo}\} \circ \{\mathtt{foo},\mathtt{bar}\}^\star \circ \{\mathtt{frak}\} \cup \{\mathtt{rak}\}\,.$$

3.4 An implementation of the derivative

As the description of a regular language is not recursive, it is straightforward to transliterate the derivative into working code:

```
(define (D c L)
  (match L
    [(empty)       (empty)]
    [(eps)         (empty)]
    [(char a)      (if (equal? c a)
                       (eps)
                       (empty))]

    [(alt L1 L2)   (alt (D c L1)
                        (D c L2))]
    [(cat (and (? δ) L1) L2)
                   (alt (D c L2)
                        (cat (D c L1) L2))]
    [(cat L1 L2)   (cat (D c L1) L2)]
    [(rep L1)      (cat (D c L1) L)]))
```

Matching a regular language `L` against a `consed` list of characters `w` is straightforward:

```
(define (matches? w L)
  (if (null? w)
      (δ L)
      (matches? (cdr w) (D (car w) L))))
```

4. Derivatives of context-free languages

Since a context-free language is a recursive regular language, it is tempting to use the same code for computing the derivative. From the perspective of parsing, this has two chief drawbacks:

1. It doesn't work.
2. It wouldn't produce a parse forest even if it did.

The first problem comes from the recursive implementation of the derivative running into the recursive nature of context-free grammars. It leads to non-termination.

The second comes from the fact that our regular implementation *recognizes* whether a string is in a language rather than parsing the string. We tackle the termination problem in this section, and the parsing problem in the next.

Example Consider the following left-recursive language:

$$\begin{aligned} L = {} & L \circ \{\mathtt{x}\} \\ & \cup\, \epsilon. \end{aligned}$$

If we take the derivative of L, we get a new language:

$$\begin{aligned} D_{\mathtt{x}}L = {} & D_{\mathtt{x}}L \circ \{\mathtt{x}\} \\ & \cup\, \epsilon. \end{aligned}$$

Mathematically, this is sensible. Computationally, it is not. The code from the previous section recurs forever as it attempts to compute the derivative of the language L.

4.1 Step 1: Laziness

Preventing the implementation of the derivative from making an infinite descent on a recursive grammar requires targeted laziness. Specifically, it requires making the fields of the structs `cat`, `alt` and `rep` by-need.[1] With by-need fields, the computation of any (potentially self-referential) derivatives in those fields gets suspended until the values in those fields are required.

4.2 Step 2: Memoization

With laziness, we can compute the derivative until it requires nullability (as in concatenation or testing membership). Nullability eagerly walks the structure of the entire language. Thus, nullability fails to terminate on a derived language such as the one above. We need the derivative to return a finite (if lazily explored) graph. By memoizing the derivative, it "ties the knot" when it re-encounters a language it has already seen:

```
(define/memoize (D c L)
  #:order ([L #:eq] [c #:equal])
  (match L
    [(empty)       (empty)]
    [(eps)         (empty)]
    [(char a)      (if (equal? a c)
                       (eps)
                       (empty))]

    [(alt L1 L2)   (alt (D c L1)
                        (D c L2))]
    [(cat (and (? δ) L1) L2)
                   (alt (D c L2)
    [(cat L1 L2)   (cat (D c L1) L2)]
    [(rep L1)      (cat (D c L1) L)]))
```

The `define/memoize` form above defines a derivative function `D` that memoizes first by pointer equality on the language and then by value equality on the character.

4.3 Step 3: Fixed points

The computation of nullability is more challenging than the computation of the derivative because it isn't looking for a structure; it's looking for a single answer: "Yes, it's nullable," or "No, it's

[1] Lisp implementations that do not support lazy fields have to provide them transparently with macros, `delay` and `force`.

not." As such, laziness and memoization can't help side-step self-dependencies the way they did for the derivative. Consider the nullability of the left-recursive language L:

$$\delta(L) = (\delta(L) \circ \emptyset) \cup \epsilon.$$

To know the nullability of L requires knowing the nullability of L. For decades, this problem has been solved by interpreting the nullability of L as the least fixed point of the nullability equations.

To bare only the essence of nullability, we can hide the computation of a least fixed point behind a purely functional abstraction: `define/fix`. The `define/fix` form uses Kleene's theorem to compute the least fixed point of a monotonic recursive definition, and it allows the prior definition of nullability to be used with little change:

```
(define/fix (δ L)
  #:bottom #f
  (match L
    [(empty)       #f]
    [(eps)         #t]
    [(char _)      #f]

    [(rep _)       #t]
    [(alt L1 L2)   (or  (δ L1) (δ L2))]
    [(cat L1 L2)   (and (δ L1) (δ L2))]))
```

The `#:bottom` keyword indicates from where to begin the iterative ascent toward the least fixed point.

The `define/fix` form defines a function mapping nodes in a graph (V, E) to values in a lattice X, so that given an instance:

(define/fix (f v) #:bottom $\bot_X$ *body*)

After this definition, the function $f : V \to X$ is a least fixed point:

$$f = \text{lfp}(\lambda f.\lambda v.\mathit{body}),$$

which is easily computed with straightforward iteration:

$$\text{lfp}(F) = F^n(\bot_{V\to X}) \text{ for some finite } n.$$

4.4 Recognizing context-free languages

No special modification is required for the `matches?` function. It works as-is for recognizing context-free languages.

With access to laziness, memoization and a facility for computing fixed points, we were able to construct a system for recognizing any context-free language in less than 30 lines of code.

5. Parsers and parser combinators

Using standard techniques from functional programming, we lifted the derivative from regular languages to context-free languages. If *recognition* of strings in context-free languages were our goal, we would be done.

But, our goal is parsing. So, our next step is to generalize the derivative to parsers. This section reviews parsers and parser combinators. (For a more detailed treatment, we refer the reader to [15, 16].) In the next section, we explore their derivative.

A partial parser p is a function that consumes a string and produces "partial" parses of that string. A partial parse is a pair containing the remaining unparsed input, and a parse tree for the prefix. The set $\mathbb{P}(A, T)$ contains the partial parsers over alphabet A that produce parse trees in the set T:

$$\mathbb{P}(A, T) \subseteq A^* \to \mathcal{P}(T \times A^*).$$

A (full) parser p consumes a string and produces all possible parses of the full string. The set $\lfloor\mathbb{P}\rfloor(A, T)$ contains the full parsers over alphabet A that produce parse trees in the set T:

$$\lfloor\mathbb{P}\rfloor(A, T) \subseteq A^* \to \mathcal{P}(T).$$

Of course, we can treat a partial parser $p \in \mathbb{P}(A, T)$ as a full parser:

$$\lfloor p\rfloor(w) = \{t : (t, \epsilon) \in p(w)\},$$

by discarding any partial parse that did not exhaust the input.

5.1 Simple parsers

Simple languages can be implicitly promoted to partial parsers:

- A character c converts into a partial parser for exactly itself:

$$c \equiv \lambda w. \begin{cases} \{(c, w')\} & w = cw' \\ \emptyset & \text{otherwise.} \end{cases}$$

- The null string becomes the consume-nothing parser:

$$\epsilon \equiv \lambda w. \{(\epsilon, w)\}.$$

- The empty set becomes the reject-everything parser:

$$\emptyset \equiv \lambda w. \{\}.$$

5.2 Combining parsers

Parsers combine in the same fashion as languages:

- The union of two parsers, $p, q \in \mathbb{P}(A, X)$, combines all parse trees together, so that $p \cup q \in \mathbb{P}(A, X)$:

$$p \cup q = \lambda w.p(w) \cup q(w).$$

- The concatenation of two parsers, $p \in \mathbb{P}(A, X)$ and $q \in \mathbb{Q}(A, Y)$, produces a parser that pairs the parse trees of the individual parsers together, so that $p \circ q \in \mathbb{P}(A, X \times Y)$:

$$p \circ q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\}$$

  In effect, the first parser consumes a prefix of the input and produces a parse tree. It passes the remainder of that input to the second parser, which produces another parse tree. The result is the left-over input paired with both of those parse trees.

- A reduction by function $f : X \to Y$ over a parser $p \in \mathbb{P}(A, X)$ creates a new partial parser, $p \to f \in \mathbb{P}(A, Y)$:

$$p \to f = \lambda w.\{((f(x), w') : (x, w') \in p(w)\}$$

  A reduction parser maps trees from X into trees from Y.

  In code, a new struct represents reduction parsers:

```
(define-struct red {lang f})
```

  Once again, the field `lang` should be lazy.

5.3 The nullability combinator

A special nullability combinator, δ, simplifies the definition of the derivative over parsers. It becomes a reject-everything parser if the language cannot parse empty, and the null parser if it can:

$$\delta(p) = \lambda w. \{(t, w) : t \in \lfloor p\rfloor(\epsilon)\}.$$

We can add a new kind of language node to represent these:

```
(define-struct δ {lang})
```

Once again, the field `lang` is lazy. (Please note that δ is no longer the *function* from the previous section.)

5.4 The null reduction parser

To implement the derivative of parsers for single characters: the null reduction partial parser, $\epsilon \downarrow S$, is handy. This parser can only parse the null string; it returns a set of parse trees stored within:

$$\epsilon \downarrow S \equiv \lambda w. \{(t, w) : t \in S\}.$$

A new struct provides null-reduction nodes:

```
(define-struct eps* {trees})
```

5.5 The repetition combinator

It is easiest to define the Kleene star of a partial parser $p \in \mathbb{P}(A, T)$ in terms of concatenation, union and reduction, so that $p^\star \in \mathbb{P}(A, T^*)$:

$$\begin{aligned} p^\star &= (p \circ p^\star) \rightarrow \lambda(\mathit{head}, \mathit{tail}).\mathit{head} : \mathit{tail} \\ &\quad \cup \epsilon \downarrow \{\langle\rangle\}\,. \end{aligned}$$

The colon operator (:) is the sequence constructor, and $\langle\rangle$ is the empty sequence.

6. Derivatives of parser combinators

If we can generalize the derivative *to* parsers and *over* parser combinators, then we can construct parse forests using derivatives. But first, we must consider the question:

"What *is* the derivative of a parser?"

Intuitively, the derivative of a parser with respect to the character c should be a new parser. It should have the same type as the original parser; that is, if the original parser consumed the alphabet A to construct parse trees of type X, then the new parser should do the same. Formally:

$$D_c : \mathbb{P}(A, T) \rightarrow \mathbb{P}(A, T).$$

But, how should the derived parser behave?

It should act as though the character c has been consumed, so that if the string w is supplied, it returns parses for the string cw. However, it also needs to strip away any null parses that come back. If it didn't strip these, then null parses containing cw would return when trying to parse w with the derived parser. It is nonsensical for a partial parser to expand its input. Thus:

$$D_c(p) = \lambda w.p(cw) - (\lfloor p \rfloor(\epsilon) \times \{cw\}).$$

To arrive at a framework for parsing, we can solve this equation for the partial parser p in terms of the derivative:

$$\begin{aligned} & D_c(p) = \lambda w.p(cw) - (\lfloor p \rfloor(\epsilon) \times \{cw\}) \\ \text{iff } & D_c(p)(w) = p(cw) - (\lfloor p \rfloor(\epsilon) \times \{cw\}) \\ \text{iff } & p(cw) = D_c(p)(w) \cup (\lfloor p \rfloor(\epsilon) \times \{cw\}). \end{aligned}$$

Fortunately, we'll never have to deal with the "left-over" null parses in practice. With a full parser, these null parses are discarded:

$$\lfloor p \rfloor(cw) = \lfloor D_c(p) \rfloor(w).$$

Given their similarity, it should not surprise that the derivative of a partial parser resembles the derivative of a language:

- The derivative of the empty parser is empty:

$$D_c(\emptyset) = \emptyset.$$

- The derivative of the null parser is also empty:

$$D_c(\epsilon) = \emptyset.$$

- The derivative of the nullability combinator must be empty, since it at most parses the empty string:

$$D_c(\delta(L)) = \emptyset.$$

- The derivative of a single-character parser is either the null reduction parser or the empty parser:

$$D_c(c') = \begin{cases} \epsilon \downarrow \{c\} & c = c' \\ \emptyset & \text{otherwise.} \end{cases}$$

  This rule is important: it allows the derived parser to retain fragments of the input string within itself. Over time, as successive derivatives are taken, the parser is steadily transforming itself into a parse forest with nodes like this.

- The derivative of the union is the union of the derivative:

$$D_c(p \cup q) = D_c(p) \cup D_c(q).$$

- The derivative of a reduction is the reduction of the derivative:

$$D_c(p \rightarrow f) = D_c(p) \rightarrow f.$$

- The derivative of concatenation requires nullability, in case the first parser doesn't consume any input:

$$D_c(p \circ q) = (D_c(p) \circ q) \cup (\delta(p) \circ D_c(q)).$$

- The derivative of Kleene star peels off a copy of the parser:

$$D_c(p^\star) = (D_c(p) \circ p^\star) \rightarrow \lambda(h, t).h : t$$

The rules are so similar to the derivative for languages that we can modify the implementation of the derivative for languages to arrive at a derivative suitable for parsers:

```
(define/memoize (D c L)
  #:order ([L #:eq] [c #:equal])
  (match L
    [(empty)       (empty)]
    [(eps* T)      (empty)]
    [(δ _)         (empty)]
    [(char a)      (if (equal? a c)
                       (eps* (set c))
                       (empty))]

    [(alt L1 L2)   (alt (D c L1) (D c L2))]
    [(cat L1 L2)   (alt (cat (D c L1) L2))
                        (cat (δ L1) (D c L2))]
    [(rep L1)      (cat (D c L1) L)]
    [(red L f)     (red (D c L) f)]))
```

(Because pairing and list-building in Lisps both use `cons`, there is no reduction around the derivative of repetition.)

6.1 Parsing with derivatives

Parsing with derivatives is straightforward—until the last character has been consumed. To parse, compute successive derivatives of the top-level parser with respect to each character in a string. When the string is depleted, supply the null string to the final parser. In code, the `parse` function has the same structure as `matches?`:

```
(define (parse w p)
 (if (null? w)
     (parse-null p)
     (parse (cdr w) (D (car w) p))))
```

The question of interest is how to define `parse-null`, which produces a parse forest for the null parses of its input.

Yet again, an equational theory guides:

$$\begin{aligned} \lfloor \emptyset \rfloor(\epsilon) &= \{\} \\ \lfloor \epsilon \downarrow T \rfloor(\epsilon) &= T \\ \lfloor \delta(p) \rfloor &= \lfloor p \rfloor(\epsilon) \\ \lfloor p \cup q \rfloor(\epsilon) &= \lfloor p \rfloor(\epsilon) \cup \lfloor q \rfloor(\epsilon) \\ \lfloor p \circ q \rfloor(\epsilon) &= \lfloor p \rfloor(\epsilon) \times \lfloor q \rfloor(\epsilon) \\ \lfloor p \rightarrow f \rfloor(\epsilon) &= \{f(t_1), \ldots, f(t_n)\} \\ &\quad \text{where } \{t_1, \ldots, t_n\} = \lfloor p \rfloor(\epsilon) \\ \lfloor p^\star \rfloor(\epsilon) &= (\lfloor p \rfloor(\epsilon))^* \end{aligned}$$

A note on repetition The rule for repetition can mislead. If the interior parser can parse null, then there are an infinite number of parse trees to return. However, in terms of descriptiveness, one gains nothing by allowing the interior of a Kleene star operation to parse null—Kleene star already parses null by definition. So, in practice, we can replace that last rule by:

$$\lfloor p^\star \rfloor(\epsilon) = \begin{cases} \{\langle\rangle\} & p \text{ cannot parse null} \\ \textit{undefined} & \text{otherwise.} \end{cases}$$

What we have at this point are mutually recursive set constraint equations that mimic the structure of the nullability function for languages. Once again, the least fixed point is a sensible way of interpreting these equations. Thus, Kleene's fixed-point theorem, via `define/fix`, returns the set of full null parses:

```
(define/fix (parse-null p)
  #:bottom (set)
  (match l
    [(empty)       (set)]
    [(eps* T)      T]
    [(δ L)         (parse-null L)]
    [(char _)      (set)]

    [(alt p1 p2)  (set-union (parse-null p1)
                             (parse-null p2))]
    [(cat p1 p2)  (for*/set ([t1 (parse-null p1)]
                             [t2 (parse-null p2)])
                            (cons t1 t2))]
    [(red p1 f)   (for/set ([t (parse-null p1)])
                            (f t))]
    [(rep _)      (set '())]))
```

It assumes that the null parse of each node is initially empty.

7. Performance and complexity

The implementation is brief. The code is pure. The theory is elegant. So, how does this perform in practice? In brief, it is awful.

We constructed a parser for Python 3.1. On one-line examples, it returns interactively. Yet, it takes just under three *minutes* to parse a (syntactically valid) 31-line input. The culprit? The size of the grammar within the parser can grow exponentially with the number of derivatives. (The rule for concatenation is to blame.) Specifically, the grammar can double in size under the derivative. The cost model for parsing with derivatives is:

$$\begin{aligned} & \text{number of derivatives} \\ \times\ & \text{cost of derivative} \\ +\ & \text{cost of fixed point at the end.} \end{aligned}$$

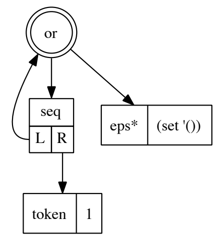
The cost of the derivative is proportional to the size of the current grammar. The cost of the fixed point is quadratic in the size of the grammar for unambiguous parses in the worst case. Thus, the worst-case complexity of parsing a grammar of size G over an input of length n is:

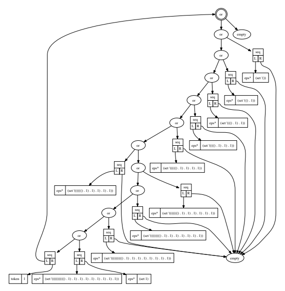
$$O(n2^nG + (2^nG)^2) = O(2^{2n}G^2).$$

Considering this complexity, it is remarkable that our example finished at all. That it finished in three minutes is astonishing.

7.1 Example: Growth in the grammar

A glance at run-time behavior on the left-recursive list grammar exposes the nature of the problem. The image on the left represents the grammar at the start; the image on the right represents the grammar after ten derivatives:





If one were to zoom in on image on the right, the node on the bottom right is `(empty)`. All of the inbound edges are from concatenation nodes—all of these nodes can be discarded.

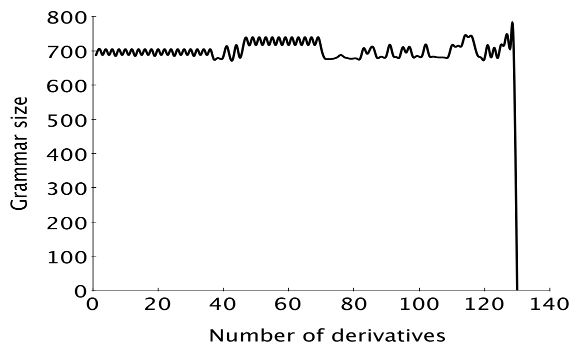
8. Compaction

Another equational theory shows how to eliminate unnecessary structure. The empty parser is an annihilator under concatenation and the identity under union; a null parser is the identity under concatenation.

It is possible to aggressively perform reductions as pieces of parse trees emerge. Our implementation utilizes the following simplifications; we use $(\Rightarrow)$ in lieu of $(=)$ to emphasize direction:

$$\begin{aligned} \emptyset \circ p = p \circ \emptyset &\Rightarrow \emptyset \\ \emptyset \cup p = p \cup \emptyset &\Rightarrow p \\ (\epsilon \downarrow \{t_1\}) \circ p &\Rightarrow p \rightarrow \lambda t_2.(t_1, t_2) \\ p \circ (\epsilon \downarrow \{t_2\}) &\Rightarrow p \rightarrow \lambda t_1.(t_1, t_2) \\ (\epsilon \downarrow \{t_1, \ldots, t_n\}) \rightarrow f &\Rightarrow \epsilon \downarrow \{f(t_1), \ldots, f(t_n)\} \\ ((\epsilon \downarrow \{t_1\}) \circ p) \rightarrow f &\Rightarrow p \rightarrow \lambda t_2.f(t_1, t_2) \\ (p \rightarrow f) \rightarrow g &\Rightarrow p \rightarrow (g \circ f) \\ \emptyset^\star &\Rightarrow \epsilon \downarrow \{\langle\rangle\}\,. \end{aligned}$$

We can implement these simplification rules in a memoized, recursive simplification function. When simplification is deeply recursive and memoized, we term it *compaction*. If the algorithm compacts after every derivative, then the time to parse the 31-line Python file drops from three *minutes* to two *seconds*. A graph of the size of the residual Python grammar with respect to each derivative hints as to why:



The size of the grammar (and the cost of each derivative) stays constant.

Warning With mere top-level simplification in lieu of memoization and deep recursive simplification, the grammar still grows with each derivative, and the cost of parsing the 31-line example explodes from two seconds to one minute.

8.1 Complexity

The worst-case complexity is unchanged: it is still exponential. However, we can hypothesize about its average performance given the observation that the grammar tends to stay roughly constant in size (until collapsing into a parse forest at the very end). The cost of each derivative remains proportional to the size of the original grammar. The cost of the fixed point at the end is negligible because the grammar has collapsed under compaction. Thus, we conjecture with reason that the cost of parsing with derivatives is $O(nG)$ in practice (for unambiguous grammars), where n is the size of the string, and G is the size of the grammar. Even for the ambiguous expression grammar, *recognition* appears to be $O(nG)$ (while producing all parse trees is exponential).

9. Related work

There has been a revival of interest in Brzozowski's derivative, itself a specialization of the well-known left quotient operation on languages. Owens, Reppy and Turon re-examined the derivative in light of lexer construction [13], and Danielsson [5] used it to prove the totality of parser combinators.

The literature on parsing is vast; there are dozens of methods for parsing, including but not limited to abstract interpretation [3, 4], operator-precedence parsing [9, 14], simple precedence parsing [7], parser combinators [15, 16], LALR parsing [6], LR(k) parsing [12], GLR parsing [17], CYK parsing [11, 20, 2], Earley parsing [8], LL(k) parsing, and recursive descent parsing [19]. packrat/PEG parsing [10, 18]. Derivative-based parsing shares full coverage of all context-free grammars with GLR, CYK and Earley.

Derivative-based parsing is not easy to classify as a top-down or bottom-up method. In personal correspondence, Stuart Kurtz pointed out that when the grammar is in Greibach Normal Form (GNF), the algorithm acquires a "parallel" top-down flavor. For grammars outside GNF, while watching the algorithm evolve under compaction, one sees what appears to be a pushdown stack emerge inside the grammar. (Pushes and pops appear as the jagged edges in the graph to the left.)

The most directly related work is Danielsson's work on total parser combinators [5]. His work computes residual parsers similar to our own, but does not detail a simplification operation. According to our correspondence with Danielsson, simplification does exist in the implementation. Yet, because it is unable to memoize the simplification operation (turning it into compaction), the implementation exhibits exponential complexity even in practice.

Acknowledgements We are grateful for the thousands of comments on reddit, hackernews, Lambda the Ultimate and elsewhere that thoroughly dissected and improved an earlier draft of this work. We also thank the ICFP 2011 and the ESOP 2010 reviewers for their thoughtful and detailed feedback. This research is partly supported by the National Science Foundation under Grant No. 1035658, by the DARPA CRASH project "GnoSys: Raising the level of discourse in systems program," and by the National Nuclear Security Administration under the "Accelerating Development of Retorfitable CO2 Capture Technologies through Predictivity" project through DOE Cooperative Agreement DE-NA0000740.

10. Conclusion

Our goal was a means to abbreviate the understanding and implementation of parsing. Brzozowski's derivative met the challenge: its theory is equational, its implementation is functional and, with an orthogonal optimization, its performance is not unreasonable.

References

[1] Brzozowski, J. A. Derivatives of regular expressions. *Journal of the ACM 11*, 4 (Oct. 1964), 481–494.

[2] Cocke, J., and Schwartz, J. T. Programming languages and their compilers: Preliminary notes. Tech. rep., Courant Institute of Mathematical Sciences, New York University, New York, NY, 1970.

[3] Cousot, P., and Cousot, R. Parsing as abstract interpretation of grammar semantics. *Theoretical Computer Science 290* (2003), 531–544.

[4] Cousot, P., and Cousot, R. Grammar analysis and parsing by abstract interpretation, invited chapter. In *Program Analysis and Compilation, Theory and Practice: Essays dedicated to Reinhard Wilhelm*, T. Reps, M. Sagiv, and J. Bauer, Eds., LNCS 4444. Springer discretionary--Verlag, Dec. 2006, pp. 178–203.

[5] Danielsson, N. A. Total parser combinators. *SIGPLAN Not. 45*, 9 (Sept. 2010), 285–296.

[6] DeRemer, F. L. Practical translators for LR(k) languages. Tech. rep., Cambridge, MA, USA, 1969.

[7] Dijkstra, E. W. *Selected Writings on Computing: A Personal Perspective*. Springer, Oct. 1982.

[8] Earley, J. An efficient context-free parsing algorithm. *Communications of the ACM 13*, 2 (Feb. 1970), 94–102.

[9] Floyd, R. W. Syntactic analysis and operator precedence. *Journal of the ACM 10*, 3 (July 1963), 316–333.

[10] Ford, B. Packrat parsing: Simple, powerful, lazy, linear time. In *Proceedings of the 2002 International Conference on Functional Programming* (Oct. 2002).

[11] Kasami, T. An efficient recognition and syntax-analysis algorithm for context-free languages. Tech. rep., Air Force Cambridge Research Lab, Bedford, MA, 1965.

[12] Knuth, D. On the translation of languages from left to right. *Information and Control 8* (1965), 607–639.

[13] Owens, S., Reppy, J., and Turon, A. Regular-expression derivatives re-examined. *Journal of Functional Programming 19*, 02 (2009), 173–190.

[14] Pratt, V. R. Top down operator precedence. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (New York, NY, USA, 1973), POPL '73, ACM, pp. 41–51.

[15] Swierstra, D. S., Pablo, and Sariava, J. Designing and implementing combinator languages. In *Advanced Functional Programming* (1998), pp. 150–206.

[16] Swierstra, S. Combinator parsing: A short tutorial. In *Language Engineering and Rigorous Software Development*, A. Bove, L. Barbosa, A. Pardo, and J. Pinto, Eds., vol. 5520 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, Berlin, Heidelberg, 2009, ch. 6, pp. 252–300.

[17] Tomita, M. LR parsers for natural languages. In *ACL-22: Proceedings of the 10th International Conference on Computational Linguistics and 22nd annual meeting on Association for Computational Linguistics* (Morristown, NJ, USA, 1984), Association for Computational Linguistics, pp. 354–357.

[18] Warth, A., Douglass, J. R., and Millstein, T. Packrat parsers can support left recursion. In *PEPM '08: Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation* (New York, NY, USA, 2008), PEPM '08, ACM, pp. 103–110.

[19] Wirth, N. *Compiler Construction (International Computer Science Series)*, pap/dsk ed. Addison-Wesley Pub (Sd).

[20] Younger, D. H. Recognition and parsing of context-free languages in time n3. *Information and Control 10*, 2 (1967), 189–208.