1. 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 (as well as parser combinators).

The supporting actors in this story are three concepts familiar to functional programmers—laziness, memoization and fixed points. These techniques allow Brzozowski's original equations to be transliterated into purely functional code; 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 malignant, contributing nothing to the meaning of the grammar, and can be removed. To eliminate it, we once again exploit laziness, memoization and fixed-points to transliterate an equational theory which prunes such debris. We introduce two optimization under which we argue parsing times become reasonable in practice.

We equip the functional programmer with executable equational theories for parsing with derivatives: a framework which encompasses both an abbreviated understanding and implementation for parsing arbitrary context-free languages.

2. Introduction

It is easy to lose sight of the essence of parsing. Implementation details like forbidden grammars, shift-reduce conflicts and opaque action tables distract from the core task at hand: the conversion of input strings into parse trees. Brzozowski's derivative ends the distraction, presenting a clean, straightforward and equational connection between an input string, a grammar and the resulting parse trees.

Brzozowski formulated the derivative in the context of regular expressions to create a straightforward, equational theory for recognizing regular languages (**?**). Yet, the theory of the derivative applies–with modest generalization–to context-free languages as well. When gently tempered with laziness, memoization and fixed points, an implementation of Brzozowski's derivative extends directly to a purely functional technique for generating parse forests from arbitrary context-free grammars. Despite its seemingly impossible simplicity, the derivative transparently handles ambiguity, left-recursion, right-recursion, ill-founded recursion and any combination thereof. No special handling is required.

2.1. 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. Recovered termination empowers the derivative to recognize arbitrary context-free languages.
- Since the modifications are so small, we construct an argument for the soundness and completeness of the derivative as applied to context-free languages.
- We supplement our soundness and completeness argument with a sequence of increasingly challenging examples in an attempt to build intuition for the effectivess of the derivative of context-free grammars.
- We generalize the derivative to parsers and parser combinators through an equational theory for generating parse forests.
- We find naive parsing with derivatives gives poor performance in both theory and practice. An analysis of the structure of grammers as they transform through the derivative highlights room for improvement.
- To correct the poor performance of the the naive parsing theory we develop an optimization—compaction—which collapses grammars by trimming excess mass. Like the derivative, compaction also comes from a clean equational theory and its implementation exploits laziness, memoization and fixed points.
- Further analysis shows that while practical parsing times are recovered through compaction, there is still grammar structure to be

exploited for performance. Through yet another equational theory we develop a second optimization—focusing—which when coupled with compaction recovers linear parse times for a wide array of sensible grammars.

In this article we provide code in Racket. Our implementation will adapt readily to other languages. All code and test cases within or referenced from this article are available from:

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

3. Preliminary: Formal languages

A language L is a set of strings. A string w is a sequence of characters from an alphabet A.

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. That is, depending on the context $\epsilon = \{\texttt{""}\}$ or $\epsilon = \texttt{""}$.

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:

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

3.1. Operations on languages

Because languages are sets, set operations like union apply:

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

Language concatenation ($\circ$) lifts string concatenation to 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:

$$L^0 = \epsilon$$
$$L^n = \{w_1 w_2 \dots w_n : w_i \in L\}.$$

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

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

3.2. Regular languages and context-free languages

A *regular expression* consists of atomic sets, union, concatenation and repitition[?]. Note that a *regular language* is any language for which there exists a *regular expression* to describe it.

Allowing mutually recursive definitions in a regular expression yields exactly the set of *context-free grammars* under a least-fixed-point interpretation[?]. For instance, given the language L:

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

The least-fixed-point interpretation of L is the set of all finite-length sequences of the character x, including the null string. (The greatest-fixed-point interpretation of L would include an infinite string of `x`'s.) Note that a *context-free language* is any language for which there exists a *context-free grammar* to describe it.

Kleene star becomes syntactic sugar for recursive union and concatenation in context-free grammars:

$$G = L^\star \Rightarrow G = L \circ G \cup \epsilon$$

3.3. Encoding languages

To represent the structure of regular expressions in code we define a `struct` for each kind of language:

```
(define-struct ∅     {})
(define-struct ϵ     {})
(define-struct token {value})

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

EXAMPLE 3.1. *In code, the language:*

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

becomes:

```
(define L (∪ (∘ L (∪ (token 'a) (token 'b)))
             (ϵ)))
```

4. Brzozowski's derivative

Brzozowski defined the derivative of regular expressions in his work on the recognition of regular languages (**?**). The derivative of a language L with respect to a character c, written $D_c(L)$, is a new language which has been "filtered" and "chopped":

1. First, retain only the strings in L which start with c.
2. Second, remove c from the beginning of every string.

Formally:

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

EXAMPLE 4.1.

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

4.1. Recognition with the derivative

The simplicity of the derivative's definition masks its power. It is straightforward to determine the membership of a string within a language through successive derivatives using the following property:

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

To determine membership of string s in language L, just derive L with respect to each character of s in sequence. The resulting language contains the null string if and only if s is contained in L.

4.2. A recursive definition of the derivative

Brzozowski noted that regular expressions, and therefore regular languages, are closed under the derivative. A constructive implementation of the derivative for regular languages is described recursively over the structure of regular expressions:

- 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 to derive:

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

- The derivative of concatenation must first consider the possibility that the first language could be null:

$$\begin{aligned} D_c(L_1 \circ L_2) &= D_c(L_1) \circ L_2 \text{ if } \epsilon \not\in L_1 \\ D_c(L_1 \circ L_2) &= (D_c(L_1) \circ L_2) \cup D_c(L_2) \text{ if } \epsilon \in L_1 \end{aligned}$$

  Concatenation without a conditional can be expressed through the nullability function: δ. The nullability function returns the null language if its input language contains the null string, and the empty set otherwise:

$$\begin{aligned} \delta(L) &= \emptyset \text{ if } \epsilon \not\in L \\ \delta(L) &= \epsilon \text{ if } \epsilon \in L. \end{aligned}$$

  Concatentation is equivalently defined:

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

4.3. 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 also straightforward:

```
(define (δ? L)
  (match L
    [(∅)          #f]
    [(ϵ)          #t]
    [(token _)    #f]

    [(∪ L1 L2)    (or  (δ? L1) (δ? L2))]
    [(◦ L1 L2)    (and (δ? L1) (δ? L2))]
    [(⋆ _)        #t]))
```

EXAMPLE 4.2. *A couple examples illustrate the derivative on regular languages:*

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

4.4. An implementation of the derivative

Because 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
    [(∅)                       (∅)]
    [(ϵ)                       (∅)]
    [(token a)                 (if (equal? c a) (ϵ) (∅))]

    [(∪ L1 L2)                 (∪ (D c L1) (D c L2))]
    [(∘ (and (? δ?) L1) L2)  (∪ (D c L2) (∘ (D c L1) L2))]
    [(∘ L1 L2)                 (∘ (D c L1) L2)]
    [(⋆ L1)                    (∘ (D c L1) (⋆ L1))]))
```

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

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

4.5. Aside: An alternative implementation

The nullability function δ admits an alternative and arguably more elegant theory and implementation. Rather than defining nullability as an operator on languages we define nullability as a language combinator like $\cup$ or $\circ$.

```
(define-struct δ {lang})
```

To fit δ into our existing framework for recognition we need only define the derivative of nullability; the interpretation follows as before. The derivative of nullability is always empty:

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

Conveniently, nullability of nullability is equal to nullability:

$$\delta(\delta(L)) = \delta(L),$$

Promoting nullability to a connective allows for a simpler definition of the derivative; concatenation no longer requires a branch in logic:

```
(define (D c L)
  (match L
    [(∅)          (∅)]
    [(ϵ)          (∅)]
    [(δ L)        (∅)]
    [(token a)     (if (equal? c a) (ϵ) (∅))]

    [(∪ L1 L2)    (∪ (D c L1) (D c L2))]
    [(◦ L1 L2)    (∪ (◦ (D c L1) L2)
                     (◦ (δ L1) (D c L2)))]
    [(⋆ L1)       (◦ (D c L1) (⋆ L1))]))
```

We will use this formulation of the derivative and nullability for the remainder of the paper, as its contrasting simplicity will be of even greater effect as we develop theories for parsing.

5. Derivatives of context-free languages

Before we discuss the derivative of context-free languages it must first be established whether or not such a thing makes sense. The derivative is a set-theoretic operation on languages and is *not* defined in terms of any particular language structure such as regularity. Brzozowski showed—constructively—that regular expressions, and thus regular languages, are closed under the derivative. It remains for us to show that context-free grammars are closed under the derivative, after which closure for context-free languages would directly follow.

Fortunately the argument for closure of context-free grammars under the derivative is trivial because context-free grammars are merely recursive regular expressions. The equations are the same.

Knowing that the derivative of context-free grammars follows the same elegant theory as that of regular expressions we turn our attention to implementation. Because a context-free grammar is a recursive regular expressions, 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 quickly leads to non-termination.

The second comes from the fact that our regular implementation only *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 a later section (7).

EXAMPLE 5.1. *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 (yet). The code from the previous section recurs forever as it attempts to compute the derivative of the language L.

5.1. 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 $\cup$, $\circ$ and $\star$ lazy. [1] With lazy fields the computation of any (potentially self-referential) derivatives in those fields gets suspended until the values in those fields are required.

5.2. Step 2: Memoization

With laziness we can repeatedly compute the derivative until the end of the recognition algorithm when the resulting languages is tested for nullability. Nullability currently walks (eagerly) the structure of the entire language and fails to terminate on a derived language such as the one above. We need the derivative to return a finite (if lazily explored) graph. Memoizing the derivative will “tie the knot” when it re-encounters a language it has already seen:

[1] Lisp implementations that do not support lazy fields can provide them transparently with macros and mechanisms like `delay` and `force`. Syntactic support for laziness is not essential to this implementation; it is merely a convenience.

```
(define/memoize (D c L)
  #:order ([L #:eq] [c #:equal])
  (match L
    [(∅)                        (∅)]
    [(ϵ)                        (∅)]
    [(token a)                  (if (equal? c a) (ϵ) (∅))]

    [(δ L)                      (∅)]
    [(∪ L1 L2)                  (∪ (D c L1) (D c L2))]
    [(○ (and (? δ?) L1) L2)  (∪ (D c L2) (○ (D c L1) L2))]
    [(○ L1 L2)                  (○ (D c L1) L2)]
    [(⋆ L1)                     (○ (D c L1) (⋆ L1))]))
```

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.

5.3. Step 3: Fixed points

The computation of nullability [2] 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 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 expose only the essence of nullability we can hide the computation of a least fixed point behind a purely functional abstraction: `define/fix`. Internally, 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:

[2] We are still committed to presenting δ as a language-level connective rather than a property of a language, as mentioned in a previous section. Here we consider the boolean variant, which will be needed to interpret the δ connective.

```
(define/fix (δ? L)
  #:bottom #f
  (match L
    [(∅)          #f]
    [(ϵ)          #t]
    [(token _)    #f]

    [(δ L)        (δ? L)]
    [(∪ L1 L2)    (or  (δ? L1) (δ? L2))]
    [(∘ L1 L2)    (and (δ? L1) (δ? L2))]
    [(⋆ _)        #t]))
```

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 of `define/fix`:

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

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

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

which is easily computed with straightforward iteration:

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

5.4. 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 have constructed a system for recognizing any context-free language in less than 30 lines of code.

6. But, does it work?

One might be skeptical that three simple modifications in the implementation of the derivative are all that it takes to move from regular languages to context-free languages. We address this skepticism with an argument for soundness and completeness. To buttress intuition, we then run the implementation on a battery of grammars designed to exhibit properties that are problematic for other parsing regimes.

6.1. An argument for soundness and completeness

Skepticism about context-free language recognition via derivatives comes in form of two broad concerns:

1. *Soundness*: If matching terminates, is the result correct?

2. *Completeness*: Does the matching process always terminate?

6.1.1. *Soundness*

The first concern—soundness—can be further subdivided: (a) Is Brzozowski's theory of derivatives correct? And, (b) is our rendering of Brzozowski's theory faithful?

With regard to (a), the equations for Brzozowski's derivative, unmodified, work directly for context-free languages: the equations for across union and concatenation hold whether the underlying language is regular or context-free.

Regarding (b), when we render equations directly as purely functional code, the specification *is* the implementation. Written as code, these equations work "out of the box" for regular languages because regular languages lack recursive structure. They fail to terminate on context-free languages that have recursive structure.

We introduced three modifications to achieve termination: laziness, memoization and a fixed point computation. Still in the context of soundness, we can answer concern (b) by examining each of these modifications in turn. For each, the key question is: does it respect equational reasoning—is it pure?

By-need laziness obeys equational reasoning: pure lazy programs are still pure. Nor does memoization destroy equational reasoning: memoizing a pure function yields another pure function. Computing the nullability of a grammar through fixed points is equational. The fixed-point computations are pure and correct under the assumption that the underlying fixed-point solver is correct.

6.1.2. *Completeness*

The remaining concern is (2): does the matching process always terminate? This question reduces to a single concern:

Is the derivative of a finite grammar always finite?

We can show that the derivative of a context-free grammar will double its size in the worst case.

Let $N_1, \dots, N_n$ be the nonterminals of the grammar and $R_1, \dots, R_n$ be the corresponding right-hand sides:

$$\begin{aligned} N_1 &::= R_1 \\ &\vdots \\ N_n &::= R_n. \end{aligned}$$

To take the derivative with respect to c, introduce n new non-terminals: $D_cN_1, \dots, D_cN_n$. For each of these we calculate a new rule:

$$\begin{aligned} D_cN_1 &::= D_c(R_1) \\ &\vdots \\ D_cN_n &::= D_c(R_n). \end{aligned}$$

For each regular right-hand side R_i, the closure of the derivative over regular languages means that $D_c(R_i)$ expands into regular operations involving the old non-terminals and the new non-terminals. Hence, the derivative is closed over context-free grammars.

Thanks to laziness the derivative will not always systematically compute the derivative of every rule, but if even if it did, the size of the grammar would at most double. By extension, it must terminate.

Hence recognition with derivatives is both sound and complete for all context-free grammars.

6.2. Challenging examples

In our experience the most compelling (if not the most rigorous) way of arguing for the correctness of recognition by derivatives comes from running the implementation from above on challenging grammars with both positive and negative inputs.

A Racket script to run all of these grammars and test cases is available:

`http://ucombinator.org/projects/parsing/derp/demo.rkt`

EXAMPLE 6.1. *On the following left-recursive grammar:*

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

the implementation correctly recognizes strings like `x` *and* `xxx`*, and it correctly rejects strings like* `xy` *and* `yx`*.*

EXAMPLE 6.2. *On the following right-recursive grammar:*

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

the implementation correctly recognizes strings like `x` *and* `xxx`*, and it correctly rejects strings like* `xy` *and* `yx`*.*

EXAMPLE 6.3. *If left-recursion is "hidden" through indirection, as in the following grammar:*

$$\begin{aligned} A &= B \circ \{\mathtt{x}\} \\ &\cup \epsilon \\ B &= A, \end{aligned}$$

the implementation still correctly recognizes strings like `x` *and* `xxx`*, and it correctly rejects strings like* `xy` *and* `yx`*. If we "hide" right-recursion is a similar fashion, it still works.*

EXAMPLE 6.4. *On the following infinitely-recursive grammar,*

$$L = L,$$

the implementation correctly rejects all strings. This particular grammar is an important corner case. In code, it might be defined as:

```
(define L L)
```

but this results in an undefined language. To get it work correctly, `L` *must pass through a lazy constructor. When we introduce parsers, we achieve this with an empty reduction. In this case, it's sufficient to union it with the empty language, e.g.:*

```
(define L (∪ L ∅))
```

EXAMPLE 6.5. *On the following grammar, we have infinite recursion buried inside an acceptable language:*

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

yet the implementation correctly accepts strings like `xxx` *and* `xxxxx` *and rejects those like* `xyxy` *and* `yxxx`*.*

EXAMPLE 6.6. *Even if we "hide" the infinite recursion with mutual indirection, as in the following grammar:*

$$\begin{aligned} A &= B \\ B &= A, \end{aligned}$$

the implementation correctly rejects all strings.

EXAMPLE 6.7. *Even if we "sandwich" the infinite recursion inside what appears to be a valid language, as in the following grammar:*

$$L = \{\mathtt{x}\} \circ L \circ \{\mathtt{x}\},$$

the implementation correctly rejects all strings.

EXAMPLE 6.8. *On ambiguous left- and right-recursive grammars like the expression grammar:*

$$\begin{aligned} E = {} & E \circ \{\mathtt{+}\} \circ E \\ & \cup E \circ \{*\} \circ E \\ & \cup \{\mathtt{(}\} \circ E \circ \{\mathtt{)}\} \\ & \cup \mathbb{N}, \end{aligned}$$

the implementation correctly accepts expressions like `3 + (4 * 4)` *and rejects ones like* `3 + (4 * 4) +`*.*

EXAMPLE 6.9. *Finally, if we hide infinite recursion in the ambiguous expression grammar, as in:*

$$\begin{aligned} E = {} & E \circ \{\mathtt{+}\} \circ E \\ & \cup E \circ \{*\} \circ E \\ & \cup \{\mathtt{(}\} \circ E \circ \{\mathtt{)}\} \\ & \cup E \\ & \cup \mathbb{N}, \end{aligned}$$

the implementation still correctly accepts expressions like `3 + (4 * 4)` *and rejects ones like* `3 + (4 * 4) +`*.*

7. 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. 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 (**?**; **?**).) 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 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 promote any partial parser $p \in \mathbb{P}(A, T)$ to a full parser:

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

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

7.1. 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 a parser which consumes nothing:

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

- The empty set becomes the parser which rejects everything:

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

7.2. 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. The remainder of the input is passed to the second parser which produces another parse tree. The result is the left-over input paired with both 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-lazy-struct → {lang f})
```

The field `lang` should be lazy for context-free parsing.

7.3. The nullability combinator

We utilize the nullability combinator δ approach to defining the derivative over parsers, since this simplifies the definition. It becomes a reject-everything parser if the language cannot parse empty, and the null parser if it can:

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

7.4. The null reduction parser

The null reduction parser $\epsilon \downarrow S$ will be necessary later for defining the derivative operation on parsers. It can only parse the null string and 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-lazy-struct ϵ↓ {trees})
```

7.5. 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 $p^\star \in \mathbb{P}(A, T^*)$:

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

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

8. 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:

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

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 then 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 steadily transforms itself into a parse forest with null reduction parsers.

- 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:

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

– 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:

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

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
    [(∅)          (∅)]
    [(ϵ ↓ S)      (∅)]
    [(token a)    (if (equal? a c)
                      (ϵ ↓ (set c))
                      (∅))]

    [(δ _)        (∅)]
    [(∪ L1 L2)    (∪ (D c L1) (D c L2))]
    [(○ L1 L2)    (∪ (cat (D c L1) L2))
                      (○ (δ L1) (D c L2))]
    [(⋆ L1)       (○ (D c L1) L)]
    [(→ L f)      (→ (D c L) f)]))
```

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

8.1. 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:

$$\begin{aligned}
\lfloor \emptyset \rfloor(\epsilon) &= \{\} \\
\lfloor \epsilon \downarrow S \rfloor(\epsilon) &= S \\
\lfloor \delta(p) \rfloor &= \lfloor p \rfloor(\epsilon) \\
\lfloor p \cup q \rfloor(\epsilon) &= \lfloor p \rfloor(\epsilon) \cup \lfloor q \rfloor(\epsilon) \\
\lfloor p \circ q \rfloor(\epsilon) &= \lfloor p \rfloor(\epsilon) \times \lfloor q \rfloor(\epsilon) \\
\lfloor p \to f \rfloor(\epsilon) &= \{f(t_1), \ldots, f(t_n) : t_i \in \lfloor p \rfloor(\epsilon)\} \\
\lfloor p^\star \rfloor(\epsilon) &= (\lfloor p \rfloor(\epsilon))^*
\end{aligned}$$

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 p
    [(∅)          (set)]
    [(ϵ ↓ T)       T]
    [(token _)    (set)]

    [(δ L)        (parse-null L)]
    [(∪ p1 p2)    (set-union (parse-null p1) (parse-null p2))]
    [(∘ p1 p2)    (for*/set ([t1 (parse-null p1)]
                             [t2 (parse-null p2)])
                    (cons t1 t2))]
    [(⋆ _)        (set '())]
    [(→ p1 f)     (for/set ([t (parse-null p1)])
                    (f t))]))
```

9. Performance and complexity

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

The culprit? The size of the grammar can and will grow exponentially with the number of derivatives. The concatenation rule is specifically to blame for doubling the size of the grammar. The general 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 grammar, which currently doubles after each derivative. The cost of the fixed point at the end is cubic in the size of the grammar. This cost, however, is quickly dwarfed by the integration of the cost of the derivative over the length of the input, and can be ignored. Thus, the cost of parsing a grammar G with an input of length n under the current implementation is:

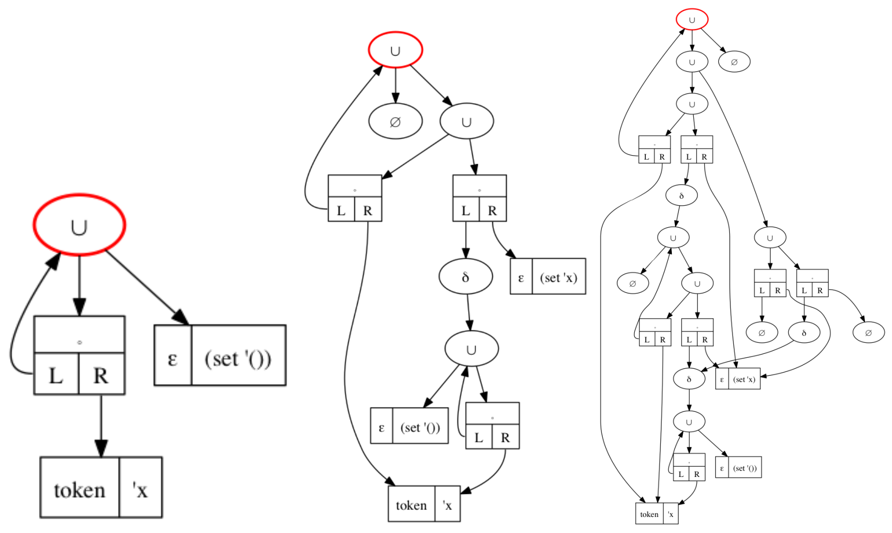
$$|G| + 2 \cdot |G| + 2 \cdot 2 \cdot |G| + ... = |G| \sum_{i=0}^{n} 2^i \in O(|G| 2^n)$$

9.1. Example: Growth in the grammar

Examining the run-time growth of any grammar would serve to expose the nature of our complexity problem. We have chosen a left-recursive sequence grammar for illustration:

```
(define xs-left
  (∪ (∘ xs-left (token ’x))
      (ϵ (set ’()))))
```

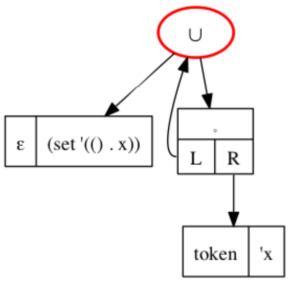
Observe the structure of xs-left after zero, one and two derivatives:

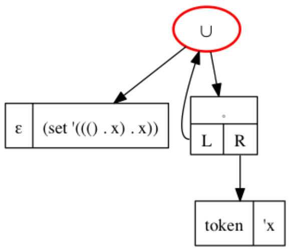


There are two things to note. First, our suspicion of exponential growth for even trivial grammars is confirmed empirically.

Second, we claim that that much smaller, yet equivalent, grammars exist for the first and second derivative. Consider the fact that context-free *languages*, not just grammars, are closed under the derivative. xs-left represents the set of all finite-length sequences of the token ’x, and its derivative, conceptually, is its self. In the context of parsing, the derivative of xs-left is not exactly xs-left; the derivative needs to remember it has seen a token through an epsilon reduction node. The *intuition*, however, that the derivative should be similar in structure to xs-left holds strong.

For example, the following grammars are equivalent to the previously presented first and second derivative of xs-left:





The change is not in the language to be recognized, but in the result to be returned through epsilon nodes. In the next section we develop a theory of optimizations to find these equivalent and more compact grammars.

10. Compaction

We introduce an equational theory of equivalence over grammars with the aim to reduce the explosion of grammar size during parsing. We use $(\Rightarrow)$ in lieu of $(=)$ to emphasize direction.

First, we identify grammars which are equivalent to $\emptyset$:

$$\begin{aligned}
\emptyset \cup \emptyset &\Rightarrow \emptyset \\
\emptyset \circ p &\Rightarrow \emptyset \\
p \circ \emptyset &\Rightarrow \emptyset \\
\emptyset \to f &\Rightarrow \emptyset \\
\delta(L) &\Rightarrow \emptyset \text{ if } L \text{ not nullable}
\end{aligned}$$

Next, grammars which are equivalent to $\epsilon \downarrow S$ for some S:

$$\begin{aligned}
(\epsilon \downarrow S_1) \cup (\epsilon \downarrow S_2) &\Rightarrow \epsilon \downarrow (S_1 \cup S_2) \\
(\epsilon \downarrow S_1) \circ (\epsilon \downarrow S_2) &\Rightarrow \epsilon \downarrow (S_1 \circ S_2) \\
(\epsilon \downarrow S) \to f &\Rightarrow \epsilon \downarrow (S \to f) \\
\delta(L) &\Rightarrow \lfloor L \rfloor(\epsilon) \text{ if } L \text{ nullable} \\
(\epsilon \downarrow S)^\star &\to \epsilon \downarrow (S^\star)
\end{aligned}$$

And lastly, grammar identities modulo reduction:

$$\begin{aligned}
&\emptyset \cup p = p \cup \emptyset \Rightarrow p \\
&(\epsilon \downarrow \{t_1\}) \circ p \Rightarrow p \to \lambda t_2.(t_1, t_2) \\
&p \circ (\epsilon \downarrow \{t_2\}) \Rightarrow p \to \lambda t_1.(t_1, t_2) \\
&(p \to f) \to g \Rightarrow p \to (g \circ f)
\end{aligned}$$

On the left-hand side of each of the above equations, $\emptyset$ and $(\epsilon \downarrow S)$ can be generalized to the equivalence classes of grammers under the same equations. This leads naturally to a fixed-point interpretation for recognition of these classes. To recognize equivalence classes $\emptyset$ and $(\epsilon \downarrow S)$ we define essentially-$\emptyset$? and essentially-ϵ?:

```
(define/fix (essentially-∅? L)
  #:bottom #t
  (match L
    [(∅)            #t]
    [(ε s)          (set-empty? s)]
    [(token _)      #f]

    [(δ p)          (not (nullable? p))]
    [(∪ L1 L2)      (and (essentially-∅? L1) (essentially-∅? L2))]
    [(∘ L1 L2)      (or  (essentially-∅? L1) (essentially-∅? L2))]
    [(⋆ L1)         #f]
    [(→ L1 _)       (essentially-∅?  L1)]))

(define/fix (essentially-ε? L)
  #:bottom #f
  (match L
    [(∅)            #f]
    [(ε _)          #t]
    [(token _)      #f]

    [(δ p)          (nullable? p)]
    [(∪ L1 L2)      (and (essentially-ε? L1) (essentially-ε? L2))]
    [(∘ L1 L2)      (and (essentially-ε? L1) (essentially-ε? L2))]
    [(⋆ L1)         (or  (essentially-ε? L1) (essentially-∅? L1))]
    [(→ L1 _)       (essentially-ε? L1)]))
```

The grammar identities, when augmented with the above equivalence classes and applied recursively, follow as the core of our simplification step K.

```
(define/memoize (K [L #:eq])
  (match L
    [(? essentially-∅?)         (∅)]
    [(? essentially-ε?)         (ε (parse-null L))]
    [(∪ (? essentially-∅?) L2)  (K L2)]
    [(∪ L1 (? essentially-∅?))  (K L1)]
    [(∪ L1 L2)                  (∪ (K L1) (K L2))]
    [(○ (singleton-ε? e) L2)    (→ (K L2) (λ (w2) (cons e w2)))]
    [(○ L1 (singleton-ε? e))    (→ (K L1) (λ (w1) (cons w1 e)))]
    [(○ L1 L2)                  (○ (K L1) (K L2))]
    [(⋆ L)                      (⋆ (K L))]
    [(→ (→ L f) g)              (→ (K L) (compose g f))]
    [(→ L f)                    (→ (K L) f)]
    [_                          L]))
```

Our optimization—which we term *compaction*—entails passing the result of each derivative through K before the next derivative.

10.1. Complexity

The worst-case complexity of parsing under compaction is surprisingly worse than naive parsing. We argue, however, that it is good in practice. At times we will refer to "sensible" grammars, which we take to be the class of unambiguous, LL(k) or LR(k) grammars. We will address generally unambiguous grammars briefly after our more targeted narrative on "sensible" grammars.

The cost-model of parsing with compaction is the same as before:

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

The worst-case cost of the derivative is now cubic in the size of the current grammar because it computes several fixed-point analyses to trigger simplification. As before, the grammar may double in size under the derivative (in the worst case), and the cost of the fixed point at the end is negligeable. The worst-case complexity of parsing under compaction is therefore:

$$|G|^3 + (2 \cdot |G|)^3 + (2 \cdot 2 \cdot |G|)^3 + ... = |G|^3 \sum_{i=0}^{n} (2^i)^3 \in O(|G|8^n)$$

For "sensible" grammars, however, we find the runtime of parsing with compaction a different story. On the previously presented xs-left example, the size of the grammar stays constant under compaction. In fact for every "sensible" gramar we have tested, compaction is able to keep the size of the grammar nearly constant. The only growth of the grammar appears when entering context-free portions of the grammar, and this growth follows a stack-like structure linear in the depth of context.

Furthermore, for "sensible" grammars the cost of the fixed-point analysis is constant in all of our experiments. Pathalogical grammars exist which stroke the cubic worst-case runtime of the fixpoint, but we have yet to see this bad behavior appear for "sensible" grammars.

We are confident that parsing with compaction is linear for "sensible" *regular* grammars. The claim is supported by observing that the cost of the fixpoint is constant and the size of the grammar doesn't grow:

$$(K + |G|) + (K + |G|) + (K + |G|) + ... = n(K + |G|) \in O(|G|n)$$

We are also confident that parsing with compaction is at worst quadratic for "sensible" *context-free* grammars. The claim is supported by observing that the cost of the fixpoint is constant and the size of the grammar grows at most by a constant factor t:

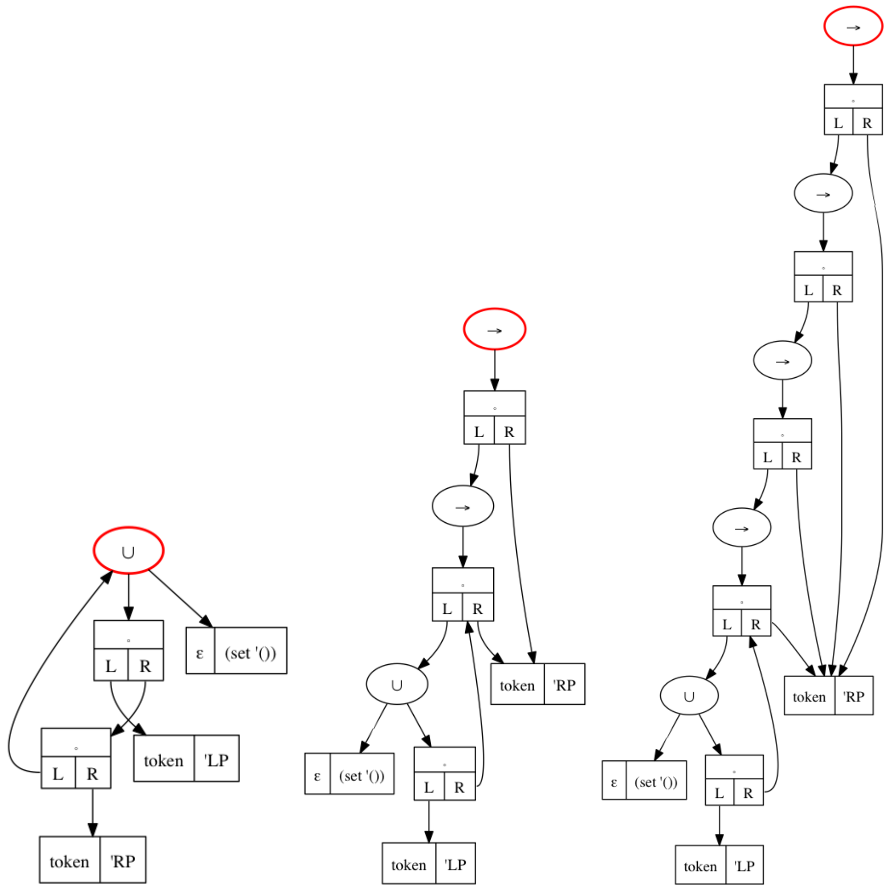
$$(K+|G|)+(K+|G|+t)+(K+|G|+2t)+... = n(K+|G|)+t\sum_{i=0}^{n} i \in O(|G|n^2)$$

11. Focusing

It is a shame that parsing with derivatives under compaction cannot be linear for "sensible" *context-free* grammars. Quadratic parse-times for every-day context-free languages are a show-stopper for the purported "practicality" of any parsing theory. We present a simple grammar—nested-parentheses—which illustrates the linear growth of a context-free grammar during parsing:

```
(define parens (∪ (∘ (token 'LP) (∘ parens (token 'RP)))
                  (ε (set '()))))
```

Observe the structure of parens after zero, two and four derivatives under compaction parsing:



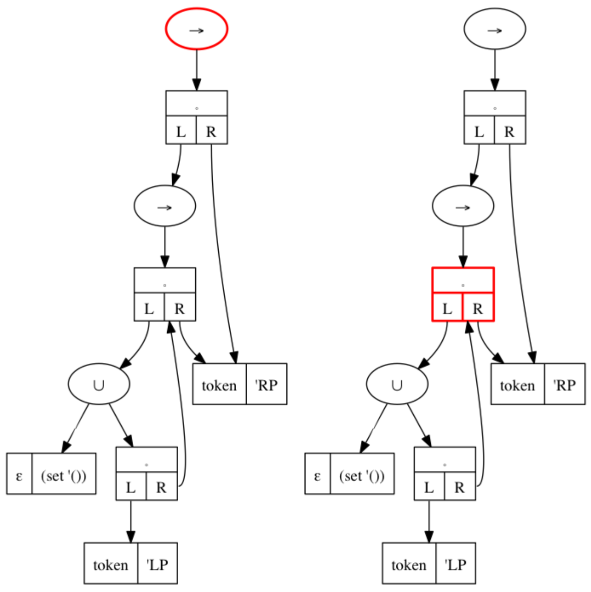
The grammar grows linearly as it retains a stack for the context-free portion of the grammar. In this section we manage this growth through yet another equational theory, bringing the cost of derivative parsing down to linear for “sensible” grammars in practice.

We introduce the following equations for the derivative, which follow trivially from definition:

$$\begin{aligned} D_c(L_1 \circ L_2) &= L_1 \circ D_c(L_2) && \text{if } L_1 = \epsilon \\ D_c(L_1 \circ L_2) &= D_c(L_1) \circ L_2 && \text{if } L_1 \text{ not nullable} \\ D_c(L \rightarrow f) &= D_c(L) \rightarrow f \end{aligned}$$

[We note that other equations exist, but these are the only three which aren’t subsumed by compaction.]

With the above equations, the “focus” of the derivative can be repositioned through concatenation and reduction connectives to a smaller sub-grammar, given that certain conditions hold. In the parens example, the focus of the derivative can be repositioned through the entire growth portion of the grammar:



The derivative equations justify the focusing; taking the derivative on the focused grammar yeilds the same result as taking the derivative on the whole.

11.1. Zippers

To represent a focused grammar we use a functional zipper datatype for the grammar. A top-level zipper parser is now represented as a parser and a context, where the top-level zipper parser *does not* occur recursively inside the parser. The context type is the *datatype derivative* of the parser type. Because we only introduced equations for concatentation and reduction, we only include their datatype derivatives:

```
(define-struct zipper parser context)

(define-struct □ ∘ • L2 k) ; ∘ with a missing left parser
(define-struct • ∘ □ L1 k) ; ∘ with a missing right parser
(define-struct □ → • L k)  ; → with a missing parser
```

The interpretation of a zipper—pzip—fills the hole of a context with a parser:

```
(define (pzip Z)
  (match Z
    [(zipper L 'top)           L]
    [(zipper L1 (□ ∘ ● L2 k))  (pzip (zipper (∘ L1 L2) k))]
    [(zipper L2 (● ∘ □ L1 k))  (pzip (zipper (∘ L1 L2) k))]
    [(zipper L (□ → ● f k))   (pzip (zipper (→ L f) k))]))
```

Finally we follow the derivative equations to write a procedure—drill—which constructs a zipper by focusing on the inner-most relevant derivative position:

```
(define (drill Z)
  (match Z
    [(zipper L k)
     (match L
       [(∘ (? essentially-ε? L1)
           (and (not (? essentially-ε?)) L2))
        ; ⇒
        (drill (zipper L2 (● ∘ □ L1 k)))]
       [(∘ (and L1 (not (? nullable?))) L2)
        ; ⇒
        (drill (zipper L1 (□ ∘ ● L2 k)))]
       [(→ (and (not (? essentially-ε?)) L) f)
        ; ⇒
        (drill (zipper L (□ → ● f k)))]
       [_
        ; ⇒
        Z])]))
```

[Note: "(and (not (? essentiall-ε?)) L)" is a Racket matching pattern for "if essentially-ε? returns false then match and bind to L".]

We remark that drill has the following property:

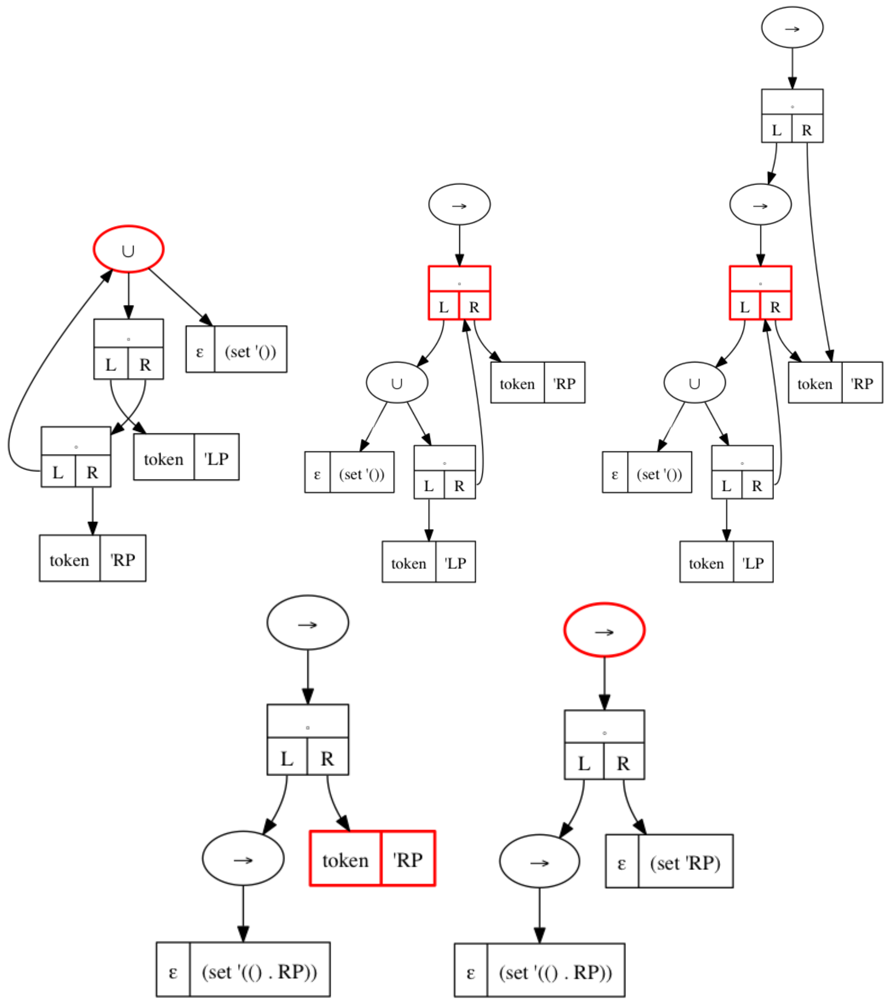
```
drill p = (zipper p' k)  ⟹
  D_c(p) = D_c(pzip p' k) = pzip D_c(p') k
```

We are not yet able to use drill in a new parsing algorithm. After drilling and performing a derivative on the focused grammar, it is possible that the resulting grammar has become nullable. Nullability of the resulting grammar may violate the second derivative equation for some concatentation node in the context, rendering the next derivative step on the focused grammar unsound. Our last procedure, regress, backs out of nullable parsers:

```
(define (regress Z)
  (match Z
    [(zipper (? nullable? L1) (□ ○ ● L2 k))
     ; ⇒
     (regress (zipper (○ L1 L2) k))]
    [(zipper (? nullable? L2) (● ○ □ L1 k))
     ; ⇒
     (regress (zipper (○ L1 L2) k))]
    [(zipper (? nullable? L1) (□ → ● f k))
     ; ⇒
     (regress (zipper (→ L1 f) k))]
    [(zipper L k)
     ; ⇒
     Z]))
```

11.2. Algorithm

The completed focusing algorithm, and our final optimization for derivative parsing, consists of the sequence: derive, compact, regress, drill. Rinse and repeat. We present images of the parentheses grammar parsing the input '(LP LP RP RP) to supplement understanding of drilling and regression:



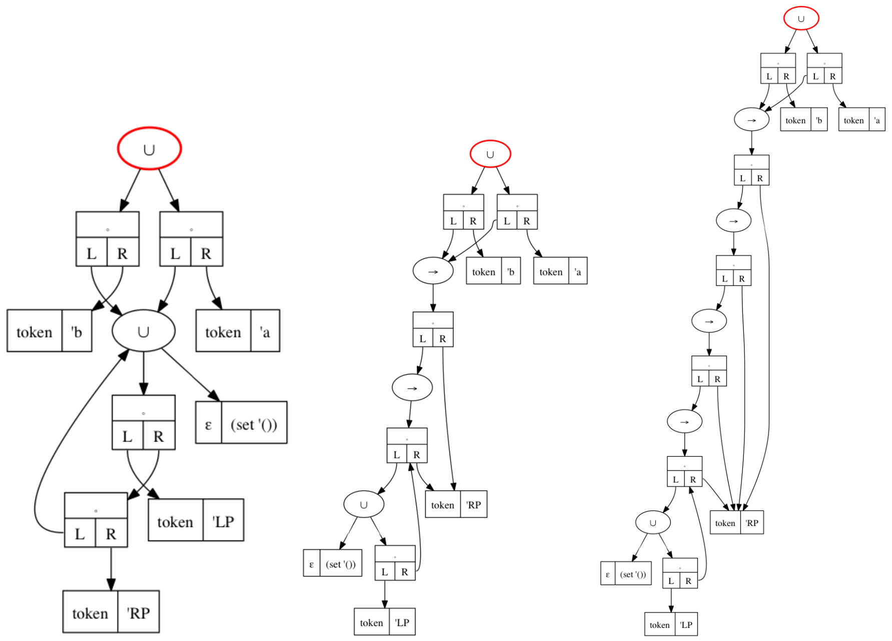
11.3. Complexity

With focusing we are able to keep the subject of the derivative operation constant, and our argument for linear parsing complexity on "sensible" grammars is repeated. Parsing with derivatives can be practical provided two optimizations, compaction and focusing.

To see where focusing is unable to keep parsing time linear for unambiguous but non-LR(k) (and therefore non-"sensible") grammars we present the after-parens grammar:

```
(define after-parens
  (∪ a-after-parens b-after-parens))
(define a-after-parens (○ parens (token 'a)))
(define b-after-parens (○ parens (token 'b)))
```

The grammar is unambiguous; a successfull parse comes from exactly one derivation. However there is rampant ambiguity during parsing: a parser does not know which branch of a-after-parens or b-after-parens to take until the very end. It is through this ambiguity during parsing that after-parens violates LL(k) or LR(k); no k exists under which the branch can always be predicted. Let's observe how parsing with derivatives under focusing looks after zero, two, and four derivatives:



The ambiguity during parsing blocks the zipper algorithm from focusing through the growth of grammar and the size grows linearly with the input. Our algorithm is therefore quadratic when parsing after-parens.

We can imagine yet another optimization which includes zippers *as a proper parser connective*, allowing parsers to recursively contain zipper parsers. In theory this would allow drill to dive through nested contexts, not just those at the top-level, keeping the size of after-parens constant during parsing. However, this addition to the algebra complicates the derivative and other analysis theories, and we leave its development for future work.

11.4. Complexity Conjectures vs Proofs

We do not given full proof of our complexity conjectures because they are just that: conjectures. A proof of complexity on LR(k) languages would involve relating our algorithm to LR(k) grammars, which are defined entirely be the shift/reduce tables which LR parsing algorithms generate. Relating the derivative to shift/reduce tables would be tiresome. However, we believe there is a class of grammars, similar to LR(k) but defined in terms of derivatives, for which parsing with derivatives under focusing *is* provably linear. We leave such a proof for future work, and merely present our preliminary empirical results: we have yet to find an unambiguous LL(k) or LR(k) grammar that our algorithm cannot parse in linear time.

12. 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 (**?**), and Danielsson (**?**) 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 (**?**; **?**), operator-precedence parsing (**?**; **?**), simple precedence parsing (**?**), parser combinators (**?**; **?**), LALR parsing (**?**), LR(k) parsing (**?**), GLR parsing (**?**), CYK parsing (**?**; **?**; **?**), Earley parsing (**?**), LL(k) parsing, and recursive descent parsing (**?**). packrat/PEG parsing (**?**; **?**). 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 (**?**). 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.

12.0.0.1. *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.

13. 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.