Parsing With Derivatives

Leif Andersen
University of Utah

Abstract—**Parsing with derivatives has previously been thought of as an exponential algorithm, which may not terminate on a context free language. As such, very little research has been done on the subject for half a century. This paper talks about the formal definition of a language, what a derivative is, and how to derive a language. Finally, it ends by showing the problems with using derivatives to parse a context free grammar, and how to work around them. It also gives code at every step of the differentiation.**

I. Introduction

There are many solutions to parsing a string in a language. However, most of these solutions are difficult for programmers to perform. Using YACC and Bison lead to many problems. And using LALR parsers do not work on every possible context free language. As such, programmers tend to pre-delimit there data, and use a solution such as **strtok()** to parse the data. While these solutions can be fast, they are very prone to security vulnerabilities[1].

To work around this, one possible solution that has been proposed is building a derivative based parser, which turns out to be surprisingly easy to implement. However, it has been thought that these parsers can only work on regular languages, and would be infinitely recursive on context free languages. Furthermore, the algorithm grows exponentially in size with respect to the size of the input string. Fortunately, there exists methods to reduce the size of the languages, so that it is not recursive, and so that it will terminate on any arbitrary context free language.

A. Layout

This paper will begin by discussing what a formal language is, and then move on to how to describe it in code (both python and racket). It will then explain what a derivative is, and how to implement it code. Next, it explains nullability and how it relates to derivatives, and how to implement it. Finally, it shows problems with implementing the code in a context free language rather than only regular languages, and solutions on how to work around the problems.

II. Formal Languages

A language is nothing more than a set of strings over an alphabet. An alphabet is a set of characters while a string is a sequence of characters. And a character is an atomic symbol. In English, the alphabet is $A, B, C, ..., a, b, ...$, but other alphabets exist, for example, the alphabet that modern computers can understand is $0, 1$. In formal languages, alphabets are finite (but arbitrary) in size, while a language can be finite, as well as infinite.

The simplest possible language is $\emptyset = \{\}$, or the empty language. This language has no strings in it, so every possible string is not in the language. The next simplest language is $\{\epsilon\}$, which is the language of an empty string. It is distinct from the empty language because it actually has an element in it, empty strings. Which is to say ϵ, or the string with no characters is part of $\{\epsilon\}$, but not $\emptyset$.

Finally, the last simple language is $\{c\}$, which is the language of only one single character c (where c is any arbitrary character). Note that $\{c\}$ is also distinct from $\{\epsilon\}$, because not only does $\{\epsilon\}$ contain c, but $\{c\}$ also does not contain ϵ.

Other (more complicated) languages do exist (infinitely more), however, these basic ones are the only ones covered in this section.

A. Language Operations

While this definition of a language is good for most purposes, it isn't enough to properly analyze languages using derivatives. This is because every language can be created by following simple operations on the primary languages listed above.

The first is union ($\cup$). Denoted by two languages L_1 and L_2 as $L_1 \cup L_2$. The resulting language is simply every string in L_1, and every string in L_2. One example is the language of $\{\epsilon, c\}$, which excepts empty strings, and ones with the character c. The formation of this language is $\{\epsilon\} \cup \{c\}$.

The second operation is concatenation ($\circ$). Again, denoted as two language L_1 and L_2 as $L_1 \circ L_2$. The resulting language is every string in the first language, concatenated with every string in the second language. An example is the language $\{ce\}$, which can be formed by $\{c\} \circ \{e\}$.

Finally, there is the Kleene star operation (*), which is denoted with one language L as L^*. This is simply an infinite amount of unions and concatenations over a language L, returning $\{\epsilon\} \cup L \cup L^2 \cup L^3 \cup ...$

All of these operations, in addition to the simple languages listed above, is enough to denote many languages. And with a few additional more languages, every possible language is possible.

B. Language Complexity

Every language that has been described so far has been a regular language. Traditionally, a regular language has been thought of as a language that a DFA or NFA can accept[2]. Other classifications are Context Free Languages, Context Sensitive Languages, which are languages that are accepted by a PDA[2], Context Free Languages,which are languages a Turing machine will accept, and halt on. There is also recursively enumerable, which are Languages a Turing machine accepts. Finally, there is undecidable, which is every other language[2].

The following image shows how each of these sets exist.

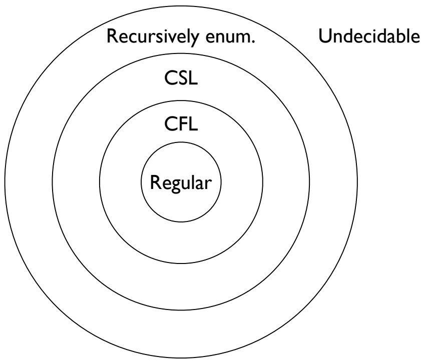

Fig. 1. Language Diagram[3]

Traditional methods of doing derivative based parsing has only worked on regular languages[4]. However, newer methods have expanded the ability to additionally work on context free languages.

III. Encoding Languages

This way of describing a language can be put in a programming language. Examples for Racket and Python are included. Racket is included because it is a staple for writing compilers and parsers. However, being an offshoot of Lisp and Scheme, many people find it hard to understand, therefore a Python version is also included, which while less clean, is just as functional.

While this version does not analyze a stream of characters, it is trivial to write one that reads each character as a regex string, and tuns it into a tree.

A. Racket

Racket contains a struct construct which is similar to structs or classes in a C like language.

Listing 1. Language Construction in Racket
```
(define-struct empty {})
(define-struct epsilon {})
(define-struct char {value})
(define-struct cat {left right})
(define-struct union {a b})
(define-struct star {lang})
```

B. Python

The python code is significantly less elegant. However, it is capable of duplicating racket's define-struct function with classes.

Listing 2. Language Construction in Python
```
class empty:
    def __init__(self):
        pass

class epsilon:
    def __init__(self):
        pass

class character:
    def __init__(self, c):
        self.c = c

class cat:
    def __init__(self, L1, L2):
        self.L1 = L1
        self.L2 = L2

class union:
    def __init__(self, L1, L2):
        self.L1 = L1
        self.L2 = L2

class star:
    def __init__(self, L):
        self.L = L
```

IV. Brzozowski's Derivatives

Brzozowsi's Derivatives is a way to differentiate a language L with respect to a character c, such that you get a resulting language that removes all strings in the language not beginning with that character, and places the remaining strings back in the language without the character. The classic definition is:

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

An example would be to differentiate the set $L = \{foo, bar, baz\}$ with respect to the character b, or $D_b(L)$. The resulting set would be $D_b(L) = \{ar, az\}$[4].

The following rules can be used to describe the derivatives of the atomic languages listed above:

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

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

$$D_c(c) = \epsilon$$

$$D_c(c') = \emptyset \text{ if } c \neq c'.$$

Furthermore, the derivative of the union is the union of the derivatives:

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

The Kleene start is also trivial, because of it's repetitive nature. It simply becomes the derivative of L, followed by L^*. For example, $D_f((foo)^*) = oo(foo)^*$, and $D_g((foo)^*) = \emptyset$. The formal definition of the Derivative of the Kleene star is:

$$D_c(L^*) = D_c(L) \circ L^*$$

Concatenation is a bit tricky, simply because the first language can contain ϵ, and as such the second string (in L_2) may be determine what the resulting language is. Fortunately, this can be worked around by doing two different things based on if ϵ is in the language.

$$D_c(L_1 \circ L_2) = D_c(L_1) \circ L2 \text{ if } \epsilon \notin L_1$$

$$D_c(L_1 \circ L_2) = (D_c(L_1) \circ L2) \cup D_c(L_2) \text{ if } \epsilon \in L_1$$

Racket and python code will be placed in a later section.

V. Nullability Function

Nullability(δ) is a cleaner way of dealing with concatenation[5]. Nullability is a function defined to return ϵ if ϵ is in the input language otherwise it returns $\emptyset$. Formally, it is written as:

$$\delta(L) = \emptyset \text{ if } \epsilon \notin L$$

$$\delta(L) = \epsilon \text{ if } \epsilon \in L$$

The nullability of the atomic languages is fairly straightforward:

$$\delta(\emptyset) = \emptyset$$

$$\delta(\epsilon) = \epsilon$$

$$\delta(c) = \emptyset.$$

The nullability of a union is the union of the nullability:

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

Kleene star, by definition, contains ϵ, so it's nullability is ϵ:

$$\delta(L^*) = \epsilon.$$

Finally, unlike the derivative, the nullability of the concatenation is the concatenation of the nullability:

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

A. Racket

Racket's pattern matching makes it fairly trivial to implement this. This function returns true and false rather than ϵ and $\emptyset$, this is because it will be more useful in future code.

Listing 3. Nullability in Racket
```
(define (δ L)
  (match L
    [(empty)         #f]
    [(epsilon)       #t]
    [(char _)        #f]
    [(star _)        #t]
    [(union L1 L2)  (or (δ L1) (δ L2))]
    [(cat L1 L2)    (and (δ L1) (δ L2))]))
```

B. Python

Python does not have the match function that racket does. However, for this simple application, it has the function call **isinstance()** which takes a class, and an object, and returns true if the class is of the same type as the object. This code also uses true and false like the racket version.

Listing 4. Nullability in Python
```
def δ(L):
    if isinstance(L, empty):
        return False
    elif isinstance(L, epsilon):
        return True
    elif isinstance(L, character):
        return False
    elif isinstance(L, star):
        return True
    elif isinstance(L, union):
        return δ(L.L1) or δ(L.L2)
    elif isinstance(L, cat):
        return δ(L.L1) and δ(L.L2)
```

VI. Putting it all Together

The nullability function allows for a simpler way to define the derivative of a concatenation. Because concatenating any language L with $\emptyset$ becomes $\emptyset$, and concatenating that same language L with ϵ becomes L, the derivative can simply be written as:

$$D_c(L_1 \circ L_2) = (D_c(L_1) \circ L2) \cup (\delta(L_1) \circ D_c(L_2))$$

Furthermore, with these two functions together (nullability and derivative). It is possible to write a parser for a string. The algorithm is fairly straightforward:

1) For every character in the string, differentiate the language by that character.
2) If the resulting language is nullable, then the original string is in the language, otherwise it is not.

For example, to test if $foo \in (foo)^*$, the following can be done:

1) $foo \in (foo)^*$
2) $D_f(foo) \in D_f((foo)^*)$
3) $oo \in oo(foo)^*$
4) $D(oo) \in D_o(oo(foo)^*)$
5) $o \in o(foo)^*$
6) $D_o(o) \in D_o(o(foo)^*)$
7) $\epsilon \in (foo)^*$
8) $\delta(\epsilon) = \epsilon$
9) Matches: $foo \in (foo)^*$

Alternatively, to test if $a \in b$ is:

1) $a \in b$
2) $D_a(a) \in b$
3) $\epsilon \in \emptyset$
4) $\delta(\epsilon)$
5) Doesn't Match: $a \notin b$

A. *Racket*

The definition for derivative is given here. It also uses rackets match construct.

Listing 5. Derivation in Racket

```
(define (D c L)
  (match L
    [(empty)
     (empty)]

    [(epsilon)
     (empty)]

    [(char a)
     (if (equal? c a)
         (epsilon)
         (empty))]

    [(union L1 L2)
     (union (D c L1) (D c L2))]

    [(cat (and (? δ) L1) L2)
     (union (D c L2)
            (cat (D c L1) L2))]

    [(cat L1 L2)
     (cat (D c L1) L2)]

    [(star L1)
     (cat (D c L1) L)]))
```

With this definition in mind, it is also trivial to make a matches function using recursion.

Listing 6. Matching in Racket

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

B. *Python*

As before, Python has no matches function, but **isinstanceof()** continues to be a good substitute for match.

Listing 7. Derivation in Python

```
def D(c, L):
    if isinstance(L, empty):
        return empty()
    elif isinstance(L, epsilon):
        return empty()
    elif isinstance(L, character):
        if c == L.c:
            return epsilon()
        else:
            return empty()
    elif isinstance(L, union):
        return union(D(c, L.L1),
                     D(c, L.L2))
    elif isinstance(L, cat):
        if δ(L.L1):
            return union(D(c, L.L2),
                         cat(D(c, L.L1), L.L2))
        else:
            return cat(D(c, L.L1), L.L2)
    elif isinstance(L, star):
        return cat(D(c, L.L), L)
```

Finally, like racket, a recursive based matches is also possible.

Listing 8. Matching in Python

```
def matches(w, L):
    if w == "":
        return δ(L)
    else:
        return matches(w[1:], D(w[0], L))
```

VII. Extending to Context Free Languages: Problems

The problem with extending the rules to a context free language is that it could lead to an infinite recursion. One way to think of a context free language is as a recursive regular language, and the following is fairly common to see as a context free language:

$$S ::= Sa|\epsilon$$

While this is technically the language a^*, not all context free grammars are specifiable to this form. Take the derivative of S with respect to some arbitrary character c, formally, it is:

$$D_c(S) = D_c(S)a|D_c(\epsilon).$$

Or in other words, to calculate the derivative of S, one must first know the derivative of S. It is trivial to see how this is infinitely recursive.

VIII. Extending to Context Free Languages: Solutions

For nearly half a century, it was thought that it was impossible to differentiate a context free language for the reasons stated above. However, recently, research has been done to stop the infinite recursion [6]. Using lazyness, memoization, and fixed points, it is possible to determine if a string is in a context free grammar.

Furthermore, while differentiation is usually thought of as an exponential time algorithm, in practice, these methods turn it into a polynomial time algorithm, even if it is still theoretically exponential in the worst case.

A. *Lazyness*

Racket is a language which eagerly evaluates the code it is given. Fortunately, with the use of macros, it is possible to make it evaluate lazily. Doing this will keep the extra derivative in the equation, but won't evaluate it until needed[7].

B. Memoization

This algorithm works until nullability is needed to be calculated. At which point, it will become infinitely recursive again. Fortunately, with the use of the **define/memoize** function, it is possible to define a version of derivative that will work. Only racket code is provided as there is no equivalent function for python.

Listing 9. Memoize in Racket
```
(define/memoize (D c L)
  #:order ([L #:eq] [c #:equal])
  (match L
    [(empty)
     (empty)]

    [(eps)
     (empty)]

    [(char a)
     (if (equal? c a)
         (eps)
         (empty))]

    [(union L1 L2)
     (union (D c L1)
            (D c L2))]

    [(cat (and (? δ) L1) L2)
     (union (D c L2)
            (cat (D c L1) L2))]

    [(cat L1 L2)
     (cat (D c L1) L2)]

    [(star L1)
     (cat (D c L1) L)]))
```

C. Fixed Points

Nullability is still a bit tricky. Unlike derivative, where a structure is needed, a definite answer is required, either yes it is nullable, or no, it is not. With the user of **define/fix**, it becomes possible to use fixed points to stop the recursion. Only racket code is provided as there is no similar function in python.

Listing 10. Fixed Point in Racket
```
(define/fix (δ L)
  #:bottom #f
  (match L
    [(empty)         #f]
    [(eps)           #t]
    [(char _)        #f]
    [(star _)        #t]
    [(union L1 L2)  (or (δ L1) (δ L2))]
    [(cat L1 L2)    (and (δ L1) (δ L2))]))
```

IX. Conclusions

The methods mentioned in this paper will determine if a string is in the language. While they don't do any actual parsing (turn the input string into a tree of useful data), they can easily be extended to do this.

This method of parsing is unique because it is easier to implement, which will allow programmers to write parsers correctly, rather than simply slapping them together with **strtok**()[1].

In addition to being thought of as impossible, this method was also thought to be very slow, however, after testing, it was shown to be practically polynomial[5].

References

[1] M. Might, "Parsing with Derivatives." [Online]. Available: http://www.youtube.com/watch?v=ZzsK8Am6dKU.

[2] G. Gopalakrishnan and T. Sorensen, "Modeling and reasoning about computation:theory and applications of automata, languages, undecidability, bdd, sat, and smt methods through declarative programming.".

[3] M. Might, "Lexical Analysis." [Online]. Available: http://matt.might.net/teaching/scripting-languages/spring-2012/lectures/05-lexing.pdf.

[4] J. Brzozowski, "Derivatives of regular expressions," *Journal of the ACM (JACM)*, vol. 11, no. 4, pp. 481–494, 1964.

[5] M. Might, "Yacc is dead: An update." [Online]. Available: http://matt.might.net/articles/parsing-with-derivatives/.

[6] M. Might, D. Darais, and D. Spiewak, "Parsing with derivatives, a functional pearl." [Online]. Available: http://matt.might.net/papers/might2011derivatives.pdf.

[7] M. Might, "Parsing With Derivatives (Racket Version)." [Online]. Available: http://matt.might.net/articles/parsing-with-derivatives/code/dparse.rkt.