Parsing with Derivatives
(Yacc is Dead)

Matt Might & David Darais
University of Utah
matt.might.net
WARNING
Not yet true.
What is parsing?
Unstructure to structure.
A transformation from a sequence to a tree.
(x + 3) * 10
$$(x + 3) \times 10$$
(x + 3) * 10
“Parsing is ubiquitous.”

Trevor Jim
Compilers

Interpreters

Syntax highlighting

Natural language

Network protocols

Command lines

Configuration files

Serialization

Query languages

API design
Compilers

Interpreters

Syntax highlighting

Natural language

Network protocols

Command lines

Configuration files

Serialization

Query languages

API design
RFC 2616 (HTTP 1.1)
2.1 Augmented BNF

All of the mechanisms specified in this document are described in both prose and an augmented Backus-Naur Form (BNF) similar to that used by RFC 822 [9]. Implementors will need to be familiar with the notation in order to understand this specification. The augmented BNF includes the following constructs:
LWS = [CRLF] 1*( SP | HT )

separators = "(" | ")" | ":" | ";" | "\" | ";" | "<" | ">" | "@" |
| "," | ":" | ";" | "<" | ">" | "@" | SP | HT

HTTP-Version = "HTTP" / "1*DIGIT "." 1*DIGIT

http_URL = "http: "//" host [ ":" port ] [ abs_path [ "?" query ]]

media-type = type "/" subtype *( ";" parameter )
type = token
subtype = token

HTTP-date = rfc1123-date | rfc850-date | asctime-date
rfc1123-date = wkday "," SP date1 SP time SP "GMT"
rfc850-date = weekday "," SP date2 SP time SP "GMT"
asctime-date = wkday SP date3 SP time SP 4DIGIT
date1 = 2DIGIT SP month SP 4DIGIT
; day-month-year (e.g., 02-Jun-82)
date2 = 2DIGIT SP month SP 4DIGIT
; day-month-year (e.g., 02-Jun-82)
date3 = month SP ( 2DIGIT | ( SP 1DIGIT ))

wkday = "Mon" | "Tue" | "Wed"
weekday = "Monday" | "Tuesday" | "Wednesday"

month = "Jan" | "Feb" | "Mar" | "Apr"
| "May" | "Jun" | "Jul" | "Aug"
| "Sep" | "Oct" | "Nov" | "Dec"

http_date = [CRLF] 1*( SP | HT )

chunked_body = *chunk
last_chunk
trailer
CRLF

chunk = chunk-size [ chunk-extension ] CRLF
chunk-data CRLF

chunk-size = 1*HEX

last_chunk = 1*("0") [ chunk-extension ] CRLF

chunk_extension = *( ";" chunk-ext-name [ ";" chunk-ext-val ] )
chunk_ext_name = token
chunk_ext_val = token | quoted-string
chunk_data = chunk-size(OCTET)

trailer = *(entity-header CRLF)
RFC 3501 (IMAPv4)
RFC 2812 (IRC)
The Augmented BNF representation for this is:

```plaintext
message = [ "":" prefix SPACE ] command [ params ] crlf
prefix = servername / ( nickname [ "!" user [ "@" host ] ] )
command = 1*letter / 3digit
params = *14( SPACE middle ) [ SPACE ":" trailing ]
       /= 14( SPACE middle ) [ SPACE [ ":" ] trailing ]
nospcrlfcl = %x01-09 / %x0B-0C / %x0E-1F / %x21-39 / %x3B-FF
          ; any octet except NUL, CR, LF, " " and ":"
middle = nospcrlfcl *( ";" / nospcrlfcl )
trailing = *( ":" / " " / nospcrlfcl )
SPACE = %x20 ; space character
crlf = %x0D %x0A ; "carriage return" "linefeed"

target = nickname / server
msgtarget = msgto *( "," msgto )
msgto = channel / ( user [ "@" host ] "@" servername )
       /= ( user "@" host ) / targetmask
msgto =/ nickname / ( nickname "!" user "@" host )
channel = ( ";" / "+" / ( ";" channelid ) / ";" ) chanstring
       [ ";" chanstring ]
servername = hostname
host = hostname / hostaddr
hostname = shortname *( "." shortname )
shortname = ( letter / digit ) *( letter / digit / ";" )
           *( letter / digit )
           ; as specified in RFC 1123 [HNAME]
hostaddr = ip4addr / ip6addr
ip4addr = 1*3digit ";" 1*3digit ";" 1*3digit ";" 1*3digit
ip6addr = 1*hexdigit 7( ";" 1*hexdigit )
ip6addr =/ "0:0:0:0:0:0:0:0:" ( "0" / "FFFF" ) ";" ip4addr
nickname = ( letter / special ) *8( letter / digit / special / ";" )
targetmask = ( ";S" / ";#" ) mask
          ; see details on allowed masks in section 3.3.1
chanstring = %x01-07 / %x08-09 / %x0B-0C / %x0E-1F / %x21-2B
chanstring =/ %x2D-39 / %x3B-FF
           ; any octet except NUL, BELL, CR, LF, ";", ";", and ":"
channelid = 5( %x41-5A / digit ) ; 5( A-Z / 0-9 )
user = 1*( %x01-09 / %x0B-0C / %x0E-1F / %x21-3F / %x41-FF )
       ; any octet except NUL, CR, LF, " " and "@"
key = 1*23( %x01-05 / %x07-08 / %x0C / %x0E-1F / %x21-7F )
     ; any 7-bit US_ASCII character,
     ; except NUL, CR, LF, FF, h/v TABs, and " 
letter = %x41-5A / %x61-7A ; A-Z / a-z
digit = %x30-39 ; 0-9
hexdigit = digit / "A" / "B" / "C" / "D" / "E" / "F"
special = %x5B-60 / %x7B-7D
        ; ";", ";", ";", ";", ";", ";", "{", ";", ";", ";"}"
```
Efficient parsing techniques exist.
Parsing tools abound.
State of the art?
*buf++
2,179 lines of C
lighttpd
1,211 lines of C
freenode IRCD
> 2000 lines of C
Courier IMAP
2,633 lines of C
Result?
**Vulnerability Details:**  
**CVE-ID:** CVE-2004-0786  
**Description:** IRCnet IRCD Buffer Overflow

<table>
<thead>
<tr>
<th>Details</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CVE-ID</strong></td>
<td>CVE-2004-0786</td>
</tr>
<tr>
<td><strong>Secunia ID</strong></td>
<td>SA9999</td>
</tr>
<tr>
<td><strong>Release Date</strong></td>
<td>16 Oct 2003</td>
</tr>
<tr>
<td><strong>Last Change</strong></td>
<td>20 Oct 2 **003</td>
</tr>
<tr>
<td><strong>Criticality</strong></td>
<td>Not Critical</td>
</tr>
<tr>
<td><strong>Solution Status</strong></td>
<td>Vendor Patch</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td>IRCD 2.x</td>
</tr>
<tr>
<td><strong>Where</strong></td>
<td>Local system</td>
</tr>
</tbody>
</table>

**Immune systems:**  
* mIRC version 6.11

**Vulnerable systems:**  
* mIRC version 6.1 and prior

**From:** Matias Soler  <g
**Date:** Tue, 2 Jan 2007 17

**Synopsis:** Apache 1.3
**Version:** 1.3.37 (latest)

**Solution**  
Vendor Patch

**Where**  
Local system

**Impacted**  
Apache 1.3
**Version:** 1.3.37 (latest)

**Product**  
Apache htpasswd Util

**Comp**  
Apache htpasswd Util

**V**  
Incorrect validation on the size of user input allows to copy a string, via strcpy, to a fixed size buffer. File: htpasswd.c, Line 421.

**Keyword(s):** FixedInTrunk, PatchAvailable

**Details**  
**Assign:**
Incorrect validation on the size of user input allows to copy a string, via strcpy, to a fixed size buffer. File: htpasswd.c, Line 421.

**Mar 11 2004 12:00AM**
These issues were disclosed by the vendor.

**Jul 12 2009 03:06AM**

**Inter7 Courier-IMAP 2.2.1**

**Inter7 Courier-IMAP 2.2.0**

**Inter7 Courier-IMAP 2.1.2**

**Inter7 Courier-IMAP 2.1.1**

**Inter7 Courier-IMAP 2.0**

**License:** GPL

**License ID:** 00000000
“High-performance network daemon”
“High-throughput exploit server”
Yet...
All of them also use lex and yacc.
(For config files.)
The tools exist.
They know how to use them.
So, why don’t they?
Compilers
  Interpreters
    Syntax highlighting
      Natural language
        Network protocols
          Command lines
            Configuration files
              Serialization
                Query languages
                  API design
$ cat /etc/*
man.conf
passwd
sudoers
notify.conf
protocols
hosts
services
networks
crontab
fstab
bind zones
syslog.conf
asl.conf
ftpd.conf
gettytab
networks
A tribute to false dichotomy.
Human
versus
Machine
There is no tradeoff.
XML: a compromise solution to a problem that does not exist.
Neither human nor machine-readable.
The tools exist.
So, why don’t we use them?
Compilers

Interpreters

Syntax highlighting

Natural language

Network protocols

Command lines

Configuration files

Serialization

Query languages

API design
Compilers

Interpreters

Syntax highlighting

Natural language

Network protocols

Command lines

Configuration files

Serialization

Query languages

API design
```c
#include <stdio.h>
#include <stdlib.h>

typedef unsigned int count;
count parens1;
unsigned int parens2;
```
#include <stdio.h>
#include <stdlib.h>

typedef unsigned int count;

count parens1;
unsigned int parens2;
emacs, vi: over 30 years
WHY!?
where we use it
A list of grievances
Opportunity cost

Transaction cost

Barriers to entry
Opportunity cost
What do you sacrifice with a parser generator?
Performance
Thread-safety
Non-blocking I/O
Thread-safety
Non-blocking I/O
Performance
Thread safety
Non-blocking I/O
What if a server blocks on I/O?
DoS
Transaction cost
Time to set up lex & yacc

versus

Time to hack it with strtok
lex/yacc

strtok()
lex/yacc
Create .y yacc file

strtok()
lex/yacc
Create .y yacc file
Define %union lval

strtok()
lex/yacc

Create .y yacc file
Define %union lval
Define tokens/types

strtok()
lex/yacc

Create .y yacc file
Define %union lval
Define tokens/types
Define yacc rules

strtok()
lex/yacc
Create .y yacc file
Define %union lval
Define tokens/types
Define yacc rules
Compile yacc spec

strtok()
lex/yacc

Create .y yacc file
Define %union lval
Define tokens/types
Define yacc rules
Compile yacc spec
Create .l lex file

strtok()
lex/yacc

Create .y yacc file
Define %union lval
Define tokens/types
Define yacc rules
Compile yacc spec
Create .l lex file
#include y.tab.h

strtok()
lex/yacc

Create .y yacc file
Define %union lval
Define tokens/types
Define yacc rules
Compile yacc spec
Create .l lex file
#include y.tab.h
Define states

strtok()
lex/yacc

Create .y yacc file
Define %union lval
Define tokens/types
Define yacc rules
Compile yacc spec
Create .l lex file
#include y.tab.h
Define states
Define lex rules

strtok()
<table>
<thead>
<tr>
<th>lex/yacc</th>
<th>strtok()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create .y yacc file</td>
<td></td>
</tr>
<tr>
<td>Define %union lval</td>
<td></td>
</tr>
<tr>
<td>Define tokens/types</td>
<td></td>
</tr>
<tr>
<td>Define yacc rules</td>
<td></td>
</tr>
<tr>
<td>Compile yacc spec</td>
<td></td>
</tr>
<tr>
<td>Create .l lex file</td>
<td></td>
</tr>
<tr>
<td>#include y.tab.h</td>
<td></td>
</tr>
<tr>
<td>Define states</td>
<td></td>
</tr>
<tr>
<td>Define lex rules</td>
<td></td>
</tr>
<tr>
<td>Deal with yywrap()</td>
<td></td>
</tr>
</tbody>
</table>
lex/yacc

Create .y yacc file
Define %union lval
Define tokens/types
Define yacc rules
Compile yacc spec
Create .l lex file
#include y.tab.h
Define states
Define lex rules
Deal with yywrap()
Call yyparse()
lex/yacc

Create .y yacc file
Define %union lval
Define tokens/types
Define yacc rules
Compile yacc spec
Create .l lex file
#include y.tab.h
Define states
Define lex rules
Deal with yywrap()
Call yyparse()

strtok()

Call strtok()
Barriers to entry
How does yacc work?
start : exp { printf("ans: \%i\n", $1 ); } 

exp : exp '+' term { $$ = $1 + $3 ; } 
    | term { $$ = $1 ; } 

term : term '*' factor { $$ = $1 * $3 ; } 
    | factor { $$ = $1 ; } 

factor : INT { $$ = $1 ; } 
    | SYM { $$ = lookup($1) ; } 
    | '(' exp ')' { $$ = $2 ; } 

Reduce

E → \_\_\_

\_\_\_ → \_\_\_
All good until...
$ bison -d exp.y
exp.y: conflicts: 57 shift/reduce
$ bison -d exp.y
exp.y: conflicts: 57 shift/reduce

What’s in the black box?
$ wc -l exp.output
5109 exp.output
state 301:

2 exp: exp . '+' exp  [$end, '+', '*', ')$']
2    | exp '+' exp .  [$end, '+', '*', ')$']
3    | exp . '*' exp

'+'  shift, and go to state 8
'*$'  shift, and go to state 9

'+'  [reduce using rule 2 (exp)]
'*$'  [reduce using rule 2 (exp)]
$default  reduce using rule 2 (exp)
\texttt{%left ' + '
\texttt{%left ' * '}

$ bison -d exp.y
exp.y: conflicts: 53 shift/reduce
Two days later...
$ bison -d exp.y
exp.y: conflicts: 1 shift/reduce
$ bison -d exp.y
exp.y: conflicts: 1 shift/reduce

“Probably just dangling else.”
Barrier to entry: One must *learn* each tool.
Barrier to entry:
Tool must exist for language.
How do we change the economics of parsing?
What we need

- Allow non-blocking I/O
- Act as library within the language
- Handle all CFGs: parse forest
- Draw on regular expressions
- Cacheable partial parses
- Be easy to implement

(Jim, Mandelbaum, Walker, POPL 2010)
Would be nice

- Dynamic reconfigurability
- Beyond context-free languages
- Built into the language
- Support compositionality

(Jim, Mandelbaum, Walker, POPL 2010)
Parsing with derivatives
Derivatives!?
Derivatives of Regular Expressions

Janusz A. Brzozowski

Princeton University, Princeton, New Jersey

Abstract. Kleene's regular expressions, which can be used for describing sequential circuits, were defined using three operators (union, concatenation and iterate) on sets of sequences. Word descriptions of problems can be more easily put in the regular expression language if the language is enriched by the inclusion of other logical operations. However, in the problem of converting the regular expression description to a state diagram, the existing methods either cannot handle expressions with additional operators, or are made quite complicated by the presence of such operators. In this paper the notion of a derivative of a regular expression is introduced and the properties of derivatives are discussed. This leads, in a very natural way, to the construction of a state diagram from a regular expression containing any number of logical operators.
A language is a set of strings.
Regular languages

\[ \epsilon \equiv \{ \\
\emptyset \equiv \\emptyset \}
\]

\[ c \equiv \{ c \}
\]
Regular languages

\[ L_1 \cup L_2 \]

\[ L_1 \cdot L_2 \]

\[ L_1^* \]
(foo|bar)*
\{ foo, bar \}^*
A less common, yet still useful, operation on formal languages is the derivative. The left derivative of a formal language \( L \) with respect to character \( c \), denoted \( D_c L \), is the remainder of the strings in the set \( L \) for which the character \( c \) can be removed from the front:

\[
D_c L = \{ w : cw \in L \}
\]

Example 3.1. The left derivative of the set \( \{ \text{foo}, \text{bar} \} \) with respect to the character \( c \) is:

\[
D_c \{ \text{foo}, \text{bar} \} = \{ \text{oo}, \text{rak} \}
\]

Where the derivative of a language is computable, it may be possible to a string in that language using derivatives, thanks to the following rule:

\[
w \in D_c y L \Rightarrow cw \in L.
\]

If it is computable to take successive derivatives of a language, and it is possible to test whether or not one of those derivatives accepts the empty string, then it is possible to use the derivative to test whether a string is in a language. The algorithm is straightforward:

1) Compute the derivative of the language with respect to each character in the string; and
2) Test whether the resulting language accepts the empty string.

3.3.1 Computing the derivative

For languages composed of other languages through the familiar formal language operations, it is frequently possible to formulate the derivative of such a language recursively.

For the empty language, the derivative is empty:

\[
D_c \emptyset = \emptyset.
\]

For the empty-string language, the derivative is also empty:

\[
D_c \{ \varepsilon \} = \emptyset.
\]

For one-character languages:

\[
D_c \{ c \} = \{ \varepsilon \}
\]

\[
D_c \{ c \varepsilon \} = \emptyset \text{ if } c \neq c \varepsilon.
\]
The derivative

1. **Filter**: Keep every string starting with $c$.

2. **Chop**: Remove $c$ from the start of each.
\[ D_f \{ \text{foo, frak, bar} \} = \{ \text{oo, rak} \} \]
\[ D_f\{\text{foo, frak, bar}\} = \{\text{oo, rak}\} \]
\[ D_b\{\text{foo, frak, bar}\} = \{\text{ar}\} \]
\( D_f \{ \text{foo, frak, bar} \} = \{ \text{oo, rak} \} \)
\( D_b \{ \text{foo, frak, bar} \} = \{ \text{ar} \} \)
\( D_x \{ \text{foo, frak, bar} \} = \emptyset \)
\[ D_f \{ \text{foo, bar} \}^* = \]
\[ D_f \{\text{foo, bar}\}^* = \{\text{oo}\} \cdot \{\text{foo, bar}\}^* \]
\( D_f \{\text{foo, bar}\}^* \cdot \{\text{frak}\} = \)
\[ D_f\{\text{foo, bar}\}^* \cdot \{\text{frak}\} = \{\text{oo}\} \cdot \{\text{foo, bar}\}^* \cdot \{\text{frak}\} \cup \{\text{rak}\} \]
Observation

\[ cw \in L \iff w \in D_c(L). \]
Matching algorithm

- Derive with respect to each character.
- Does the derived language contain $\varepsilon$?
foo ∈ \{foo, bar\}^*?
\[ D_f \{\text{foo, bar}\}^* = \{\text{oo}\} \cdot \{\text{foo, bar}\}^* \]
\[ D_f \{\text{foo, bar}\}^* = \{\text{oo}\} \cdot \{\text{foo, bar}\}^* \]

\[ D_o \{\text{oo}\} \cdot \{\text{foo, bar}\}^* = \{\text{o}\} \cdot \{\text{foo, bar}\}^* \]
\[ D_f \{\text{foo, bar}\}^* = \{\text{oo}\} \cdot \{\text{foo, bar}\}^* \]

\[ D_o \{\text{oo}\} \cdot \{\text{foo, bar}\}^* = \{\text{o}\} \cdot \{\text{foo, bar}\}^* \]

\[ D_o \{\text{o}\} \cdot \{\text{foo, bar}\}^* = \{\text{foo, bar}\}^* \]
Nullability

\[
\delta(L) = \epsilon \text{ if } \epsilon \in L \\
\delta(L) = \emptyset \text{ if } \epsilon \notin L
\]
\[ \delta(\epsilon) = \epsilon \]
\[ \delta(\emptyset) = \emptyset \]
\[ \delta(L_1 \cup L_2) = \delta(L_1 \cup \delta(L_2)) \]
\[ \delta(L_1 \cup L_2) = \delta(L_1) \cup \delta(L_2) \]
\[ \delta(L_1 \cdot L_2) = \delta(L_1) \cdot \delta(L_2) \]
\[ \delta(L_1^*) = \epsilon \]
(define (δ L)
    (match L
        [(empty)       #f]
        [(eps)         #t]
        [(char _)      #f]
        [(rep _)       #t]
        [(or  L1 L2)   (or  (δ L1) (δ L2))]
        [(seq L1 L2)   (and (δ L1) (δ L2))])))
Case: Empty set

\[ D_c \emptyset = \]
Case: Empty set

\[ D_c \emptyset = \emptyset \]
Case: Empty string

\[ D_c(\epsilon) = \]
Case: Empty string

\[ D_c(\epsilon) = \emptyset \]
Case: Character

\[ D_c \{c\} = \emptyset \quad \text{if} \quad c' = c \]
Case: Character

\[ D_c\{c\} = \epsilon \]
\[ D_c\{c'\} = \emptyset \text{ if } c \neq c' \]
Case: Union

\[ D_c(L_1 \cup L_2) \]
Case: Union

\[ D_c(L_1 \cup L_2) = \{ w : cw \in L_1 \cup L_2 \} \]
\[ = \{ w : cw \in L_1 \text{ or } cw \in L_2 \} \]
\[ = \{ w : w \in D_cL_1 \text{ or } w \in D_cL_2 \} \]
\[ = \{ w : w \in D_cL_1 \} \cup \{ w : w \in D_cL_2 \} \]
\[ = D_cL_1 \cup D_cL_2. \]
Case: Intersection

\[ D_c(L_1 \cap L_2) = \]
Case: Intersection

\[ D_c(L_1 \cap L_2) = D_cL_1 \cap D_cL_2. \]
Case: Difference

\[ D_c(L_1 - L_2) = \]
Case: Difference

\[ D_c(L_1 - L_2) = D_cL_1 - D_cL_2. \]
Case: Catenation

\[ D_c(L_1 \cdot L_2) = \]
Case: Catenation

\[ D_c(L_1 \cdot L_2) = (D_cL_1 \cdot L_2) \cup (\delta(L_1) \cdot D_cL_2) \]
Case: Complement

\[ D_{cL} \]
Case: Complement

\[ D_c \overline{L} = \{ w : cw \in \overline{L} \} \]
\[ = \{ w : cw \notin L \} \]
\[ = \{ w : \text{not} \ cw \in L \} \]
\[ = \{ w : cw \in L \} \]
\[ = \{ w : w \in D_c L \} \]
\[ = \overline{D_c L}. \]
Case: Option

\[ D_c(L^?) = \]

\[ = D_c L_1 \cup D_c L_2. \]
Case: Option

\[ D_c(L?) = D_cL. \]
Case: Kleene star

\[ D_c(L^*) = \]
Case: Kleene star

\[ D_c(L^*) = (D_cL) \cdot L^* \]
Case: Kleene plus

\[ D_c(L^+) = \]
Case: Kleene plus

\[ D_c(L^+) = (D_cL) \cdot L^* \]
(define (D c L)
  (match L
      [(empty)   (empty)]
      [(eps)     (empty)]
      [(char c)  (eps)]
      [(char _)  (empty)]
      [(or L1 L2) (alt (D c L1) (D c L2))]
      [(seq (and (? nullable?) L1) L2) (alt (D c L2) (cat (D c L1) L2))]
      [(seq L1 L2) (cat (D c L1) L2)]
      [(rep L1)  (cat (D c L1) L)]))
How about context-free grammars?
What’s a context-free grammar?
Recursive regular expressions.
\[ P = (\cdot P \cdot) \cdot P \]

\[ \cup \epsilon \]
The derivative?
Works the same, but...
Problem

\[ L = L \cdot x \]

U \in \varepsilon
Problem

\[ D_x L = D_x L \cdot x \cup \epsilon \]
(define (D c L)
  (match L
    [(empty)        (empty)]
    [(eps)          (empty)]
    [(char c)       (eps)]
    [(char _)       (empty)]
    [(or L1 L2)     (alt (D c L1) (D c L2))]
    [(seq (and (nullable?) L1) L2) (alt (D c L2) (cat (D c L1) L2))]
    [(seq L1 L2)    (cat (D c L1) L2)]
    [(rep L1)       (cat (D c L1) L)])
(define (D c L)
  (match L
    [(empty)  (empty)]
    [(eps)    (empty)]
    [(char c) (eps)]
    [(char _) (empty)]
    [(or L1 L2) (alt (D c L1) (D c L2))]
    [(seq (and (nullable?) L1) L2) (alt (D c L2) (cat (D c L1) L2))]
    [(seq L1 L2) (cat (D c L1) L2)]
    [(rep L1) (cat (D c L1) L)]))
Problem

\[ \delta(L) = \delta(L) \cdot \delta(x) \]

\[ \cup \delta(\epsilon) \]
Solution?
Laziness
Memoization
Fixed points
Laziness delays computation until moment of use.
Memoization caches return values.
Fixed points solve the chicken-and-egg problem.
Laziness

(define-syntax cat
  (syntax-rules ()
    [(_ L1 L2) (concatenation L1 L2)]))
Laziness

(define-syntax cat
  (syntax-rules ()
    [(_ L1 L2)  (concatenation (delay L1) (delay L2))])))
Memoization

(define (D c L)
  (match L
    [(empty) (empty)]
    [(eps) (empty)]
    [(char c) (eps)]
    [(char _) (empty)]
    [(or L1 L2) (alt (D c L1) (D c L2))]
    [(seq (and (nullable?) L1) L2) (alt (D c L2) (cat (D c L1) L2))]
    [(seq L1 L2) (cat (D c L1) L2)]
    [(rep L1) (cat (D c L1) L)])
Memoization

(define/memoize (D c L)
    #:order ([L #:eq] [c #:equal])
    (match L
        [(empty)      (empty)]
        [(eps)        (empty)]
        [(char c)     (eps)]
        [(char _)     (empty)]
        [(or L1 L2)                       (alt (D c L1)
                                                      (D c L2))]
        [(seq (and (nullable?) L1) L2)    (alt (D c L2)
                                                      (cat (D c L1) L2))]
        [(seq L1 L2)                      (cat (D c L1) L2)]
        [(rep L1)                         (cat (D c L1) L)])
)
Fixed points

(define (δ L)
  (match L
    [(empty)       #f]
    [(eps)         #t]
    [(char _)      #f]
    [(rep _)       #t]
    [(or L1 L2)   (or  (δ L1) (δ L2))]
    [(seq L1 L2)   (and (δ L1) (δ L2))])))
Fixed points

(define/fix (δ L)
    #:bottom #f
    (match L
        [(empty)   #f]
        [(eps)     #t]
        [(char _ ) #f]
        [(rep _ )  #t]
        [(or  L1 L2) (or  (δ L1) (δ L2))]
        [(seq L1 L2) (and (δ L1) (δ L2))])))
Now it works.
(For recognizing.)
What about parsing?
What is a parser?
\( \mathcal{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*) \)
\( \mathbb{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*) \)
\[ \mathcal{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*) \]
\( \mathbb{P}(A, T) = A^* \rightarrow \mathcal{P}(T \times A^*) \)
\[ [\mathcal{P}] (A, T) = A^* \rightarrow \mathcal{P}(T) \]
Input string

\[ \mathcal{P}(A, T) = A^* \rightarrow \mathcal{P}(T) \]
\[ \mathcal{P}(A, T) = A^* \rightarrow \mathcal{P}(T) \]
\[ p \in \mathbb{P}(A, T) \]

\[ [p](w) = \{ t : (t, \epsilon) \in p(w) \} \]
Context-free parsers
\[ w \equiv \lambda w'. \begin{cases} \{(w, w'')\} & w' = ww'' \\ \emptyset & \text{otherwise.} \end{cases} \]
\[ \varepsilon \equiv \lambda w. \{ (\varepsilon, w) \} \]
∅ ≡ \lambda w. \{\}
\[ p \in \mathbb{P}(A, X) \]
\[ q \in \mathbb{P}(A, Y) \]
\[ p \cdot q \in \mathbb{P}(A, X \times Y) \]
\[ p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\} \]
\[ p \cdot q = \lambda w. \{ ((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w') \} \]

Input
\[p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\}\]
$p \cdot q = \lambda w. \{(x, y, w'') : (x, w') \in p(w), (y, w'') \in q(w')\}$
\[ p \cdot q = \lambda w.\{(x, y, w''): (x, w') \in p(w), (y, w'') \in q(w)\} \]
\[ p \cdot q = \lambda w.\{((x, y), w'') : (x, w') \in p(w), (y, w'') \in q(w')\} \]
\[ p \in \mathbb{P}(A, X) \]
\[ q \in \mathbb{P}(A, X) \]
\[ p \cup q \in \mathbb{P}(A, X) \]
\[ p \cup q = \lambda w.p(w) \cup q(w) \]
\[ f \in X \rightarrow Y \]
\[ p \in \mathcal{P}(A, X) \]
\[ p \rightarrow f \in \mathcal{P}(A, Y) \]
\[ p \rightarrow f = \lambda w.\{((f(x), w') : (x, w') \in p(w)\}\]
Defining the derivative
\[ D_c : L \rightarrow L \]
$D_c : \mathbb{P}(A, T) \rightarrow \mathbb{P}(A, T)$
\[ D_c(p) = \lambda w. p(cw) - ([p](\epsilon) \times \{cw\}) \]
\[ D_c(p) = \lambda w. p(cw) - (\|p\|(\epsilon) \times \{cw\}) \]

\[ p(cw) = D_c(p)(w) \cup (\|p\|(\epsilon) \times \{cw\}) \]
\[ [p](cw) = [D_c(p)](w) \]
Calculating the derivative
\[ D_c(c) = \epsilon \rightarrow \lambda \epsilon.c \]
\[ D_c(c') = \emptyset \text{ if } c \neq c' \]
\[ D_c(p \cup q) = D_c(p) \cup D_c(q) \]
\[ D_c(p \rightarrow f) = D_c(p) \rightarrow f \]
\[ D_c(p \cdot q) = \begin{cases} 
D_c(p) \cdot q & \epsilon \not\in \mathcal{L}(p) \\
D_c(p) \cdot q \cup (\epsilon \to \lambda \epsilon. [p] (\epsilon)) \cdot D_c(q) & \text{otherwise.}
\end{cases} \]
Code
; Derivative of a parser combinator:
(define/memoize (DP c L)
  #:order ([l #:eq] [c #:equal])
  (match L
    [(empty)             (empty)]
    [(eps)               (empty)]
    [(token pred class) (if (pred c) (eps* (set c)) (empty))]
    [(orp L1 L2)                      (alt (DP c L1)
                                         (DP c L2))]
    [(seqp (and (nullp) l1) l2)       (cat (eps* (parse-null l1)) (DP c L2))]
    [(seqp (and (nullablep) l1) l2)   (alt (cat (eps* (parse-null l1)) (DP c L2))
                                         (cat (DP c l1) L2))]
    [(seqp l1 l2)                     (cat (DP c l1) L2)]
    [(repp l1)                        (cat (DP c l1) L)]
    [(redp l f)                       (red (DP c l) f)]))
(define/fix (parse-null l)
  #:bottom (set)
  (match l
    [(empty) (set)]
    [(eps* S) S]
    [(eps) (set l)]
    [(token _ _) (set)]
    [(repp (nullp)) (error "infinite parse-null")]
    [(repp _) (set '())]
    [(orp  l1 l2) (set-union (parse-null l1) (parse-null l2))]
    [(seqp l1 l2) (for*/set ([t1 (parse-null l1)]
                               [t2 (parse-null l2)])
                     (cons t1 t2))]
    [(redp l1 f) (for/set ([t (parse-null l1)]
                            (f t)))]))
Performance?
Atrocious.
Complexity?
$O(n2^n | G|)$
(define OneList (lang (or (seq OneList '1)
                           (eps* (set '())))))
or

or

empty

dead

seq

L

R

token

1

eps*

(set '())

eps*

(set 1)
Needs compaction.
p \cdot \emptyset = \emptyset
\[ p \cdot q \rightarrow f = q \rightarrow \lambda x. f(\lfloor p \rfloor(\epsilon), x) \]

if \{\epsilon\} = \mathcal{L}(p)
(define  (compact )
  (match l
    [(empty)  l]
    [(eps)    l]
    [(emptyp) (empty)]
    [(nullp)  (eps* (parse-null l))]
    [(token p c) l]

    [(orp (emptyp) l2)  (compact l2)]
    [(orp l1 (emptyp)) (compact l1)]

    [(seqp (nullp t) l2)  (red (compact l2) (lambda (w2) (cons t w2)))]
    [(seqp l1 (nullp t)) (red (compact l1) (lambda (w1) (cons w1 t)))]

    [(orp l1 l2)   (alt (compact l1) (compact l2))]
    [(seqp l1 l2)  (cat (compact l1) (compact l2))]

    [(redp (and e (nullp)) f)  
     (eps* (for/set ([t (parse-null e)]) (f t)))]

    [(redp (seqp (nullp t) l2) f)  
     (red (compact l2) (lambda (w2) (f (cons t w2)))]

    [(redp (redp l f) g)  
     (red (compact l) (lambda (w) (g (f w)))]

    [(redp l f)   (red (compact l) f)])}
(define/memoize (compact [l #:eq])
  (match l
    [(empty)       l]
    [(eps)         l]
    [(emptyp)      (empty)]
    [(nullp)       (eps* (parse-null l))]
    [(token p c)   l]
    [(orp (emptyp) l2)  (compact l2)]
    [(orp l1 (emptyp)) (compact l1)]
    [(seqp (nullp t) l2)  (red (compact l2) (lambda (w2) (cons t w2)))]
    [(seqp l1 (nullp t))  (red (compact l1) (lambda (w1) (cons w1 t)))]
    [(orp l1 l2)   (alt (compact l1) (compact l2))]
    [(seqq l1 l2)  (cat (compact l1) (compact l2))]
    [(redp (and e (nullp)) f)
      (eps* (for/set ([t (parse-null e)]] (f t)))]
    [(redp (seqp (nullp t) l2) f)
      (red (compact l2) (lambda (w2) (f (cons t w2)))]
    [(redp (redp l f) g)
      (red (compact l) (lambda (w) (g (f w)))]
    [(redp l f)   (red (compact l) f)]]
or

eps* (set '(() . 1))

seq
L R

token 1
or

seq
L R

token 1

eps* (set '((((1) . 1) . 1) . 1) . 1))
Average complexity?
$O(\frac{n}{|G|})?$
What we need

- Allow non-blocking I/O
- Act as library within the language
- Handle all CFGs: parse forest
- Draw on regular expressions
- Cacheable partial parses
- Be easy to implement
What we need

- Allow non-blocking I/O
- Act as library within the language
- Handle all CFGs: parse forest
- Draw on regular expressions
- Cacheable partial parses
- Be easy to implement
What we need

- Allow non-blocking I/O
- Act as library within the language
  - Handle all CFGs: parse forest
  - Draw on regular expressions
  - Cacheable partial parses
  - Be easy to implement
What we need

- Allow non-blocking I/O
- Act as library within the language
- Handle all CFGs: parse forest
  - Draw on regular expressions
  - Cacheable partial parses
  - Be easy to implement
What we need

- Allow non-blocking I/O
- Act as library within the language
- Handle all CFGs: parse forest
- Draw on regular expressions
  - Cacheable partial parses
  - Be easy to implement
What we need

- Allow non-blocking I/O
- Act as library within the language
- Handle all CFGs: parse forest
- Draw on regular expressions
- Cacheable partial parses
  - Be easy to implement
What we need

- Allow non-blocking I/O
- Act as library within the language
- Handle all CFGs: parse forest
- Draw on regular expressions
- Cacheable partial parses
- Be easy to implement
Would be nice

• Dynamic reconfigurability
• Beyond context-free languages
• Built into the language
• Support compositionality
Would be nice

- Dynamic reconfigurability
  - Beyond context-free languages
  - Built into the language
  - Support compositionality
Would be nice

- Dynamic reconfigurability
- Beyond context-free languages
  - Built into the language
  - Support compositionality
Would be nice

- Dynamic reconfigurability
- Beyond context-free languages
- Built into the language
  - Support compositionality
Would be nice

- Dynamic reconfigurability
- Beyond context-free languages
- Built into the language
- Support compositionality
Further reading

• Brzozowski. JACM 1964.
• Owens, Reppy, Turon. JFP 2010.
• Danielsson. ICFP 2010.
• Might, Darais. arXiv 2010.
Thank you!

Want more?

See blog.might.net