

COMPUTER SCIENCE 51

Spring 2009
cs51.seas.harvard.edu

Prof. Greg Morrisett
Prof. Ramin Zabih

Today

Check your email: do the survey!!!

Parsing

- how to describe syntax (BNF)
- how to implement a *recognizer* (later, a *parser*) from the syntax description.
- using lambda to generate code for you.

Some Motivation

Consider a service like Google search:

www.google.com/search?q=Morrisett

In essence, calling a search procedure with a parameter of "Morrisett".

www.google.com/search?q=Morrisett+Zabih

Now searching for pages with "Morrisett" and "Zabih".

More Google

Image search:

www.google.com/images?q=Greg

Return first 100 links instead of default:

www.google.com/search?q=Greg&num=100

Where the pages are in Chinese:

www.google.com/search?q=Greg&lr=lang_zh-CN

There are Many Options...

- Search Terms
 - conjunction: ?q=Greg+Morrisett
 - disjunction: &oq=Amy+John
 - unwanted terms: &eq=Tanya
 - Results per page: &num=100
 - Page Language: &lr=eng
 - File Type: &filetype=pdf
 - Restrict search to domain: &site=harvard.edu
 - ...
- Order doesn't matter.

What Has to Happen?

- Google gets URL as a request:
 - search?q=Zabih&num=100
- It must turn this string of characters into procedure calls to its search engine.
 - This is proprietary to Google
 - But for many web services, the URL gets converted into a SQL query for a database.
 - e.g., "SELECT * FROM table WHERE ..."
 - perform post-processing (e.g., get only 100)
 - print the results as HTML

Imagine...

- You have a "database" of papers.
 - title, authors, journal, date, etc.
 - perhaps represented as a list of structs.
- You have written a set of procedures:
 - add entries to the list
 - sort the list by some field
 - filter the list by some predicate
- And now you want to turn this into a web service like Google's search.
 - accept URL's as commands
 - perform commands specified by the URL
 - print any results as HTML

Recognition & Parsing

Taking a string of characters like:

"search?author=Morrisett"

or

"add?author=Zabih&year=2008&
title=Kittens&journal=J.of.Cute"

- recognize: checking that the string is legal
 - i.e., it is a proper command
- parse: figuring out what command the string represents
 - i.e., call filter and sort with appropriate arguments
 - i.e., create new struct, and add it to the list

First Task: Recognition

We can describe the *syntax* of commands using something like Backus-Naur Form (BNF):

```
com ::= ? add fieldarg+
      | ? search fieldarg+ sortarg?
fieldarg ::= & field = id+ | & year = number
sortarg ::= & sort = field | & sort = year
field ::= author | title | journal
id ::= [a-zA-Z]+
number ::= [1-9][0-9]*
```

Some abbreviations:

* is zero or more, + is one or more, ? is 0 or 1,
[0-9] = 0 | 1 | 2 | 3 | ... | 8 | 9

How Not to Write a Parser

```
::com ::= ? add fieldarg+ | ? search fieldarg+ sortarg?
```

```
(define (check_com cs)
  (if (or (empty? cs)
        (not (equal? (head cs) #\?)))
      (error ...)
      (cond [(empty? (cdr cs)) (error ...)]
            [(equal? (cadr cs) #\a)
             (cond [(empty? (caddr cs)) (error ...)]
                   [(equal? (caddr cs) #\d) ...]
                   [else (error ...)])]
            [(equal? (cadr cs) #\s)
             (cond [(empty? (caddr cs)) (error ...)]
                   [(equal? (caddr cs) #\e) ...]
                   [else (error ...)])])]))
```

Instead...

- Define a parser to be a function:
 - given a list of characters [C₁,C₂,C₃,...,C_n]
 - if the parser "matches" the first part of the list, then it returns the remaining part of the list as a result.
 - if the parser fails to match, then we'll return the number 0.
 - this is a hack!
- Provide ways to glue parsers together.
 - called "combinators" because they combine one function with another.

For Example:

:The parser that accepts the character c only.

```
(define (char c)
  (lambda (cs)
    (cond [(empty? cs) 0]
          [(equal? c (car cs)) (cdr cs)]
          [else 0])))
```

:If p1 accepts string s1, and p2 accepts string s2, then (cat p1 p2) accepts (append s1 s2)

```
(define (cat p1 p2)
  (lambda (cs)
    (let ([r (p1 cs)])
      (if (equal? r 0)
          (p2 r))))))
```

:A parser for the string "add"

```
(define add-parser
  (cat (char #\a) (cat (char #\d) (char #\d))))
```

Hmmm...

```

; We want parser for strings of characters
(define (str s) ???)

(define add-parser (str "add"))
(define search-parser (str "search"))

; We can simplify using the string->list function
; which explodes a string into a list of characters,
; and apply the char parser to each one of those characters
; to get a list of parsers.
(define (str s) (??? (map char (string->list s))))

; How can we combine the list of parsers to get a str parser?

```

Even Better:

```

; The parser that always accepts
; (or the parser that accepts an empty string of characters).
(define always (lambda (cs) cs))

; Given a list of parsers ps, returns the parser that accepts the
; concatenation of all of the strings that the parsers in ps accepts.
(define (cats ps) (foldr cat always ps))

; A parser that matches the string s.
(define (str s)
  (cats (map char (string->list s))))

; Parsers for the strings "add" and "search".
(define add-parser (str "add"))
(define search-parser (str "search"))

```

To See Better

```

(str "foo") =
(cats (map char (string->list "foo"))) =

(cats (map char (list #\f #\o #\o))) =

(cats (list (char #\f) (char #\o) (char #\o))) =

(foldr cat always
 (cons (char #\f)
       (cons (char #\o)
             (cons (char #\o) empty)))) =

(cat (char #\f)
     (cat (char #\o) (cat (char #\o) always)))

```

Disjunction

```

field ::= author | title | journal

; Tries parser p1, and if that fails, re-tries with parser p2.
(define (alt p1 p2)
  (lambda (cs)
    (let ([r (p1 cs)])
      (if (equal? r 0) (p2 cs) r))))

(define fields
  (alt (str "author")
       (alt (str "title") (str "journal"))))

```

Same Trick with Alt

```

; Tries parser p1, and if that fails, re-tries with parser p2.
(define (alt p1 p2)
  (lambda (cs)
    (let ([r (p1 cs)])
      (if (equal? r 0) (p2 cs) r))))

; The parser that always fails.
(define never (lambda (cs) 0))

; Combines a list of alternative parsers.
(define (alts ps) (foldr alt never ps))

(define fields
  (alts (map str (list "author" "title" "journal"))))

```

Expanding Out

```

(alts (map str (list "author" "title" "journal"))) =

(alts (list (str "author") (str "title")
           (str "journal"))) =

(foldr alt never
 (cons (str "author")
       (cons (str "title")
             (cons (str "journal") empty)))) =

(alt (str "author")
     (alt (str "title")
          (alt (str "journal") never)))

```

Further Expansion

```
(alt (str "author")
     (alt (str "title")
          (alt (str "journal") never))) =

(lambda (cs)
  (let ([a1 ((str "author") cs)])
    (if (not (equal? a1 0)) a1
        (let ([a2 ((str "title") cs)])
          (if (not (equal? a2 0)) a2
              (let ([a3 ((str "journal") cs)])
                (if (not (equal? a3 0)) a3
                    (never cs))))))))))
```

Moral

A careful use of design will let you write code that *generates* the code you need.

- infinitely more readable
- infinitely more maintainable
- e.g., trivial to add another field now

Key: finding combinators (cat, always, alt, never, etc.) that let you combine functionality in a flexible way.

- often an *algebraic* structure gives hints
- cat like multiplication, always like 1
- alt like addition, never like 0

Other Derived Parsers

```
(define (charset s)
  (alts (map char (string->list s))))

(define non-zero-digit (charset "123456789"))

(define digit (alt (char #\0) non-zero-digit))

(define lc-alpha
  (charset "abcdefghijklmnopqrstuvwxy"))

(define uc-alpha
  (charset "ABCDEFGHIJKLMNPQRSTUVWXYZ"))

(define alpha (alt lc-alpha uc-alpha))
```

Defining ?

```
(define (optional p) (alt always p))
```

Alas, this doesn't work. Consider:

```
(define myp (cats (list (str "Greg")
                        (optional (str "ory"))
                        (str " Morrisett"))))
```

This fails on the string "Gregory Morrisett".
 ((str "Greg") "Gregory Morrisett") returns "ory Morrisett"
 ((optional (str "ory")) "ory Morrisett") =
 ((alt always (str "ory")) "ory Morrisett")

The always matches, returning "ory Morrisett"!

But this doesn't match "Morrisett".

So We Must Be Careful

- We need to define optional as follows:

```
(define (optional p) (alt p always))
```

- Because p is tried first, it has a chance to succeed. Only if p fails do we decide to choose the always.
- In general, these combinators are sensitive to the order of alternation so we must take some care.
 - so the algebra is non-commutative
 - but cat was non-commutative too!

Kleene Star

- A similar issue arises with star:

```
(define (star p)
  (lambda (cs)
    ((alt (cat p (star p)) always) cs)))
```

- Implicitly: the *longest* match possible.
 - shortest match is always the empty string!
- Once we have star, plus is easy:

```
(define (plus p) (cat p (star p)))
```

So Let's Encode our Grammar

```
com ::= ? add fieldarg+
      | ? search fieldarg+ sortarg?
fieldarg ::= & field = id+ | & year = number
sortarg ::= & sort = field | & sort = year
field ::= author | title | journal
id ::= [a-zA-Z]+
number ::= [1-9][0-9]*
```

Encoding Our Grammar

```
::id ::= [a-zA-Z]+
(define id (plus alpha))

::number ::= [1-9][0-9]*
(define number (cat non-zero-digit (star digit)))

::sortarg ::= & sort = fields | & sort = year
(define sortarg
  (cat (str "&sort=") (alt fields (str "year"))))

::fieldarg ::= & field = id+ | & year = number
(define fieldarg
  (cat (char #\&)
    (alt (cats (list fields (char #\=) (plus id)))
      (cat (str "year=") number))))
```

Encoding continued

```
com ::= ? add fieldarg+
      | ? search fieldarg+ sortarg?

(define com
  (alt (cat (str "?add") (plus fieldarg))
    (cat (str "?search")
      (cat (plus fieldarg)
        (optional sortarg))))))
```

A Different Grammar

```
exp ::= number
      | exp + exp

(define exp
  (lambda (cs)
    ((alt number
      (cat exp (cat (char #\+) exp))))
    cs)))
```

Alas, doesn't work!
For "3 + 4" it will stop after the 3 when we want it to match the whole term.

Fix?

```
(define exp
  (lambda (cs)
    ((alt
      (cat exp (cat (char #\+) exp))
      number) cs)))
```

To match `exp`, we first try to match `exp + exp`.

To match `exp + exp`, we first try to match `exp + exp + exp`.

To match `exp + exp + exp`, we ...

No Left Recursion

- These parsing combinators do not support immediate *left recursion*.
 - a parsing combinator `X` defined in terms of itself that doesn't consume at least 1 character first is said to be left-recursive.

```
exp ::= number | exp + exp
```

- Ideas for fixing this?

Left Factoring

```
exp ::= number
      | exp + exp
```

```
exp ::=
  number (+ exp)*
```

Adding White Space

```
exp ::= num (plus exp)* ws
num ::= ws number
plus ::= ws +
```

```
white ::= #\space | #\tab | #\newline
ws ::= white*
```

General rule: add white space at the beginning of each "token" (and at the end of the whole grammar).

So Far...

- Defined parsers as functions from lists of characters to lists of characters (or zero).
 - the "rest" of the input is returned so that we can string parsers together via `cat`.
- Combinators let us combine parsers:
 - `cat`, `always`: sequencing (conjunction)
 - `alt`, `never`: alternation (disjunction)
 - but here, `alt` is not commutative!
 - `star`, `plus`, etc. can be derived using recursion
 - but care must be taken to avoid loops.
- Still, we can only *recognize* valid input.
- We want to *do* something with the input.

Recognize->Parse

Instead of just returning the list of characters that have yet to be consumed, we will also return a value.

For instance, parsing a number "425" should return the numeric value 425.

```
(char c) -- return the character c
(cat p1 p2) -- cons the values returned
(pmap f p) -- run the parser p, and apply f to the
             result that it returns.
```

Starting Over

```
; Parsers now return ans or fail values.
(define-struct ans (result unconsumed))
(define-struct fail ())

(define (char c)
  (lambda (cs)
    (cond [(empty? cs) (make-fail)]
          [(equal? (car cs) c)
           (make-ans c (cdr cs))]
          [else (make-fail)])))

(define never
  (lambda (cs) (make-fail)))
(define always
  (lambda (cs) (make-ans empty cs)))
```

These are the same...

```
(define (alt p1 p2)
  (lambda (cs) (let ([a1 (p1 cs)]
                    [a2 (if (fail? a1) (p2 cs) a1)])))

(define (optional p) (alt p always))

(define (alts ps) (foldr alt never ps))

(define (charset s)
  (alts (map char (string->list s))))

(define (star p)
  (lambda (cs) ((alt (cat p (star p)) always) cs)))

(define (plus p) (cat p (star p)))
```

New: pmap

```
; applies f to the result of p
(define (pmap f p)
  (lambda (cs)
    (let ([a (p cs)])
      (if (fail? a) a
          (make-ans (f (ans-result a))
                    (ans-unconsumed a)))))))
```

Parsing Digits into Numbers

```
; convert a digit to a corresponding numeric value
; char->integer returns the ASCII value of the character,
; so we subtract the ASCII value of "0" to get the right answer.
(define (digit->num c)
  (- (char->integer c) (char->integer #\0)))

; parse a digit, returning a numeric value
(define digit
  (pmap digit->num (charset "0123456789")))
```

Cat Combinator

```
; similar to before but cons-es the results
(define (cat p1 p2)
  (lambda (cs)
    (let ([a1 (p1 cs)])
      (if (fail? a1) a1
          (let* ([r1 (ans-result a1)]
                 [cs1 (ans-unconsumed a1)]
                 [a2 (p2 cs1)])
            (if (fail? a2) a2
                (make-ans (cons r1 (ans-result a2))
                          (ans-unconsumed a2))))))))))
```

Parsing IDs and Numbers

```
; parses 1 or more alphabetic characters, returning a string
(define id (pmap (list->string (plus alpha))))

; parses a list of identifiers
(define list-id (plus id))

; collapse (list 3 4 7) into the number 347
(define (num-list->num ds)
  (foldl (lambda (d a) (+ d (* 10 a))) 0 ds))

; parses 1 or more digits, returning the numeric value
(define number (pmap num-list->num (plus digit)))
```

A More Interesting Grammar

```
exp ::= number | ( exp )
      | exp + exp | exp - exp
      | exp * exp | exp / exp
```

Left-factoring the grammar yields:

```
exp ::= (number | ( exp )) (op exp)*
op ::= + | - | * | /
```

Adding White Space

```
exp ::= (num | lp exp rp) (op exp)* ws
num ::= ws number
lp ::= ws (
rp ::= ws )
op ::= ws (+ | - | * | /)
```

Translating into Scheme

```
exp ::= (num | lp exp rp ) (op exp)* ws
num ::= ws number
lp ::= ws (
rp ::= ws )
op ::= ws (+ | - | * | /)

(define num (cat ws number))
(define lp (cat ws (char #\() ))
(define rp (cat ws (char #\)) ))
(define op (cat ws (charset "+-*/")))
(define exp
  (lambda (cs)
    ((cat (alt num (cat lp (cat exp rp)))
          (cat (star (cat op exp)) ws)) cs)))
```

Getting Rid of WS Values

Recall that `cat` returns cons'es results.
We want to drop the white space from our results.

```
(define (catr p1 p2)
  (pmap cdr (cat p1 p2)))

(define num (catr ws number))
(define lp (catr ws (char #\() ))
(define rp (catr ws (char #\)) ))
(define op (catr ws (charset "+-*/")))
(define exp (catr ws (lambda (cs)
  ((cat (alt num (cat lp (cat exp rp)))
        (cat (star (cat op exp)) ws)) cs))))
```

Simplifying Exp's

```
(define exp
  (lambda (cs)
    ((cat (alt num (cat lp (cat exp rp)))
          (cat (star (cat op exp)) ws)) cs)))
```

We don't need the parens around nested exp's
and can drop the whitespace:

```
(define (catl p1 p2) (pmap car (cat p1 p2)))

(define exp
  (lambda (cs)
    ((cat (alt num (catr lp (catl exp rp)))
          (catl (star (cat op exp)) ws)) cs)))
```

Simplifying Further

```
(define (reduce pair)
  (if (empty? (cdr pair)) (car pair)
      (list (caadr pair) (car pair) (cdadr pair))))

(define exp
  (lambda (cs)
    (pmap reduce
      (cat (alt num (catr lp (catl exp rp)))
            (catl (star (cat op exp)) ws)) cs)))
```

Even Better

```
(define (const v p) (pmap (lambda (r) v) p))
(define pls (const + (char #\+)))
(define mns (const - (char #\-)))
(define tms (const * (char #\*)))
(define div (const / (char #\/)))
(define op (catr ws (alts (list pls mns tms div))))

(define (reduce pair)
  (if (empty? (cdr pair)) (car pair)
      ((caadr pair) (car pair) (cdadr pair))))

(define exp
  (lambda (cs)
    (pmap reduce
      (cat (alt num (catr lp (catl exp rp)))
            (catl (star (cat op exp)) ws)) cs)))
```

Precedence

Oops! This parser treats `*` and `/` as
having the same precedence as `+`
and `-`.

We have to re-examine the grammar:

```
exp ::= (number | ( exp )) (op exp)*
op ::= + | - | * | /
```

Long Story Short

```
exp ::= term (addop exp)*
term ::= factor (mulop exp)*
factor ::= number | ( exp )
addop ::= + | -
mulop ::= * | /
```

This slurps up as many factors,
separated by mulops as possible
before treating the result as a term.

How About Core Scheme?

```
sexp ::= number | op | ( exp* )
op ::= + | - | * | /
```

No need for left factoring, or worrying about
precedence!

So our translation to our combinators is
straightforward...

```
(define (sexp cs)
  ((alt num (alt op
    (catr lp (catl (star sexp) rp)))) cs))
```

More Realistically

```
sexp ::= atom |
  ( lambda ( id* ) sexp ) |
  ( if sexp sexp sexp ) | ...
  ( sexp* )
atom ::= id | number
sym ::= + | - | * | = | ? | < | > | ...
id ::=
  digit*
  (alpha | sym)
  (alpha | sym | digit)*
```