Programming languages

Parsing up strings with Python's string.find()

"Ada Lovelace".find(" ")
↑ returns the index of the first " " occurrence

"Alan Turing".find("n", 4)
↑ same, but starting from index 4

Selecting substrings

"Hello"[1:3] = el
0 1 2 3 4
↑ start here ↑ up to, but not including

"Hello"[1:] = ello
↑ go as far as possible

Split words by spaces

So common that Python has: string.split()

"Jane Eyre".split()
["Jane", "Eyre"] ← we get a list where each element corresponds one of the words in the string, separated by spaces (whitespace)

Regular expressions

We need more control over splitting strings

regular language ↙ ↘ concise notation

[1-3] → "1" or "2" or "3" (matches)

A module is a repository, or library of code
import brings in a module

import re ← for Python regular expression module

r"[0-9]" ← Python re look just like strings, but begin with r outside the double quotes

re.findall(r"[0-9]", "1+2=3") ← returns a list of all of the substrings that match the re
['1', '2', '3']

Concatenation

r"[a-c][1-2]" matches a1 a2 b1 b2 c1 c2

re.findall(r"[0-9][0-9]", "July 28, 1821")
['28', '18', '21']
↑ persion independence

re.findall(r"[0-9][0-9]", "12345")
['12', '34']

One or more: +

r"a+" matches 'a' 'aa' 'aaa' ...
r"[0-1]+" matches '0' '1' '00' '11' '01' '10' '1010' ...

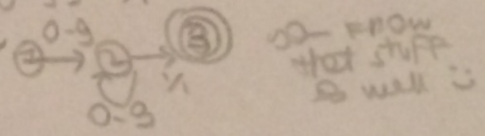
re.findall(r"[0-9]+", "13 from 1 in 1776")
✓ ['13', '1', '1776'] or
✗ ['1', '3', '1', '1', '7', '7', '6']

Maximal munch!
A re must consume the longest string it can and not a smaller part

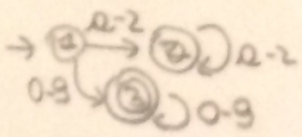
Finite state machines

"visual representation that shows what's going on behind the scenes"

r"[0-9]+½" 30½ 99½ 2½



Disjunctions



re = "[a-z]+|[0-9]+"

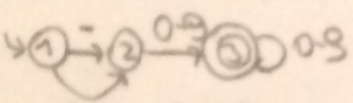
'disjunction' or 'OR'

re.findall(r"[a-z]+|[0-9]+", "goethe 1749")
['goethe', '1749']

[a-z] is really a|z|z

Options

a|b a|nothing



new RE: ? "optional" "the previous thing 0 or 1 times"

re.findall(r"-?[0-9]+", "1861-1941 R. Tagore")
['1861', '-1941']

Escape sequences

* zero or more times

a+ = aa*

So + * ? [ ] all mean something special

'P&P is Jane's book' ← escape the quote

escape sequence

same with RE: r"\+\+" Finds only '++'

Regex that matches lowercase words or single-hyphenated ~~words~~ lowercase words

r"[a-z]+(-?[a-z]+)*"

doesn't match 'a' or 'a-'

by making it r"[a-z]+(-[a-z]+)*"

now accepts '-a' which is wrong

r"[a-z]+(-[a-z]+)*" ← via next

↑ make that part optional

Quoted strings

Regular expressions for "quoted" "strings" are tricky

"I said, \"Hello.\" "

Find('"') will find "[said" and ""

New regexp . any character except newline

re.findall(r"[0-9].[0-9]", "1a1 222 a3")
['1a1', '222']

New regexp ^ inside [] means "Not" or "set complement"

re.findall(r"[0-9][^ab]", "1a1 222 a3")
['1 ', '22', '2 ']

Structure

(?:)

for the open paren.

(?:xyz)+ xyz xyzxyz xyzxyzxyz ...

re.findall(r"do+|re+|mi+)

+ binds to last character only

re.findall(r"(?:do|re|mi)+"

mimi ← rare midote ...

r"(?:[^\\]|(?:\\.))*"

"I say \"hello\" "

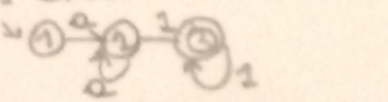
=

"When reading FSM, tracing with a finger" is basically what computers do for tracking "state we're in" in which

How to encode the FSM

dict → edges[(1,'a')] = 2 (current state, input, next state) ← tuple

Let's encode our FSM



edges = {(1,'a'):2, (2,'a'):2, (2,'1'):3, (3,'1'):3}

accepting = [3]

python's re module uses Fsmsim under the hood, something similar

Handling & and [redacted] ambiguity

Any number of groups of any non-empty size separated by one hyphen
Each group is [0-9]+

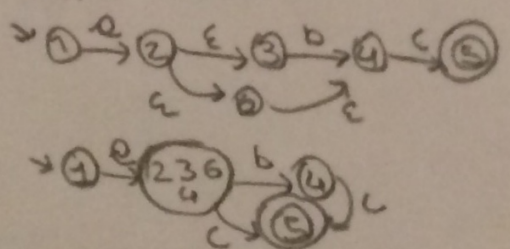
r"[0-9]*(?:-[0-9]+)*" accepts -6
r"(?:[0-9]-)*[0-9]+" worth!
r"([0-9]+(?:-[0-9]+)*" also works

Non-Deterministic FSM
They are easy-to-write FSMs with epsilon transitions and ambiguity

A FSM with no epsilon edges or ambiguity is a deterministic FSM

Every non deterministic FSM has a corresponding FSM
(NOT MORE POWERFUL!)

Build a deterministic FSM D where every state in D corresponds to a SET of states in the non-deterministic FSM



r"ab?c"

Lesson 2
Specify HTML + JavaScript
HTML tells a web browser how to display a page
tag tells how to interpret text

Lexical Analysis:
Break sentences down into component words to figure out what's going on

token, smallest unit of the output of lexical analysis
(words, numbers, punctuation, NOT: whitespace)

string → lexical analysis → list of TOKENS

LANGLE <
LANGLESLASH </
RANGLE >
EQUAL =
STRING "google.com"
WORD welcome!

TOKEN values
Has default is the string it matched, but we can transform it

```
def t_NUMBER(token):
    r"[0-9]+"
    TOKEN.value = int(TOKEN.value)
    return TOKEN
```

Ambiguities with NUMBER and WORD
Grace Hopper 1906
↗
matches regular expression for both WORD and NUMBER

TOKEN definition for WORD and STRING also overlap

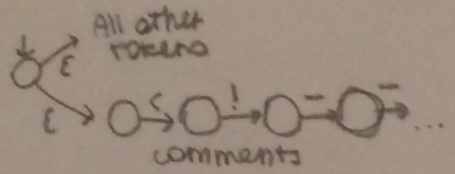
First rule wins!
=
string Sniffing
"quoted string" ↖
↳ remove quotes for token value

```
import ply.lex
t_ignore = ' '
```
implicitly ignore everything matching this regex

TO make a lexer
```
htmllexer = lex.lex()
htmllexer.input(webpage)
while True:
    tok = htmllexer.token()
```
↗ returns the next token that's available
```
    if not tok: break
    print tok
```

A lexer is a collection of token definitions

Comments
<!-- comment -->




```
states = (('htmlcomment', 'exclusive'),)
def t_htmlcomment(token):
    r'<!--'
    token.lexer.begin('htmlcomment')
def t_htmlcomment_end(token)
    r'-->'
    token.lexer.begin('INITIAL')
    # token.lexer.lineno += token.value.count('\n')
```

↑ count lines in comment and add those to the current line number

```
def t_htmlcomment_error(token):
    token.lexer.skip(1)
```

↳ that's a comment, do not match it

=

Everything within the <script> tag is javascript

```
<p>...<script type="text/javascript">
document.write("Hello world");
</script></p>
```

Identifier
variable name or function name

```
def t_IDENTIFIER(token):
    r'[a-zA-Z][a-zA-Z_]*'
    return token
```

- not allowed as first character

Numbers

```
def t_NUMBER(token):
    r'-?[0-9]+(?:\.[0-9]*)?'
    return float(token)
```

```
def t_ccomment(token):
    r'//[^\n]*'
    pass
```

comments in javascript

summary
tokens, like english can be broken up into words, languages can be broken into tokens.
Typically separated by spaces
[redacted] regular expressions to specify them

Noam Chomsky 1955
Lesson 3 [redacted] syntactic structures

utterances have rules (formal grammar)
grammatical sentence

Formal Grammar
Sentence → Subject Verb
Subject → Teacher
Subject → Students
Verb → Write
Verb → Think

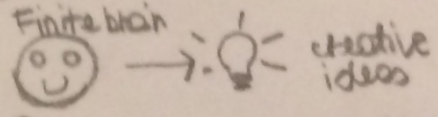
rewrite rules

derivation
Sentence → subject verb → students think

subj → subj and subj
↑ By adding this rule I have infinite derivations
subj → subj and subj ← recursion rewrite rule

...

Finite → Infinite
grammar utterances



↑
academics, including Chomsky, argue that this is one of the ways our Finite brain can produce infinite ideas

Lexical Analysis: "Lexing"
String → Token List
Syntactical Analysis "Parsing"
Token list → valid in grammar?

Lexing + Parsing = Expressive power!
=
stmt → identifier = exp
exp → exp + exp
exp → exp - exp
exp → number

a = 1 + 1
a = 3 - 3
...

Optional parts of Languages
"I think" vs "I think correctly"

sent → optAdj Subj verb
subj → William
optAdj → accurately
verb → shoots

Subj → Tell
optAdj → ε
Verb → lows

Grammars can encode Regular Languages

number → digit more_digits
more_digits → digit more_digits
more_digits → ε
digit → 0
...
digit → 9

Grammars have more expressive power than regular expressions
=
regular expressions describe regular languages
grammars describe context free languages
=
A → B can be replaced regardless of the context
yzAxy2 → yzBxy2

r'ab' → q → ab

r'a+' → q → ε
q → aq

r'a|b' → q → a
q → b

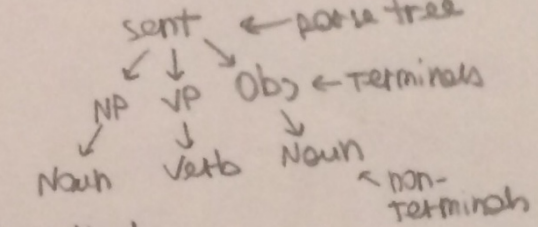
Balanced parenthesis are not regular!
P → (P)
P → ε

Formal Grammar ← HTML, Javascript

We're going to use formal grammars TO understand or describe HTML and Javascript

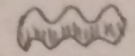
sent → NP VP Obj
NP → Noun
NP → Adj Noun
VP → Adj Verb
Obj → Noun
...

NP VP Obj
Was Beorn Romance of the three kingdoms



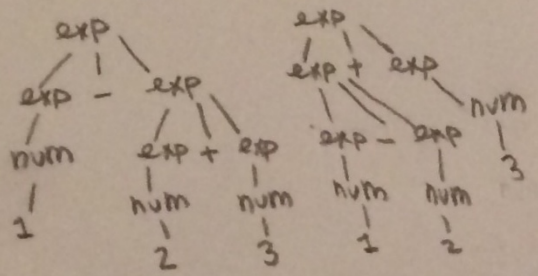
Ambiguity!
I saw Jane Austen using binoculars

1-2+3 means (1-2)+3
or 1-(2+3)



...

ooo

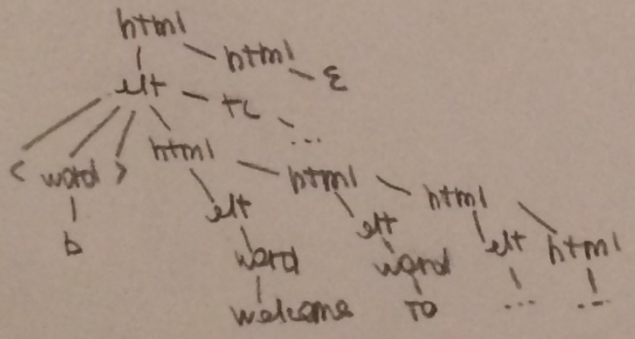


A grammar is ambiguous if there's at least one string in it, that has more than one parse tree

Grammars for html and javascript

html → element html
html → ε
element → word
element → tag-open html tag-close
tag-open → <word>
tag-close → </word>

 welcome to <i> my <li> webpage



Javascript grammar

stmt → identifier = exp
stmt → return exp
stmt → if exp compoundstmt
stmt → if exp compoundstmt
else compoundstmt

compoundstmt → {stmtS}
stmtS → stmt ; stmtS
stmtS → ε ...

exp → identifier
exp → numlit
exp → string
exp → exp + exp
exp → exp - exp
exp → exp * exp
exp → exp / exp
...

exp → exp < exp
exp → exp == exp
exp → exp && exp
exp → TRUE
exp → FALSE
...

js → element js
js → ε
element → function identifier (optparams) compoundstmt

element → stmt ;
optparams → params
optparams → ε

params → identifier, params
params → identifier

exp → ...
exp → identifier (optargs)
optargs → args

args → exp, args
args → exp
optargs → ε

Pretty much it, some flaws to be fixed later!

python lambdas

mystery = lambda (x): x+2
mystery(3) → 5
pete = mystery
pete(4) → 6

list power

def mySquare (x): return x**x
map(mySquare, [1,2,3,4,5])
→ [1,4,9,16,25]

map(lambda (x): x**x, [1,2,3,4,5])
↑ Anonymous Function

[x**x for x in [1,2,3,4,5]]
[len(x) for x in ["hello","my","friends"]]

list comprehension: computehypd a new list to be made of transformations on the elements in the old one
...

Python generators

```
[x+1 for x in (0,1,2)]
```

List comprehensions are great but we have to write down lists

```
def odds_only(numbers):
    for N in numbers:
        if N%2 == 1:
            yield N
```

← makes the function a generator!

```
[x for x in odds_only([1,2,3,4,5])]
[x for x in [1,2,3,4,5] if x%2 == 1]
                          ↑ guard
```

Checking valid strings

A → BC encoded as ("A", ["B","C"])

utterance "print exp ;" becomes ["print","exp",";"]

```
utterance = ["print","exp",";"]
["print","exp","_","exp",";"]
```

to perform that substitution:

```
pos = 1
result = utterance[0:pos] + rule[1] + utterance[pos+1:]
```

Programming Practice Fun

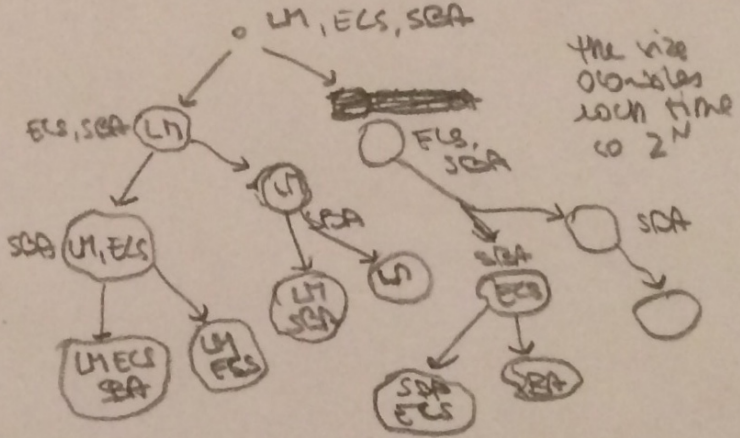
Given a list, print all sublists

```
["LM","ECS","SBA"]
→ []            ["LM","ECS"]
  ["LM"]        ["LM","SBA"]
  ["ECS"]       ["ECS","LM"]
  ["SBA"]       ["LM","ECS","SBA"]
```

"You're throwing a party. Who shows up?" → possibilities

For N entries, 2^N possibilities

How to solve it? Hint: recursion!




```
def sublists(big_list, selected_so_far):
    if big_list == []:
        print selected_so_far
    else:
        current_element = big_list[0]
        rest_of_big_list = [1:]
        sublists(rest_of_big_list, selected_so_far +
                 [current element])
        sublists(rest_of_big_list, selected_so_far)
```

Lesson 4

Last time's algorithm was brute force, and too expensive

The algorithm enumerates all strings in a grammar until a matching string is found. However that also runs forever if the string is not in the grammar!

Parsing idea...

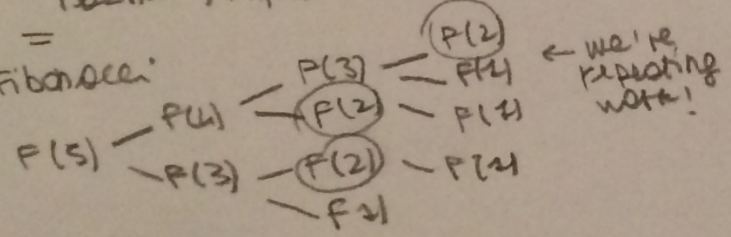
Doing Less! don't duplicate work

The virtue of Lazy

Larry Wall, in his textbook says the virtues of a programmer are: laziness, impatience, hubris

=

Fibonacci:



Memoization!

Solution: chart!

N	1	2	3	4	5	6
Fibo(N)	1	1	2	3	5	8

← if I need Fibo(6), I look the previous two Ns I already computed them

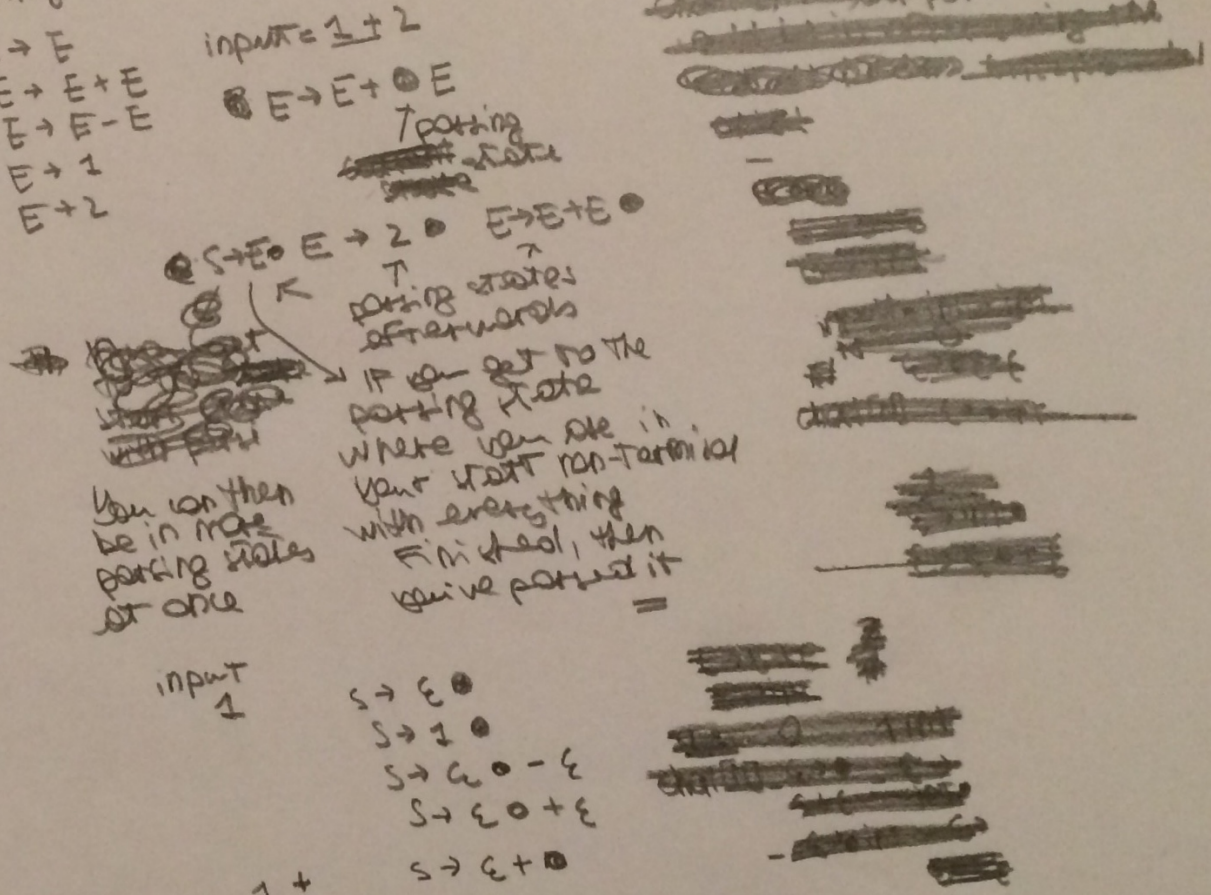
```
chart = {}
chart[5] = 5
chart[6] = 8

if 6 in chart
    print chart[6]

else
    ... ← compute Fibo(6)
```

Idea: Use memoization for parsing!

parsing state

S → E
E → E + E
E → E - E
E → 1
E → 2

input = 1 + 2



Memo with grammar

parse([t1, t2, t3 ... tN ... tlast])

chart[N] = all parse states we could be in after seeing t1, t2, t3 ... tN only!

E → E + E

E → int

input = int + int

N	0	1 int	2 int +
chart[N]	E → • E + E E → • int	E → int • E → E • + int	E → E + • E E → • int
	↑	1 (seen 0)	2 (seen 2)

Must add starting position of from position to our parse states.

PARSING is the inverse of producing strings

int + int
E + int
E + E
E

parsing

=

S → E
E → ...

starting at 0
S → E •
we've parsed it!

IF input is T tokens long

S → E • in chart[T], IF it is, our input is in the language

starting at 0

START → : → END

S → • E
chart[0]

(→ E •
chart[T]

Making intermediate entries!

S → E + • E From j in chart[i]
seen i tokens

Find all rules E → something in grammar

chart[i] has x → a b • c d From j

For all grammar rule

(→ p q r

we add

(→ • p q r From i

to chart[i]

Predicting or computing the closure!

=

Shifting, another way to help complete the parsing chart

There is yet another way, and we're going to need all three ways to produce a complete parsing chart!

x → a b • c d From j in chart[i]

c = non-terminal → compute closure

c = terminal → shift

→ x → a b c • d From j in chart[i+1]

if c is the ith input token!

Using re-write rules

x → a b • c d closure!

c d: non terminal

x → a b • c d: terminal shift!

x → a b • reduce!

a b c → reduced x c

=

E → E + E E → int

• int + int + int → int • + int + int

→ E • + int + int → E + • int + int

→ E + int • + int → E + E • + int

→ E • + int ... → E

...

• • •

seen B [redacted] tokens
coming from chart[B]

E → E+E• from B in [redacted]
chart[A]

input1 input2 ... inputB | E+E
→ input1 input2 ... inputB | E

suppose chart[B] has
E → E-•E From C

add E → [redacted]
E-E•
From C
TO chart[A]

=

T → •BC
B → bb input abbc

N	0	1 a	2 ab	3 abb
chart[N]	T → •BC From 0	T → a•BC From 0	B → b•b From 1	B → bb• From 1
		B → •bb From 1		* T → aB•C From 0, REDUCTION!

=

This chart based approach is due to Jay Earley!

```
def addToChart(chart, index, state):
    IF not (state in chart[index]):
        chart[index] = [state] + chart[index]
        return True

    else:
        return False
```

say we're looking at chart[i]
and [redacted] we see X → ab•cd
From j

We'll call

```
next_states = closure(grammar, i,
                      X, ab, cd)

For [redacted] next_state in next_states:
    any_changes = addToChart(chart, i,
                             next_state) or any_changes
```

```
def closure(grammar, i, X, ab, cd, j):
    next_states = [([redacted] rule[0], [], rule[1], i)
        for rule in grammar
            cd <> [] and rule[0] == cd[0]
    ]
    return next_states
```

We're currently looking at chart[i] and
we see X → ab•cd From j
The input is TOKENS

```
next_state = shift(tokens, i, X, ab, cd, j)
[redacted] IF next_state <> None:
    any_changes = add to chart(chart,
                  i+1, next_state) or anychanges
```

```
def shift(tokens, i, X, ab, cd, j):
    IF cd [redacted] <> [] and tokens[i] == cd[0]:
        return (X, ab + cd[0], cd[1:], j)
    else:
        return None
```

We're looking at chart[i] and we see x → ab • cd from j

We'll write

```
next_states = reductions(chart, i, x,
                         ab, cd, j)
for next_state in next_states:
    any_changes = add_to_chart(chart,
                  i, next_state) or
                  any_changes
```

```
def reductions(chart, i, x, ab, cd, j):
    return [(j, state[0], state[1]+[x],
             (j, state[2])[1:], j, state[3])
            for jstate in chart[j]
            if cd == [] and jstate[2] != []
               and (jstate[2])[0] == x]
```

[redacted]

Parse Trees

```
def p_exp_number(p):
    "exp : NUMBER"
    p[0] = ("number", p[1])
```

parse rule

under the hood, function! of object p (the parse tree)

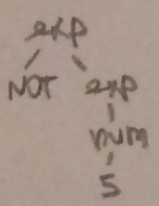
p[0] is exp
p[1] is NUMBER
I want exp to be a tuple with the word number and the value of the token NUMBER

exp → NUMBER

```
def p_exp_not(p):
    "exp : NOT exp"
    p[0] = ("not", p[2])
```

NOT 5 → ("not", ("number", 5))

encoding of the tree:




```
def p_elt_tag(p):
    "elt : LANGLE WORD tag_args RANGLE
          html LANGLESLASH"
    p[0] = ("tag-element", p[2], p[3],
            p[5], p[7])
```

→ doing-javascript

```
def p_exp_binop(p):
    """exp : exp PLUS exp
           | exp MINUS exp
           | exp TIMES exp"""
    p[0] = ("binop", p[1], p[2], p[3])
```

Ambiguity! 1-3-5

Associativity (1-3)-5 left associative 1-(3-5) right associative

Precedence 2+4+6 [redacted] 14 or 20?

```
precedence = (('left', 'PLUS', 'MINUS')
              ('left', 'TIMES', 'DIVIDE')
              )
```

↓ higher precedence

[redacted] NEXT interpreting languages, by walking over their parse trees!