

t1\_tree **=** nltk**.**Tree**.**fromstring(t1) t1\_tree**.**pretty\_print()

S-> NP VP, VP -> VBZ SBAR, SBAR -> IN S



Definition A tree domain X is a subset of N\* satisfying the following conditions.

If ak ∈ X and 0 ≤ i < k then ai ∈ X . (ii) If ai ∈ X then a ∈ X

SBAR

IN VP

that is on the second floor

-and CONJ

went to the classroom IN

from nltk.tree import Tree

t1 = Tree.fromstring("(S (NP I) (VP (V saw) (NP him)))")

Tree.draw(t1)

t1[1][1].label() 'NP'

t1.leaves() ['I', 'saw', 'him']

set(t1.treepositions()) {(), (0,), (0, 0), (1,)

, (1, 0), (1, 0, 0), (1, 1), (1, 1, 0)}

gpp = nltk.CFG.fromstring("""

... NP -> NP PP

... PP -> P NP

... NP -> 'covers' | 'doors' | 'books' | 'tables'

... NP -> 'titles' | 'carpets' | 'floors' | 'windows'

... P -> 'in' | 'by' | 'near' | 'on' | 'with' | 'under'

... """) <Grammar with 16 productions>

s1 = ['titles','on','books','under','tables']

parser = nltk.ChartParser(gpp)

gen1 = parser.parse(s1)

for t in gen1: print(t)

(NP

(NP (NP titles) (PP (P on) (NP books)))

(PP (P under) (NP tables)))

(NP

(NP titles)

(PP (P on) (NP (NP books) (PP (P under) (NP tables)))))

next(gen1) -> a graph

gpp.productions[1] P -> 'under'

gpp.productions()[-1].rhs() ('under',)

nltk.grammar.is\_terminal(gpp.producti

ons()[-1].rhs()[0]) True

showGrammar(g1)

i1 = "Maria criticizes Bill".split()

a0 = (i1,['S'])

>>(['Maria', 'criticizes', 'Bill'], ['S'])

b0 = tdstep(g1,a0)

>> expand S -> ['DP', 'VP']

[(['Maria', 'criticizes', 'Bill'], a['DP', 'VP'])]

b1 = tdstep(g1,a1)

>>expand DP -> ['D', 'NP']

expand DP -> ['NP']

expand DP -> ['Name']

expand DP -> ['Pronoun']

[(['Maria', 'criticizes', 'Bill'], ['D', 'NP', 'VP']),

(['Maria', 'criticizes', 'Bill'], ['NP', 'VP']),

(['Maria', 'criticizes', 'Bill'], ['Name', 'VP']),

(['Maria', 'criticizes', 'Bill'], ['Pronoun', 'VP'])]

a2 = b1[2]

b2 = tdstep(g1,a2)

from tdp import \*

parse(g1,['the','student','praises','the','beer'])

CNF has three conditions: 1. non-terminal rules are binary, 2. terminal rules are unitary, 3. if the langauge contains the empty string, then we allow S -> (empty string), where we consider S to be the start symbol.

g1 = CFG.fromstring("""

S -> VP NPs

NPo -> D N | 'ahau' | 'koutou'

NPs -> N D | 'ahau' | 'koutou'

VP -> V NPo | NEG VP

N -> 'dog'

D -> 'te'

NEG -> 'e'

V -> 'kite' | 'tutaki'

""")

parser = nltk.ChartParser(g1)

test1 = 'tutaki te dog koutou'.split()

trees1 = list(parser.parse(test1))

CFGs

feature grammars

**% start S**

**# Grammar Rules**

**S -> NP VP. S**[**/NP**](https://file+.vscode-resource.vscode-cdn.net/NP) **-> NP**[**/NP**](https://file+.vscode-resource.vscode-cdn.net/NP) **VP**

**S**[**/NP**](https://file+.vscode-resource.vscode-cdn.net/NP) **-> NP VP**[**/NP**](https://file+.vscode-resource.vscode-cdn.net/NP) **NP -> D N**

**VP -> TV NP. VP**[**/NP**](https://file+.vscode-resource.vscode-cdn.net/NP) **-> TV NP**[**/NP**](https://file+.vscode-resource.vscode-cdn.net/NP)

**CP -> 'that' S**[**/NP**](https://file+.vscode-resource.vscode-cdn.net/NP) **N -> N CP**

**# Lexical Rules. NP -> 'Angus'**

**NP -> 'Cyril'. D -> 'every'**

**D -> 'some'. N -> 'man'**

**N -> 'dog'. TV -> 'chases'**

**# Primitive slash symbol**

**NP**[**/NP**](https://file+.vscode-resource.vscode-cdn.net/NP) **->**

**# Grammar Productions**

**S[-INV] -> NP VP. S[-INV]/?x -> NP VP/?x**

**S[-INV] -> NP S/NP**

**S[-INV] -> Adv[+NEG] S[+INV]**

**S[+INV] -> V[+AUX] NP VP**

**S[+INV]/?x -> V[+AUX] NP VP/?x**

**SBar -> Comp S[-INV]**

**SBar/?x -> Comp S[-INV]/?x**

**VP -> V[SUBCAT=intrans, -AUX]**

**VP -> V[SUBCAT=trans, -AUX] NP**

**VP/?x -> V[SUBCAT=trans, -AUX] NP/?x**

**VP -> V[SUBCAT=clause, -AUX] SBar**

**VP/?x -> V[SUBCAT=clause, -AUX] SBar/?x**

**VP -> V[+AUX] VP**

**VP/?x -> V[+AUX] VP/?x**

**# Lexical Productions**

**V[SUBCAT=intrans, -AUX] -> 'walk' | 'sing'**

**V[SUBCAT=trans, -AUX] -> 'see' | 'like'**

**V[SUBCAT=clause, -AUX] -> 'say' | 'claim'**

**V[+AUX] -> 'do' | 'can'**

**NP[-WH] -> 'you' | 'cats'**

**NP[+WH] -> 'who'**

**Adv[+NEG] -> 'rarely' | 'never'**

**NP/NP ->**

**Comp -> 'that'**

N[SEM=<?P | ?Q>] -> N[SEM=?P] 'or' N[SEM=?Q]

3.1: This logical expression captures the "or" conjunction in English, where you can combine two full noun phrases together with the word "or". As an operator, "or" allows our function to be true whenever either (or both) noun phrases satisfy the semantic restrictions of the verb; "exclusive or" would require additional semantic conditions on the left-hand side of the rule, namely something like ?P != ?Q. An example where the expression is true would be: "The skyscrapers or the huts are tall." An example where the expression is false would be: "The skyscrapers or the huts are flying.".

No TA or student is a new professor

Det[SEM=<\P Q. none n.(P(n) -> Q(n))>] -> 'no'

base = r'''

... S[SEM=<?subj(?vp)>] -> DP[SEM=?subj] VP[SEM=?vp]

... N[SEM=<?P | ?Q>] -> N[SEM=?P] 'or' N[SEM=?Q]

... VP[SEM=?Q] -> 'is' NP[SEM=?Q]

... DP[SEM=<?X(?P)>] -> Det[SEM=?X] N[SEM=?P]

... NP[SEM=?Q] -> 'a' N[SEM=?Q]

... Det[SEM=<\P Q. no n.(P(n) -> Q(n))>] -> 'no'

# New terminal productions #

... N[SEM=<\x.ta(x)>] -> 'TA'

... N[SEM=<\x.student(x)>] -> 'student'

... N[SEM=<\x.professor(x)>] -> 'professor'

... JJ[SEM=<\x.new(x)>] -> 'new'

# New non-terminal productions

... N[SEM=<\x. ?P(x) & ?Q(x)] -> JJ[SEM=?P] N[SEM=?Q]

'''

1. Combine the N with the JJ: `N[SEM=<new(x) & professor(x)> -> JJ[SEM=<\x. new(x)>] N[SEM=<\x. professor(x)>]`

2. Add the Det: `NP[SEM=<new(x) & professor(x)> -> a N[SEM=<new(x) & professor(x)>]`

Mathematically, a context free grammar is a tuple

$N,\Sigma,P,S$, where

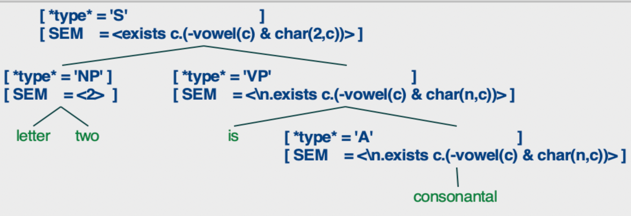
$N$ is a finite set (called the non-terminal symbols),

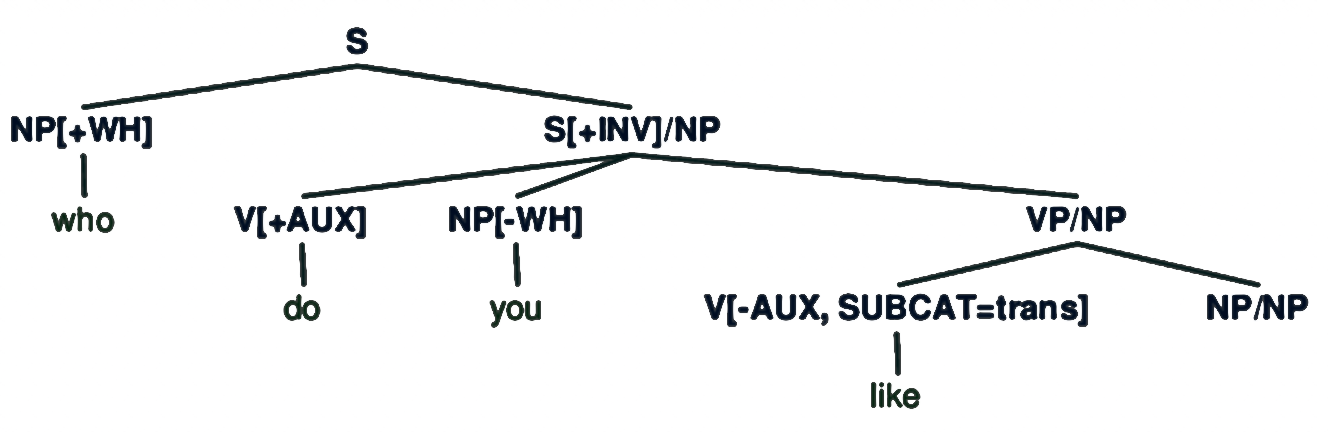
$\Sigma$ is a set disjoint from $N$ (called the terminal symbols),

$P$ is a finite subset of $N \times (P \cup \Sigma)\*$ (called the

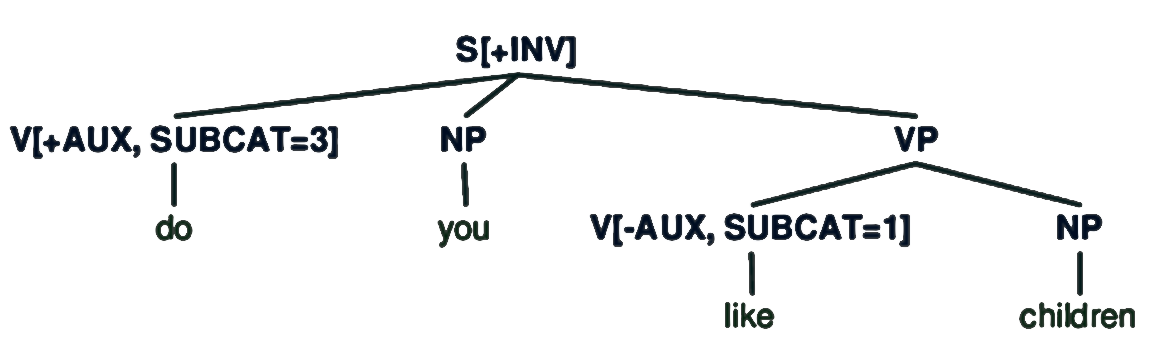
productions or rules), and

$S$ is an element of $N$ (called the start symbol).





S[+INV] -> V[+AUX] NP VP



**Syncope**

Syncope **=** hfst**.**regex(" V-> 0 || ?\* V C \_ C V n",definitions**=**defs)

**O2U**

o2u = hfst.regex(" o -> u || ?\* \_ C o [C|0] V n",definitions=defs)

**end vowel silence**

silence = hfst.regex(" V -> h || ?\* \_ V n",definitions=defs)

**nasal change**

nasal\_ng = hfst.regex(" n -> ŋ || ?\* V \_ i S V n",definitions=defs)

nasal\_m = hfst.regex(" n -> m || ?\* V \_ a C V n",definitions=defs)

**metathesis**

meta = hfst.regex(" [t p] -> [p t], [l b]-> [b l],[ n m ] -> [m n]|| ?\* V \_ V n",definitions=defs)

TagalogPHON = hfst.regex('nang .o. nam .o. ou .o. Syncope .o. silence .o. meta', definitions=defs)

Tagalog = hfst.regex('TagalogPHRASE .o. TagalogMOR .o. TagalogPHON', definitions=defs)

*# P2 is the language of all strings that contain at least three 'a's or 'b's*

*# ^ means repeat n times*

*# \* means zero or more of anything*

*# a|b means a or b*

*# () means the union of the language and the empty string*

P2 **=** hfst**.**regex('([a|b]^3\*)')

*# on the phonetic side so English.l*

defs **=** {'English':English}

expr **=** '[English .o. [K ?\* K]].u'

P4 **=** hfst**.**regex(expr, definitions**=**defs)

sample\_input(P4)

'klap|pe|rich'

defs **=** {"N":Nasals,"V":Vowels, 'English':English}

expr **=** '[English .o. [[N|V]\*]].u'

P5 **=** hfst**.**regex(expr, definitions**=**defs)

sample\_input(P5)

'oney', 'noaa'

Define a machine P6 for the phonetic forms of two-syllable words that have the main stress coming first. Use definitions from phoneclass.fst for the stress classes.

*# expr = '[English .o. [[C\* V0 C\* V1 C\*] | [C\* V0 C\* V2 C\*]| [C\* V1 C\* V2 C\*] ]]'*

expr **=** '[English .o. [[C\* V1 C\* V0 C\*] | [C\* V2 C\* V0 C\*]| [C\* V2 C\* V1 C\*] ]]'

P6 **=** hfst**.**regex(expr, definitions**=**defs6)

sample\_input(P6)

'but|try'

Infl\_str = "MasPst : l | [[FemPst : 0] 0:l 0:a]"

Infl = hfst.regex(Infl\_str)

Infl.view()

PHRASE = Vstem.copy()

PHRASE.input\_project()

# PHRASE.view()

separator = hfst.regex('"+":0')

separator\_up = separator.copy()

separator\_up.input\_project()

Infl\_up = Infl.copy()

Infl\_up.input\_project()

PHRASE.concatenate(separator\_up)

PHRASE.concatenate(Infl\_up)

PHRASE.view()

sample\_output(PHRASE)

'CRAWL+FemPst'

M = Vstem.copy()

M.disjunct(separator)

M.disjunct(Infl)

MOR = M.copy()

MOR.repeat\_plus()

x = PHRASE.copy()

x.compose(MOR)

sample\_input(x)

‘CARRY+MasPst'

sample\_output(x)

'sekl', 'beregla'

LDrop = hfst.regex("l -> 0 || b | č | d | k | g | l | m | n | p | r | s | t | t' | v | z \_ .#.")

DSD = hfst.regex("[t | d | t'] -> 0 || \_ l;")

Devoi = hfst.regex("b -> p, d -> t, g -> k, z -> s || \_ .#.")

apply\_rules('skrebl',[DSD,LDrop,Devoi])

Russian = PHRASE.copy()

Russian.compose(MOR)

Russian.compose(PHON)

LDrop = hfst.regex("l -> 0 || C \_ .#.", definitions=defs)

Russian.minimize()

PHON2 = hfst.regex('DSD .o. LDrop .o. Devoi', definitions=defs)

PHON2.minimize()

Russian2 = hfst.regex('PHRASE .o. MOR .o. PHON2', definitions=defs)

∥R .o. S∥ = {<x,z> | for some y, <x,y>∈ ∥R∥ and <y,z> ∈∥S∥ }

some man that Angus chases chases Irene

exists x.(man(x) & chase(angus,x) & chase(x,irene))

gagr.productions()[0]

S[] -> NP[NUM=?n] VP[NUM=?n]

gagr.productions()[0].rhs()[0][tp]

'NP'

fs1 = nltk.FeatStruct(TENSE='past', NUM='sg')

print(fs1)

[ NUM = 'sg' ]

[ TENSE = 'past' ]

gslash.productions()[1]

S[-INV][/](https://file+.vscode-resource.vscode-cdn.net/)?x[] -> NP[] VP[][/](https://file+.vscode-resource.vscode-cdn.net/)?x[]

VP[][/](https://file+.vscode-resource.vscode-cdn.net/)?x[] -> V[-AUX, SUBCAT='trans'] NP[][/](https://file+.vscode-resource.vscode-cdn.net/)?x[]

A[SEM=<\n.exists c.(consonant(c) & char(n,c))>] -> 'consonantal'

TV[SEM=<\m n.le(n,m)>] -> 'precedes'

N[SEM=<\n.exists c.(vowel(c) & char(n,c))>] -> 'vowel'

A[SEM=<\n.exists c.(capital(n) & char(n,c))>] -> 'capitalized'

Det[SEM=<\P Q.all n.(P(n) -> -Q(n))>] -> 'no'

N[SEM=<\n.exists c.(glide(c) & char(n,c))>] -> 'glide'

**def** nullable(g):

set **=** {p**.**lhs() **for** p **in** g**.**productions() **if** p**.**rhs() **==** ()}

*# repeat*

**while**(**True**):

initial\_len **=** len(set)

**for** p **in** g**.**productions():

flag **=** **True**

*# print(p)*

**for** r **in** p**.**rhs():

*# print(r)*

**if**(r **not** **in** set):

*# print(f"not in list: {r}")*

flag **=** **False**

**break**

**if**(flag): set**.**add(p**.**lhs())

*# until no new elements are added to the set*

**if** len(set) **==** initial\_len:

**break**

**return** set

defs = {}

roots = '''[

[BRAIN .x. {breyn}] |

[FLESH .x. {flesm}] |

[HEAD .x. {hedn}] |

[HEADS .x. {heds}] |

[EYE .x. {ebm}] |

[EYES .x. {eby}]

]'''

ROOT = hfst.regex(roots, definitions=defs)

defs['ROOT'] = ROOT

suffixes = '''[

[EAT .x. {e}] |

[ATE .x. {i}]

]'''

SUFFIX = hfst.regex(suffixes, definitions=defs)

defs['SUFFIX'] = SUFFIX

# ANSWER FOR RULE 1 (METATHESIS):

expr1 = '[y n] -> [n y], [s m] -> [m s], [d n] -> [n d], [d s] -> [s d], [b m] -> [m b], [b y] -> [y b] || \_ e'

rule1 = hfst.regex(expr1, definitions=defs)

defs['rule1'] = rule1

# ANSWER FOR RULE 2 (DE-NASALIZATION):

expr2 = '[n -> d || \_ i] .o. [m -> b || \_ i]'

rule2 = hfst.regex(expr2, definitions=defs)

defs['rule2'] = rule2

# ANSWER FOR RULE 3 (DUPLICATE DELETION):

expr3 = '[d d] -> d, [b b] -> b || \_ i'

rule3 = hfst.regex(expr3, definitions=defs)

defs['rule3'] = rule3

def apply\_rules(u,rs):

m = hfst.regex(" ".join([x for x in u]))

print(list(m.extract\_paths(max\_cycles=3).keys())[0])

for r in rs:

m.compose(r)

m.output\_project()

m.minimize()

print(list(m.extract\_paths(max\_cycles=3).keys())[0])

apply\_rules('breyne', [rule1, rule2, rule3])