Chapter 1 Top-down CF recognition

After introducing various substitution tests as structural probes, Fromkin's [3, p.175] introduction to linguistics builds a grammar like this:[1]

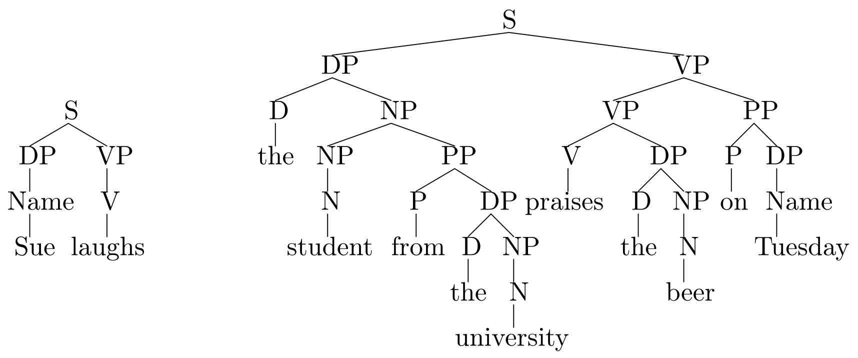
	basic rules for selected elements		rules for modifiers
(137)	S → DP VP		
(124a)	$\text{DP} \rightarrow \begin{Bmatrix} \text{(D) NP} \\ \text{Name} \\ \text{Pronoun} \end{Bmatrix}$		
(123)	NP → N (PP)		
(130)	$\text{VP} \rightarrow \text{V (DP) (}\begin{Bmatrix} \text{PP} \\ \text{CP} \\ \text{VP} \end{Bmatrix}\text{)}$		
(120)	PP → P (DP)		
(122)	AP → A (PP)		
(138)	CP → C S		
	AdvP → Adv		
		(73c.iii)	NP → AP NP
			NP → NP PP
			NP → NP CP
			VP → AdvP VP
			VP → VP PP
			AP → AdvP AP
	$\alpha \rightarrow \alpha$ Coord α		(for α=D,V,N,A,P,C,Adv,VP,NP,DP,AP,PP,AdvP,S,CP)

The numbering in parentheses comes from the Fromkin text. Recall that parentheses in the rules indicate optionality and braces mean 'choose exactly one'. The lexical items are given in a different format, but the same format could be used:

$\text{D} \rightarrow \begin{Bmatrix} \text{the} \\ \text{a} \\ \text{some} \\ \text{every} \\ \text{one} \\ \text{two} \end{Bmatrix}$	$\text{A} \rightarrow \begin{Bmatrix} \text{gentle} \\ \text{clear} \\ \text{honest} \\ \text{compassionate} \\ \text{brave} \\ \text{kind} \end{Bmatrix}$	$\text{N} \rightarrow \begin{Bmatrix} \text{student} \\ \text{teacher} \\ \text{city} \\ \text{university} \\ \text{beer} \\ \text{wine} \end{Bmatrix}$	$\text{V} \rightarrow \begin{Bmatrix} \text{laughs} \\ \text{cries} \\ \text{praises} \\ \text{criticizes} \\ \text{says} \\ \text{knows} \end{Bmatrix}$	$\text{Adv} \rightarrow \begin{Bmatrix} \text{happily} \\ \text{sadly} \\ \text{impartially} \\ \text{generously} \end{Bmatrix}$
$\text{Name} \rightarrow \begin{Bmatrix} \text{Bill} \\ \text{Sue} \\ \text{José} \\ \text{Maria} \\ \text{Presidents Day} \\ \text{Tuesday} \end{Bmatrix}$	$\text{Pronoun} \rightarrow \begin{Bmatrix} \text{he} \\ \text{she} \\ \text{it} \\ \text{her} \\ \text{him} \end{Bmatrix}$	$\text{P} \rightarrow \begin{Bmatrix} \text{in} \\ \text{on} \\ \text{with} \\ \text{by} \\ \text{to} \\ \text{from} \end{Bmatrix}$	$\text{C} \rightarrow \begin{Bmatrix} \text{that} \\ \epsilon \\ \text{whether} \end{Bmatrix}$	$\text{Coord} \rightarrow \begin{Bmatrix} \text{and} \\ \text{or} \\ \text{but} \end{Bmatrix}$

This grammar defines an infinite set of derivations like these:

[1]A grammar like this is a standard first step in linguistics texts. We find similar grammars in [4, pp.33-4] and [6, p.189-192] and [2, p.94] and [7, p.124].



This is a *context free grammar* (CFG) since on the left side of each rule we have just a category, which can 'expand to' (or 'be built up from') any of the sequences on the right side of the rules in the grammar. This particular grammar has many defects, besides just being radically incomplete:

- There are many regularities in English that this grammar does not enforce, like subject-verb agreement, case requirements on pronouns, etc.
- There are regularities in the grammar which appear non-accidental: e.g. that VPs have Vs in them, PPs have Ps in them. That is, the fundamental properties of phrases often seem to be (largely) determined by one of their elements, the 'head'.

Nevertheless, this grammar is a good starting point for us. We will get to better grammars later.

If a derivation roughly like the one shown above is computed when you hear that sentence in ordinary fluent speech, how could that happen?

Q. What algorithms can map orthographic or phonetic representations of sentences to their structures?

This is a first, rough indication of the main question for the class. We will see that the question is not stated precisely enough, and will formulate it more carefully on page 39 in §2, below.

1.1 Top-down backtrack CF recognition

The way we recognize a sentence as grammatical, as one that is allowed by the grammar, is to find a derivation of the sentence. This step, *recognition* is often separated from the formulation of a structural representation which is called *parsing*. A parser is usually just a recognizer that keeps a record of the steps used in the successful derivation, but still it is useful to think just about the recognizer first.

So the recognition problem is: given a grammar and a string of words, return True if the string has a derivation and False otherwise.

One way to proceed is to begin with the sentence category S (or whatever it is) and rewrite the leftmost elements until we get to a word, at which point we can check that word against the input, and so on. So, given the grammar and sentence derived above, we begin, step 0, by predicting S, and then expand the S as our first step:

step	predicted	input
0.	S	Sue laughs
1.	DP VP	Sue laughs

At this point we get stuck, though, since there are several ways to expand the DP. A simple strategy for handling this problem – the standard top-down, backtracking strategy – is to take all of the next steps, put them in a list, and then work on one of the possibilities. If that possibility does not work out, then we will try one of the other possibilities. So for the next step we actually take 5 steps (listed here in the order they appear in the grammar above):

step	predicted	input
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
2e.	DP Coord DP VP	Sue laughs

At this point, we choose one of 2a-2e, for example the first one 2a, and expand the leftmost category, and again we have choices:

step	predicted	input
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
2e.	DP Coord DP VP	Sue laughs
3aa.	D Coord D NP VP	Sue laughs
3ab.	the NP VP	Sue laughs
3ac.	a NP VP	Sue laughs
3ad.	some NP VP	Sue laughs
3ae.	every NP VP	Sue laughs
3af.	one NP VP	Sue laughs
3ag.	two NP VP	Sue laughs

Now we could expand 3aa in all possible ways:

step	predicted	input
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
2e.	DP Coord DP VP	Sue laughs
3aa.	D Coord D NP VP	Sue laughs
3ab.	the NP VP	Sue laughs
3ac.	a NP VP	Sue laughs
3ad.	some NP VP	Sue laughs
3ae.	every NP VP	Sue laughs
3af.	one NP VP	Sue laughs
3ag.	two NP VP	Sue laughs
4aaa.	D Coord D Coord D NP VP	Sue laughs
4aab.	the Coord D NP VP	Sue laughs
4aab.	a Coord D NP VP	Sue laughs
4aab.	some Coord D NP VP	Sue laughs
4aab.	every Coord D NP VP	Sue laughs
4aab.	one Coord D NP VP	Sue laughs
4aab.	two Coord D NP VP	Sue laughs

Now it is clear we are in trouble... this procedure will never terminate!

The problem here is a famous one: it comes from 'left recursion' through the category D. Let's say that a **category X is recursive** iff in one or more steps we can derive something else containing X (we use the superscript + to indicate one or more steps ⇒):[2]

$$\text{(recursion)} \quad X \Rightarrow^{+} \ldots X \ldots$$

We call this kind of recursion **left recursion** iff the category X can contain another category X as its first element (equivalently, on a leftmost branch):

$$\text{(left recursion)} \quad X \Rightarrow^{+} X \ldots$$

Similarly for **right recursion**:

$$\text{(right recursion)} \quad X \Rightarrow^{+} \ldots X$$

In the grammar above, we see for example that

[2]This terminology is standard in this kind of context [1, p.153], but there are other important senses of 'recursive'. A function definition that calls itself is recursive. And a language (a set of strings) is often said to be recursive iff it is Turing decidable [5, 8].

Coord	is not recursive
D	is both left and right recursive because of coordination
NP	is left recursive in coordination and in PP modification

Left recursion is a tricky problem, so for the moment, let's just remove all left recursion from the grammar, as follows:

	basic rules for selected elements		rules for modifiers
(137)	S → DP VP		
(124a)	$\text{DP} \rightarrow \left\{\begin{array}{c}\text{(D) NP}\\ \text{Name}\\ \text{Pronoun}\end{array}\right\}$		
(123)	NP → N (PP)		
(130)	$\text{VP} \rightarrow \text{V (DP) } (\left\{\begin{array}{c}\text{PP}\\ \text{CP}\\ \text{VP}\end{array}\right\})$		
(120)	PP → P (DP)		
(122)	AP → A (PP)		
(138)	CP → C S		
	AdvP → Adv		
		(73c.iii)	NP → AP NP
			~~NP → NP PP~~
			~~NP → NP CP~~
			VP → AdvP VP
			~~VP → VP PP~~
			AP → AdvP AP
	~~$\alpha \rightarrow \alpha$ Coord α~~	(for α=D,V,N,A,P,C,Adv,VP,NP,DP,AP,PP,AdvP,S,CP)	

So let's remove the left recursion from the example we started and continue from step 3 which now looks like this:

step	predicted	input
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3aa.	the NP VP	Sue laughs
3ab.	a NP VP	Sue laughs
3ac.	some NP VP	Sue laughs
3ad.	every NP VP	Sue laughs
3ae.	one NP VP	Sue laughs
3af.	two NP VP	Sue laughs

At this point the first symbol in step 3aa begins with a word, but it does not match the input, so we throw 4aaa away:

step	predicted	input
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3aa.	the NP VP	Sue laughs
3ab.	a NP VP	Sue laughs
3ac.	some NP VP	Sue laughs
3ad.	every NP VP	Sue laughs
3ae.	one NP VP	Sue laughs
3af.	two NP VP	Sue laughs

3ab is no good either, nor are any of our attempts at step 3, so we throw them all away and try to proceed from 2b:

step	predicted	input
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3ba.	N VP	Sue laughs
3bb.	N PP VP	Sue laughs

Clearly neither of these will work either, and so skipping some steps, we get to the point where we consider steps from 2c:

step	predicted	input
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3ca.	Bill VP	Sue laughs
3cb.	Sue VP	Sue laughs
3cc.	José VP	Sue laughs
3cd.	Maria VP	Sue laughs
3ce.	Presidents Day VP	Sue laughs
3cf.	Tuesday VP	Sue laughs

3ca is thrown out, but 3cc give us a match against our input. When we scan an element from the input (indicated by the prime mark), let's indicate the successful analysis in the input by crossing out the scanned element:

step	predicted	input
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3cb.	Sue VP	Sue laughs
3cc.	José VP	Sue laughs
3cd.	Maria VP	Sue laughs
3ce.	Presidents Day VP	Sue laughs
3cf.	Tuesday VP	Sue laughs
4cb'	VP	~~Sue~~ laughs

Continuing:

step	predicted	input
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3cb.	Sue VP	Sue laughs
3cc.	José VP	Sue laughs
3cd.	Maria VP	Sue laughs
3ce.	Presidents Day VP	Sue laughs
3cf.	Tuesday VP	Sue laughs
4cb'	VP	~~Sue~~ laughs
4cb'a	V	~~Sue~~ laughs
4cb'b	V DP	~~Sue~~ laughs
4cb'c	V PP	~~Sue~~ laughs
4cb'd	V CP	~~Sue~~ laughs
4cb'e	V VP	~~Sue~~ laughs
4cb'f	V DP PP	~~Sue~~ laughs
4cb'g	V DP CP	~~Sue~~ laughs
4cb'h	V DP VP	~~Sue~~ laughs

Taking 4cb'a first, we expand:

step	predicted	input
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3cb.	Sue VP	Sue laughs
3cc.	José VP	Sue laughs
3ce.	Presidents Day VP	Sue laughs
3ce.	Monday VP	Sue laughs
3cf.	Tuesday VP	Sue laughs
4cb'	VP	~~Sue~~ laughs
4cb'a	V	~~Sue~~ laughs
4cb'b	V DP	~~Sue~~ laughs
4cb'c	V PP	~~Sue~~ laughs
4cb'd	V CP	~~Sue~~ laughs
4cb'e	V VP	~~Sue~~ laughs
4cb'f	V DP PP	~~Sue~~ laughs
4cb'g	V DP CP	~~Sue~~ laughs
4cb'h	V DP VP	~~Sue~~ laughs
4cb'aa	laughs	~~Sue~~ laughs
4cb'aa	cries	~~Sue~~ laughs
4cb'aa	praises	~~Sue~~ laughs
4cb'aa	criticizes	~~Sue~~ laughs
4cb'aa	says	~~Sue~~ laughs
4cb'aa	knows	~~Sue~~ laughs

Now when we check 4cb'aa against the input, we fulfill the last prediction and also consume the last input symbol, which means we have found a derivation:

step	predicted	input
0.	S	Sue laughs
1.	DP VP	Sue laughs
2a.	D NP VP	Sue laughs
2b.	NP VP	Sue laughs
2c.	Name VP	Sue laughs
2d.	Pronoun VP	Sue laughs
3cb.	Sue VP	Sue laughs
3cc.	José VP	Sue laughs
3cd.	Maria VP	Sue laughs
3ce.	Presidents Day VP	Sue laughs
3cf.	Tuesday VP	Sue laughs
4cb'	VP	~~Sue~~ laughs
4cb'a	V	~~Sue~~ laughs
4cb'b	V DP	~~Sue~~ laughs
4cb'c	V PP	~~Sue~~ laughs
4cb'd	V CP	~~Sue~~ laughs
4cb'e	V VP	~~Sue~~ laughs
4cb'f	V DP PP	~~Sue~~ laughs
4cb'g	V DP CP	~~Sue~~ laughs
4cb'h	V DP VP	~~Sue~~ laughs
4cb'aa	laughs	~~Sue~~ laughs
4cb'aa	cries	~~Sue~~ laughs
4cb'aa	praises	~~Sue~~ laughs
4cb'aa	criticizes	~~Sue~~ laughs
4cb'aa	says	~~Sue~~ laughs
4cb'aa	knows	~~Sue~~ laughs
4cb'aa'	ϵ	~~Sue laughs~~

Success!

Even with this simple example, the procedure is tedious, so it is impractical to use reasonably sized grammars without a calculator. But first, let's describe more carefully what we did. The usual 'pseudocode' description is this something like this. First each analysis (each line in our calculation above) contains a sequence of predicted categories together with the remaining input – two lists. And the whole sequence of lines in our calculation is then a list of pairs of lists – the so-called 'backtrack stack'. There are just two basic parsing steps. The first one is called **expand** (which as we will say later means 'reduce complete'):

$$\text{(expand)}\ \frac{\text{input}, \text{X}\alpha}{\text{input}, \beta\alpha}\ \text{ if X} \rightarrow \beta$$

That is, when the predicted sequence begins with a predicted category X and we have a rule that says X rewrites as β, we can predict β followed by whatever else follows the X, leaving the input unchanged. The second rule is **scan** (which as we will say later means 'shift complete'):

$$\text{(scan)}\ \frac{\text{w input}, \text{w}\alpha}{\text{input}, \alpha}.$$

That is, when the predicted sequence begins with a word w and the input also begins with w, we can delete both of them (sometimes indicated informally by crossing them out). When either of these steps applies to α to produce β we write $\alpha \Rightarrow_{td} \beta$.

The following pseudocode uses the variable names chosen to be helpful:

ds	for the (typically incomplete) 'derivations'; ds is the 'backtrack stack'
cs	for the sequence of predicted categories in each derivation; cs is the 'stack'
i	for the remaining words of the input

Given a list of elements l=[3,5,7], the rightmost (or sometimes leftmost) element is called the top element. Then we can pop the top element off the list and use that element like this

$$\text{x} = \text{pop(l)}$$

This means that the last element of l is popped off and assigned to x. And the operation push(x,l)

means that l is extended with the top element x.

Then our first recognition algorithm is this one, for CFGs G in which no category is left-recursive and any (possibly empty) input i:

```
TOP-DOWN BACKTRACK CF RECOGNITION(G, i)
0  ds=[(i,S)] where S is the start category
1  while ds≠ [] and ds[0]≠([],[]):
2      ds=[d| ds[0] ⇒_td d] + ds[1:]
3  if ds==[] then False else True
```

The loop entered on line 1 is given simply by line 2: if there is a first derivation ds[0], compute the list of derivations [d| ds[0] $\Rightarrow_{td}$ d], that is, the list of derivations d such that d can be derived with one top-down step from ds[0], and then let ds be these new derivations together with any other derivations in the list ds[1:].

1.2 A naive python implementation

First we represent the grammar from page 11, but without any left recursive rules:

```
""" file: g1.py
    a grammar with no left recursion
"""
g1 = [('S',['DP','VP']),  # categorial rules
      ('DP',['D','NP']),
      ('DP',['NP']),
      ('DP',['Name']),
      ('DP',['Pronoun']),
      ('NP',['N']),
      ('NP',['N','PP']),
      ('VP',['V']),
      ('VP',['V','DP']),
      ('VP',['V','PP']),
      ('VP',['V','CP']),
      ('VP',['V','VP']),
      ('VP',['V','DP','PP']),
      ('VP',['V','DP','CP']),
      ('VP',['V','DP','VP']),
      ('PP',['P']),
      ('PP',['P','DP']),
      ('AP',['A']),
      ('AP',['A','PP']),
      ('CP',['C','S']),
      ('AdvP',['Adv']),
      ('NP',['AP','NP']),
      ('VP',['AdvP','VP']),
      ('AP',['AdvP','AP']),
      ('D',['the']), # now the lexical rules
      ('D',['a']),
      ('D',['some']),
      ('D',['every']),
      ('D',['one']),
      ('D',['two']),
      ('A',['gentle']),
      ('A',['clear']),
      ('A',['honest']),
      ('A',['compassionate']),
      ('A',['brave']),
      ('A',['kind']),
      ('N',['student']),
      ('N',['teacher']),
      ('N',['city']),
      ('N',['university']),
      ('N',['beer']),
      ('N',['wine']),
      ('V',['laughs']),
      ('V',['cries']),
      ('V',['praises']),
      ('V',['criticizes']),
      ('V',['says']),
      ('V',['knows']),
      ('Adv',['happily']),
      ('Adv',['sadly']),
      ('Adv',['impartially']),
      ('Adv',['generously']),
      ('Name',['Bill']),
      ('Name',['Sue']),
      ('Name',['Jose']),
      ('Name',['Maria']),
      ('Name',['Presidents','Day']),
      ('Name',['Tuesday']),
      ('Pronoun',['he']),
      ('Pronoun',['she']),
      ('Pronoun',['it']),
      ('Pronoun',['him']),
      ('Pronoun',['her']),
      ('P',['in']),
      ('P',['on']),
      ('P',['with']),
      ('P',['by']),
      ('P',['to']),
      ('P',['from']),
      ('C',['that']),
      ('C',[]),
      ('C',['whether']),
      ('Coord',['and']),
      ('Coord',['or']),
      ('Coord',['but'])]
```

Then we can implement the recognizer like this:

JTH: use the one posted on eLC instead, that one is updated for python3

```
""" file: td.py
    a simple top-down backtrack CF recognizer
"""
def showGrammar(g):  # pretty print grammar
    for (lhs,rhs) in g:
        print lhs,'->',
        for cat in rhs:
            print cat,
        print

def showDerivations(ds):  # pretty print the 'backtrack stack'
    for (n,(i,cs)) in enumerate(reversed(ds)):
        print n,'(',
        for w in i: # print each w in input
            print w,
        print ',',
        for c in cs: # print each predicted c in cs
            print c,
        print ')'
    print '---------'

def tdstep(g,(i,cs)): # compute all possible next steps from (i,cs)
    if len(cs)>0:
        cs1=cs[1:] # copy of predicted categories except cs[0]
        nextsteps=[]
        for (lhs,rhs) in g:
            if lhs == cs[0]:
                print 'expand',lhs,'->',rhs  # for trace
                nextsteps.append((i,rhs+cs1))
        if len(i)>0 and i[0] == cs[0]:
            print 'scan',i[0] # for trace
            nextsteps.append((i[1:],cs1))
        return nextsteps
    else:
        return []

def recognize(g,i):
    ds = [(i,['S'])]
    while ds != [] and ds[-1] != ([],[]):
        showDerivations(ds)  # for trace
        d = ds.pop()
        ds.extend(tdstep(g,d))
    if ds == []:
```

```
44          return False
45      else:
46          showDerivations(ds)  # for trace
47          return True
48
49  # Examples:
50  # recognize(g1,['Sue','laughs'])
51  # recognize(g1,['Bill','knows','that','Sue','laughs'])
52  # recognize(g1,['Sue','laughed'])
53  # recognize(g1,['the','student','from','the','university','praises','the','beer','on','Tuesday'])
54  # recognize(g1,['the','student','from','the','university','praises','the'])
55  # recognize(g1,['Sue','knows','that','Maria','laughs'])
```

Now in idle, we can load `g1.py` and then `td.py` using F5, or else we can type the load commands like this:

```
>>> from g1 import *
>>> from td import *
```

Now we are ready to try some examples – a few of these are given in comments at the bottom of `td.py`:

```
>>> recognize(g1,['Sue','laughs'])
0 ( Sue laughs , S )
---------
expand S -> ['DP', 'VP']
0 ( Sue laughs , DP VP )
---------
expand DP -> ['D', 'NP']
expand DP -> ['NP']
expand DP -> ['Name']
expand DP -> ['Pronoun']
0 ( Sue laughs , D NP VP )
1 ( Sue laughs , NP VP )
2 ( Sue laughs , Name VP )
3 ( Sue laughs , Pronoun VP )
---------
...
True
```

We omit many lines here, since the program does quite a lot of work on even this simple example.

This program correctly implements standard top-down backtrack parsing, but it has some problems! Listing some of them, from least to most serious:

P1. It is unnecessarily slow! Checking the 'innermost' loop:

a. In line 26 we see that finding the *expand* steps requires looping through the whole list of rules.

b. In line 27 we see an identity checks on the string names for categories. Since strings are sequences, this check is slower than, for example, checking the identity of two integers.

P2. It is necessarily slow! That is, even if we fix the problems in P1, the number of steps required to process an input of length n can be on the order of k^n for some $k > 1$.[3] To fix this problem, we need to somehow reduce or eliminate ambiguity that leads to backtracking. If only we had an oracle who could tell us what to do whenever there is a choice.[4]

P3. With left recursion, it can be non-terminating! This is just problem P2 again in its most extreme form. We need an oracle or something similar to help us avoid making the wrong choices repeatedly.

1.3 Exercises

Get one of our implementations of the top-down recognizer (`td.py`, or one of the others we looked at) and the grammar `g1.py`. Rename the grammar g⟨YOUR INITIALS⟩.py and then modify your version as follows:

1. Unfortunately, the grammar g1 accepts ** Sue laughs the student.* In the Fromkin text, this is excluded by the lexical entry for *laughs* which does not allow this verb to occur with a direct object. There, the verbs come with additional information about what they select: intransitives like *laughs* do not take DP complements, but transitives like *praises* do. Verbs like *knows* can select DP or CP complements, but verbs like *laugh* cannot. Fix the grammar in td0 to get all these things right, for the 6 verbs given.

[3]For proof see [1, p.299].

[4]We will build oracles later. And notice that with an artificial languages, we can simply modify the language to eliminate ambiguity!

2. If you did the previous step right, it should turn out that your new grammar will never use the rule `VP` → `V VP`. That rule was included in the Fromkin text for auxiliaries, and grammar g1 does not include any auxiliaries in its list of verbs.

   Add auxiliaries (using 3rd person present for the finite forms) and bare verbs (*laugh*) and participle forms (*laughed, laughing*), so that the recognizer accepts the examples the left, but not the ungrammatical forms on the right:

Sue will laugh	* Sue will laughs
Sue has laughed	* Sue has laughs
Sue is laughing	* Sue is laughed
Sue has been laughing	* Sue will been laughing
Sue will be laughing	* Sue has be laughing

Your changes should be as minimal and as linguistically reasonable as possible. Test your new grammar with the recognizer, and when it is working as required, turn it in via eLC along with a human-readable explanation.

References

[1] Aho, A. V., and Ullman, J. D. *The Theory of Parsing, Translation, and Compiling. Volume 1: Parsing.* Prentice-Hall, Englewood Cliffs, New Jersey, 1972.

[2] Carnie, A. *Syntax: A Generative Introduction (Second edition).* Blackwell, Oxford, 2006.

[3] Fromkin, V., Ed. *Linguistics: An Introduction to Linguistic Theory.* Blackwell, Oxford, 2000.

[4] Johnson, K. Introduction to transformational grammar. Tech. rep., University of Massachusetts, Amherst, 2007.

[5] Lewis, H. R., and Papadimitriou, C. H. *Elements of the Theory of Computation.* Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[6] O'Grady, W., Dobrovolsky, M., and Katamba, F. *Contemporary Linguistics: An Introduction (Third Edition).* Pearson Education, Essex, UK, 1996.

[7] Radford, A. *Transformational Grammar: A First Course.* Cambridge University Press, Cambridge, 1988.

[8] Rogers, H. *Theory of recursive functions and effective computability.* MIT Press, Cambridge, Massachusetts, 1967.