Chapter 2 Top-down CF parsing

2.1 TD backtrack parsing (complete derivations)

It is a simple matter to modify our recognizer to record a history of the steps taken in the derivation. We modify the pseudocode from page 18 as follows:

```
TOP-DOWN BACKTRACK CF PARSING($G, i$)
0   ds=[(i,S,[(i,S)])] where S is the start category
1   while ds$\neq$ [] and ds[0]$\neq$([],[]):
2           ds=[d| ds[0] $\Rightarrow_{tdh}$ d] + ds[1:]
3   if ds==[] then False else True
```

where the steps $\Rightarrow_{tdh}$ are defined as follows:

$$\text{(expand)}\ \frac{\text{input}, X\alpha, \text{h}}{\text{input}, \beta\alpha, \text{h(input}, \beta\alpha)}\ \text{if } X \to \beta \qquad \text{(scan)}\ \frac{\text{w input}, \text{w}\alpha, \text{h}}{\text{input}, \alpha, \text{h(input}, \alpha)}$$

The implementation is also an easy change from the `td.py`. Now that we are looking at the derivations, it is sometimes of interest to see more than one of them, and so we separate the main `derive` step from the `parse` function which now asks the user whether to look for another parse:

```python
""" file: tdh.py
    first parser: collect history
"""
def tdhstep(g,(i,cs,h)): # 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
                h1 = h[:] # copy of history
                h1.append((i,rhs+cs1))
                nextsteps.append((i,rhs+cs1,h1))
        if len(i)>0 and i[0] == cs[0]:
            print 'scan',i[0] # for trace
            i1=i[1:]
            h1 = h[:] # copy of history
            h1.append((i1,cs1))
            nextsteps.append((i1,cs1,h1))
        return nextsteps
    else:
        return []

def derive(g,ds):
    while ds != [] and not (ds[-1][0] == [] and ds[-1][1] == []):
        d = ds.pop()
        ds.extend(tdhstep(g,d))

def parse(g,i):
    ds = [(i,['S'],[(i,['S'])])]
    while ds != []:
        derive(g,ds)
        if ds == []:
            return 'False'
        else:
            d=ds.pop()
            for n,step in enumerate(d[2]):
                print n,step
            ans = raw_input('more? ')
            if len(ans)>0 and ans[0]=='n':
                return d[2]

# Examples:
# parse(g1,['Sue','laughs'])
# parse(g1,['the','student','laughs'])
# parse(g1,['the','student','praises','the','beer'])
# parse(g1,['Bill','knows','Sue','laughs'])
```

With this code, we get sessions like this one (we have removed many output lines):

```
>>> from g1 import *
>>> from tdh import *
>>> parse(g1,['Bill','knows','Sue','laughs'])
expand S -> ['DP', 'VP']
expand DP -> ['D', 'NP']
...
scan laughs
0 (['Bill', 'knows', 'Sue', 'laughs'], ['S'])
1 (['Bill', 'knows', 'Sue', 'laughs'], ['DP', 'VP'])
2 (['Bill', 'knows', 'Sue', 'laughs'], ['Name', 'VP'])
3 (['Bill', 'knows', 'Sue', 'laughs'], ['Bill', 'VP'])
4 (['knows', 'Sue', 'laughs'], ['VP'])
5 (['knows', 'Sue', 'laughs'], ['V', 'DP', 'VP'])
6 (['knows', 'Sue', 'laughs'], ['knows', 'DP', 'VP'])
7 (['Sue', 'laughs'], ['DP', 'VP'])
8 (['Sue', 'laughs'], ['Name', 'VP'])
9 (['Sue', 'laughs'], ['Sue', 'VP'])
10 (['laughs'], ['VP'])
11 (['laughs'], ['V'])
12 (['laughs'], ['laughs'])
13 ([], [])
more? y
expand NP -> ['N']
...
scan laughs
0 (['Bill', 'knows', 'Sue', 'laughs'], ['S'])
1 (['Bill', 'knows', 'Sue', 'laughs'], ['DP', 'VP'])
2 (['Bill', 'knows', 'Sue', 'laughs'], ['Name', 'VP'])
3 (['Bill', 'knows', 'Sue', 'laughs'], ['Bill', 'VP'])
4 (['knows', 'Sue', 'laughs'], ['VP'])
5 (['knows', 'Sue', 'laughs'], ['V', 'CP'])
6 (['knows', 'Sue', 'laughs'], ['knows', 'CP'])
7 (['Sue', 'laughs'], ['CP'])
8 (['Sue', 'laughs'], ['C', 'S'])
9 (['Sue', 'laughs'], ['S'])
10 (['Sue', 'laughs'], ['DP', 'VP'])
11 (['Sue', 'laughs'], ['Name', 'VP'])
12 (['Sue', 'laughs'], ['Sue', 'VP'])
13 (['laughs'], ['VP'])
14 (['laughs'], ['V'])
15 (['laughs'], ['laughs'])
16 ([], [])
more? y
expand NP -> ['N']
...
expand D -> ['two']
'False'
```

We see there are two derivations of this string! Two points to notice:

- Neither derivation is intended! That is, neither derivation corresponds to an analysis of the string which we expect competent speakers of English to formulate. We got these derivations because the grammar `g1.py` is not enforcing the selection requirements of the verbs.
- If you check, you will see that the number of steps in each derivation is exactly the number of nodes in the corresponding derivation tree! More on this later.

2.2 TD backtrack parsing (rules used)

We could get the whole derivation tree from the history, but it is more convenient to use a smaller representation: the list of rules used in the derivation, in order. A very minor change in the previous program achieves this:

```
TOP-DOWN BACKTRACK CF PARSING($G,i$)
0   ds=[(i,S,[])] where S is the start category
1   while ds≠ [] and ds[0]≠([],[]):
2           ds=[d| ds[0] $\Rightarrow_{tdp}$ d] + ds[1:]
3   if ds==[] then False else True
```

where the steps $\Rightarrow_{tdp}$ are defined as follows:

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

The implementation is easy:

```python
""" file: tdp.py  stabler@ucla.edu
    return the rules used in succsesful derivation
"""
def tdpstep(g,(i,cs,p)): # compute all possible next steps from (i,cs)
    if len(cs)>0:
        cs1=cs[1:] # copy of predicted categories except cs[0]
        p1 = p[:]  # copy of rewrites so far
        nextsteps=[]
        for (lhs,rhs) in g:
            if lhs == cs[0]:
                #print 'expand',lhs,'->',rhs  # for trace
                nextsteps.append((i,rhs+cs1,p1+[[lhs]+rhs]))
        if len(i)>0 and i[0] == cs[0]:
            #print 'scan',i[0] # for trace
            i1=i[1:]
            nextsteps.append((i1,cs1,p1))
        return nextsteps
    else:
        return []

def derive(g,ds):
    while ds != [] and not (ds[-1][0] == [] and ds[-1][1] == []):
        d = ds.pop()
        ds.extend(tdpstep(g,d))

def parse(g,i):
    ds = [(i,['S'],[])]
    while ds != []:
        derive(g,ds)
        if ds == []:
            return 'False'
        else:
            d=ds.pop()
            print 'll=',d[2]
            ans = raw_input('another? ')
            if len(ans)>0 and ans[0]=='n':
                return d[2]

# Examples:
# parse(g1,['Sue','laughs'])
# parse(g1,['the','student','laughs'])
# parse(g1,['the','student','praises','the','beer'])
# parse(g1,['Bill','knows','Sue','laughs'])
```

With this code, we get sessions like this one:

```
>>> from g1 import *
>>> from tdp import *
>>> parse(g1,['Sue','laughs'])
ll= [['S', 'DP', 'VP'], ['DP', 'Name'], ['Name', 'Sue'], ['VP', 'V'], ['V', 'laughs']]
another? n
[['S', 'DP', 'VP'], ['DP', 'Name'], ['Name', 'Sue'], ['VP', 'V'], ['V', 'laughs']]
>>>
```

2.3 Pretty print list format trees

A standard format for trees is to use a list where the first element is the root and the rest of the list is a list of its subtrees. So for example, [S] is the tree containing just one node. [S DP VP] is the tree with root S and subtrees DP VP. When trees get larger, the list notation is less easy to read, so we can "pretty print" these trees in a more readable form:

```
""" file: pptree.py  stabler@ucla.edu
    pretty print a tree given in list format
"""
def pptree(n,t): # pretty print t indented n spaces
    if isinstance(t,list) and len(t)>0:
        print n*' ', t[0] # print root
        for subtree in t[1:]:  # then subtrees indented by 4
            pptree(n+4,subtree)
    else:
        print '\n'+' '*n,t
""" example:
pptree(0,['TP', ['DP', ['John']], ['VP', ['V',['praises']], ['DP', ['Mary']]]])
pptree(0,[1, 2, [3, 4], [5, 6]])
"""
```

With this code, we get sessions like this one:

```
>>> from pptree import *
>>> t = ['S', ['DP', ['Name', 'Sue']], ['VP', ['V', 'laughs']]]
>>> pptree(0,t)
 S
     DP
         Name

             Sue
     VP
         V

             laughs
>>>
```

2.4 From rules to derivation trees in list format

Our parser returns the list of rules used in a successful derivation, listed in leftmost derivation order, sometimes called 'preorder' or LL order. Getting from the rules to derivation trees requires that we be able to recognize terminal productions. Here, we simply assume that the categories are disjoint from the nonterminals, and so if we are building a left branch, the terminal at the leaf will always be distinct from the category to be expanded next. With this convention, we do not need an independent way to tell whether a string is a category or terminal:

```
""" ll2dt.py
    convert LL list of rules to list-format tree
"""
def ll2dt(ll):
    print 'll=',ll
    if isinstance(ll,list):
        ll.reverse()
        return llr2dt(ll)
    else:
        return False

def llr2dt(llr):
    t=llr.pop()
    for i in range(1,len(t)):
        if len(llr)>0 and len(llr[-1])>0 and t[i]==llr[-1][0]:
            t[i]=llr2dt(llr)  # recursive def most natural, and not too deep
    return t
# example:
# ll=[['S', 'DP', 'VP'], ['DP', 'Name'], ['Name', 'Sue'], ['VP', 'V'], ['V', 'laughs']]
# ll2dt(ll)
# pptree(0,ll2dt(parse(g1noe,['Sue','laughs'])))
# pptree(0,ll2dt(parse(g1noe,['the','student','laughs'])))
```

With this code, we get sessions like this one:

```
>>> from g1 import *
>>> from tdp import *
>>> from ll2dt import *
>>> ll = parse(g1,['Sue','laughs'])
ll= [['S', 'DP', 'VP'], ['DP', 'Name'], ['Name', 'Sue'], ['VP', 'V'], ['V', 'laughs']]
another? n
>>> ll
[['S', 'DP', 'VP'], ['DP', 'Name'], ['Name', 'Sue'], ['VP', 'V'], ['V', 'laughs']]
>>> ll2dt(ll)
ll= [['S', 'DP', 'VP'], ['DP', 'Name'], ['Name', 'Sue'], ['VP', 'V'], ['V', 'laughs']]
['S', ['DP', ['Name', 'Sue']], ['VP', ['V', 'laughs']]]
>>>
```

2.5 From list format trees to NLTK trees

It is easy to convert our trees to the format used by the python NLTK library, and then we can use their graphical display utilities:

```
""" list2nltktree.py
    convert a list-format tree to an NLTK tree
"""
from nltk.util import in_idle
from nltk.tree import Tree
from nltk.draw import *

def list2nltktree(listtree):
    if isinstance(listtree,list):
        if listtree==[]:
            return []
        else:
            subtrees=[list2nltktree(e) for e in listtree[1:]]
            if subtrees == []:
                return listtree[0]
            else:
                return Tree(listtree[0],subtrees)
    else:
        return listtree

""" With an NLTK tree, we can use NLTK tree display:
t0 = ['S', ['DP', ['Name', 'Sue']], ['VP', ['V', 'laughs']]]
list2nltktree(t0).draw()
TreeView(t0)
TreeView(list2nltktree(ll2dt(parse(g1,['Sue','laughs']))))
"""
```

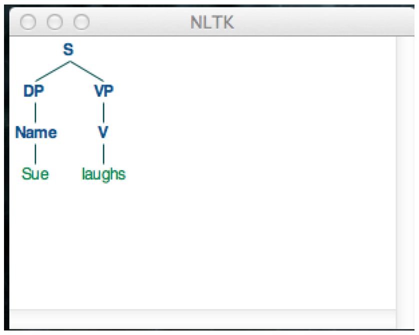
With this code, we get sessions like this one:

```
>>> from list2nltktree import *
>>> t = ['S', ['DP', ['Name', 'Sue']], ['VP', ['V', 'laughs']]]
>>> list2nltktree(t)
Tree('S', [Tree('DP', [Tree('Name', ['Sue'])]), Tree('VP', [Tree('V', ['laughs'])])])
>>>
```

The advantage of the nltk format is that we can then use the nltk graphical display tools. If we type

`list2nltktree(t).draw()`

We get a tree display like this:



2.6 Standard TD backtrack parsing

If you do not have NLTK:

```python
""" file: tdp_setup.py
    load grammar, td parser and tree utilities
"""
from g1 import *
from tdp import *
from ll2dt import *
from pptree import *

def eg0():
    t = ll2dt(parse(g1,['Sue','laughs']))
    pptree(0,t)

def eg1():  # nb: g1 allows 2 unintended parses for this one!
    t = ll2dt(parse(g1,['Bill','praises','the','student','on','Tuesday']))
    pptree(0,t)

def rpp(): # simple read-parse-print
    line = raw_input(': ')
    i = line.split()  # built-in python function, splits line at spaces
    t = ll2dt(parse(g1,i))
    pptree(0,t)
```

Now we can have sessions like this:

```
>>> rpp()
: the student praises the beer on Tuesday
ll= [['S', 'DP', 'VP'], ['DP', 'D', 'NP'], ['D', 'the'], ['NP', 'N'], ['N', 'student'], ['VP', 'V', 'DP', 'PP'], ['V',
another? n
ll= [['S', 'DP', 'VP'], ['DP', 'D', 'NP'], ['D', 'the'], ['NP', 'N'], ['N', 'student'], ['VP', 'V', 'DP', 'PP'], ['V',
 S
    DP
        D

            the
        NP
            N

                student
    VP
        V

            praises
        DP
            D

                the
            NP
                N

                    beer
        PP
            P

                on
            DP
                Name

                    Tuesday
>>>
```

If you have NLTK, you can do the same thing but with a nicer, graphical display of successful parses:

```python
""" file: tdp_setup2.py
    load grammar, td parser and NLTK tree utilities
"""
from g1 import *
from tdp import *
from ll2dt import *
from pptree import *
from nltk.util import in_idle
from nltk.tree import Tree
from nltk.draw import *
from list2nltktree import *

def eg0():
    t = ll2dt(parse(g1,['Sue','l[illegible]hs']))
    if isinstance(t,list):
        list2nltktree(t).draw()

def eg1():  # nb: g1 allows 2 unintended parses for this one!
    t = ll2dt(parse(g1,['Bill','praises','the','student','on','Tuesday']))
    list2nltktree(t).draw()

def rpp(): # simple read-parse-print
    line = raw_input(': ')
    i = line.split()
    t = ll2dt(parse(g1,i))
    if isinstance(t,list):
        list2nltktree(t).draw() # NLTK tree drawing function
```

2.7 Assessment: TD time and space requirements

We have seen that getting from a TD recognizer to a parser is fairly easy: we simply record the steps in the derivation, in a way which has no influence on the course of the derivation but only on the output. The parser makes it easier to explore properties of the recognizer, so let's return now to consider more carefully the basic properties of the top-down backtracking recognizer.

2.7.1 Soundness, completeness, and search

It is easy to see (and not difficult to prove conclusively) that the rules $\Rightarrow_{td}$ are sound and complete, for any context free grammar (CFG) in the following senses:

(soundness) For any CFG, if $(i, S) \Rightarrow^*_{td} (\epsilon, \epsilon)$ then $i \in L(G)$.

(completeness) For any CFG, if $i \in L(G)$, then $(i, S) \Rightarrow^*_{td} (\epsilon, \epsilon)$

On the other hand, the backtrack search has these undesirable properties:

(nontermination) For CFGs that have left recursion, the backtrack search can fail to terminate.

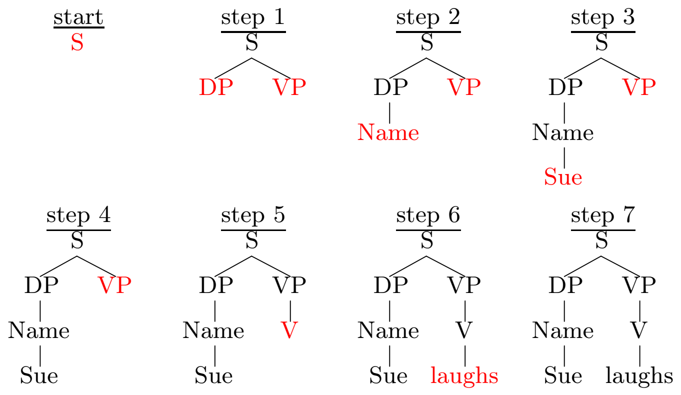
(inefficiency) Even for CFGs lacking left recursion, the number of steps required to recognize a string $i \in L(G)$ is not bounded by any polynomial function of the length of i.

These problems do not arise if the grammar has no left recursion and is deterministic, as discussed further in §2.7.7 below.

2.7.2 Predictiveness and incrementality

Although we have only printed out tree representations at the end of a parse, it is clear that we could have formulated the tree representations built at every step. When we do that, we see that the predicted categories at every step are, at every point, connected to the root. At every point, we have one, connected structure with its predicted categories in the recognizer's memory. This is a possible advantage for making sense of the fact that people normally understand what they hear on a word-by-word incremental basis [4, 35, 37].

Consider, for example, the seven step recognition of *Sue laughs* using our grammar g1. We can depict these steps by showing in red what is in the parser memory at each point:



At each point, the completed parts (shown in black) form connected, incrementally interpretable structures. With TD parsing, the completed structures always have this character, no matter what the grammar is. As we will see later, most other parsers do not have this property. Now that we have parsers, it is easier to see, as scientists or formal-language-theorists, what is going on in top-down recognition.

2.7.3 One parse at a time: garden pathing

It is often proposed that humans have difficulty with certain local ambiguities (or fail completely), resulting in the familiar "garden path" effects:[1] The following sentences exhibit extreme difficulty, but other less extreme variations in difficulty may also evidence the greater or less backtracking involved:

[1]There are many studies of garden path effects in human language understanding. Some of the prominent early studies are the following: [3, 12, 14, 11, 7, 30].

a. The horse raced past the barn fell
b. Horses raced past barns fall
c. The man who hunts ducks out on weekends
d. Fat people eat accumulates
e. The boat floated down the river sank
f. The dealer sold the forgery complained
g. Without her contributions would be impossible

In all of these cases, it seems that we initially are considering an analysis that must later be rejected, and sometimes it is difficult to recover from our mistake. This initially very plausible idea has not been easy to defend. One kind of problem is that some constructions which should involve backtracking are relatively easy: see for example [30, 13].

2.7.4 Right branching derivations, regularity, and space

Among the grammars that the TD recognizer can handle are some that define only regular languages. In particular, we know that if a grammar has only right branching (or only left branching), the language it defines is regular.[2] A regular language is one that can be recognized with only finite memory, so let's explore how much memory the TD recognizer needs for regular languages.

Our derivations divide the memory requirements into 3 parts:

(input buffer, recognizer memory, output buffer),

where the recognizer memory is what contains the predictions that must be remembered in order to complete the recognition correctly. The syntax acts only on contents of the recognizer memory: expanding an element using a rule of the grammar. Let's set aside the "input buffer" and the "output buffers" for a moment, and consider just the recognizer memory.

Since many grammars are non-deterministic, presenting choices in how to expand categories, we need another kind of memory too, the backtrack stack that remembers choices that have not yet been pursued. Let's set non-determinism aside for a moment too, returning to it in §2.7.7 below, and consider the recognizer memory.

First, consider how much memory is needed to parse grammars that have only simple right branching parses – where the left categories in binary productions are all lexical. That is, every rule has the one of the forms

$$\begin{array}{ll} A \to BC & \text{where } B \to w \text{ but not } B \to CD \\ A \to w & \text{for } w \text{ a vocabulary element} \\ A \to [] & \end{array}$$

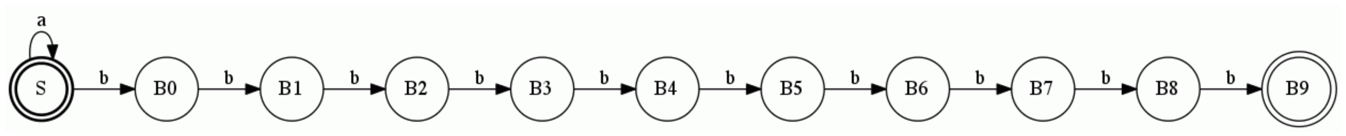
Grammars of this form can be regarded as finite state machines, where there is a transition from state A to C labeled with w iff there is a rule $A \to BC$ where $B \to w$; the start category of the grammar is the unique start state of the machine; and a state A is a final state iff there is a rule $A \to []$.

Consider the following grammar, for example:

```
""" file: g2.py
    a right branching grammar (no left recursion!)
"""
g2 = [('S',['A','S']),
      ('S',[]),
      ('S',['B','B0']),
      ('B0',['B','B1']),
      ('B1',['B','B2']),
      ('B2',['B','B3']),
      ('B3',['B','B4']),
      ('B4',['B','B5']),
      ('B5',['B','B6']),
      ('B6',['B','B7']),
      ('B7',['B','B8']),
      ('B8',['B','B9']),
      ('B9',[]),
      ('A',['a']),
      ('B',['b']),
      ]
```

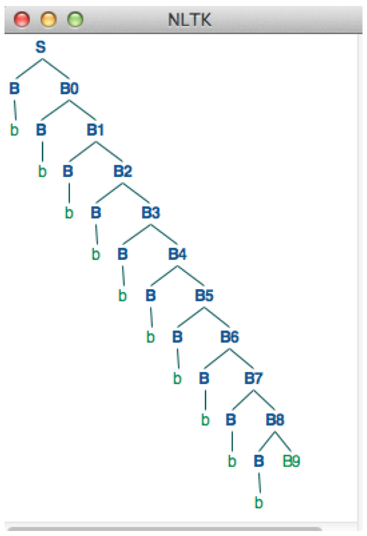
[2]In fact, a 'regular' or 'finite state' language is often defined to be one that can be generated by a right branching (or left branching) grammar. The equivalence between these grammars and 'finite state machines' is then an easy proof. See for example, [34, 36, 27, 18].

If we draw this grammar as a finite state machine it looks like this:



If we look at the derivation tree for b^{10}, we see that it has 31 nodes:

```
>>> from tdp_setup2 import *
>>> ll=parse(g2,[’b’,’b’,’b’,’b’,’b’,’b’,’b’,’b’,’b’,’b’])
ll= [[’S’, ’B’, ’B0’], [’B’, ’b’], [’B0’, ’B’, ’B1’], [’B’, ’b’], [’B1’, ’B’, ’B2’], [’B’, ’b’], [’B2’, ’B’, ’B3’], [’
another? n
>>> t=ll2dt(ll)
ll= [[’S’, ’B’, ’B0’], [’B’, ’b’], [’B0’, ’B’, ’B1’], [’B’, ’b’], [’B1’, ’B’, ’B2’], [’B’, ’b’], [’B2’, ’B’, ’B3’], [’
>>> list2nltktree(t).draw()
```



And so we know that our TD recognition will require 31 steps too:

```
>>> from g2 import *
>>> from tdh import *
>>> parse(g2,[’b’,’b’,’b’,’b’,’b’,’b’,’b’,’b’,’b’,’b’])
....
0 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’S’])
1 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B’, ’B0’])
2 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’b’, ’B0’])
3 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B0’])
4 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B’, ’B1’])
5 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’b’, ’B1’])
6 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B1’])
7 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B’, ’B2’])
8 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’b’, ’B2’])
9 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B2’])
10 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B’, ’B3’])
11 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’b’, ’B3’])
12 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B3’])
13 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B’, ’B4’])
14 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’b’, ’B4’])
15 ([’b’, ’b’, ’b’, ’b’, ’b’], [’B4’])
16 ([’b’, ’b’, ’b’, ’b’, ’b’], [’B’, ’B5’])
17 ([’b’, ’b’, ’b’, ’b’, ’b’], [’b’, ’B5’])
18 ([’b’, ’b’, ’b’, ’b’], [’B5’])
19 ([’b’, ’b’, ’b’, ’b’], [’B’, ’B6’])
20 ([’b’, ’b’, ’b’, ’b’], [’b’, ’B6’])
21 ([’b’, ’b’, ’b’], [’B6’])
22 ([’b’, ’b’, ’b’], [’B’, ’B7’])
23 ([’b’, ’b’, ’b’], [’b’, ’B7’])
24 ([’b’, ’b’], [’B7’])
25 ([’b’, ’b’], [’B’, ’B8’])
26 ([’b’, ’b’], [’b’, ’B8’])
```

```
27 (['b'], ['B8'])
28 (['b'], ['B', 'B9'])
29 (['b'], ['b', 'B9'])
30 ([], ['B9'])
31 ([], [])
more? n
```

It is easy to see that with simple right branching derivations like this one, the amount of space required in the stack of predicted categories is bounded by a finite constant. For this grammar, the bound is 2. We never need more space than that to remember what is required in the rest of the string. This makes sense, since the language is finite state.

2.7.5 Digression: Finite state models of human language

Evidence for non-finite-stateness of human languages. Let's write

HG for the grammars of human languages,
HL for the set of human languages, defined by the grammars HG,
Reg for the set of finite state (i.e. regular) grammars, and
L(Reg) for the set of regular languages, defined by the grammars Reg.

These two ideas are widely accepted:

a. **(HG$\not\in$Reg)** The syntax (i.e. the grammar) used by speakers of human languages is not finite state.

b. **(HL$\not\in$L(Reg))** The languages (i.e. the set of sentences generated by the grammars) used by speakers of human languages are not finite state.

It is quite possible to use a non-finite-state CFG to define a finite or finite state language. In fact, CFG definitions of finite and regular sets can be exponentially more succinct than the smallest regular, finite state grammar (Reg) definitions of the same languages [17]. Consequently, accepting (HG$\not\in$Reg) does not entail (HL$\not\in$L(Reg)). In the other direction though, we do have entailment: if (HL$\not\in$L(Reg)), then obviously (HG$\not\in$Reg).

a. **Evidence for (HL$\not\in$L(Reg)).** The evidence for this claim is usually given by taking some example languages, and arguing that they are not regular in a way that is naturally stated using the following important result:

(Nerode-Myhill theorem) Given any language L over alphabet Σ, for any string of words x, define the 'good finals' of x with respect to L:

$$\text{goodFinals}_L(x) = \{y|\ xy \in \text{L}\}.$$

And let's say $x \equiv y$ iff $\text{goodFinals}_L(x)$=$\text{goodFinals}_L(y)$, and for any string x let's write

$$[x]_L = \{y|\ x \equiv y\}.$$

Then L is regular iff the set of equivalence classes is finite:[3]

$$\{[x]_L|\ x \in \Sigma^*\} \text{ is a finite set.}$$

For English, the argument can be given by presenting an infinite sequence of sequences of words that all have different good finals, like this:

oysters
oysters oysters
oysters oysters oysters
...

[3]The Myhill-Nerode theorem is presented in Khoussainov&Nerode'01 [22, Thm 2.4.1]; in Salomaa'69 [33, Thm 5.4]; in Sakarovitch'09 [32, Props 3.11,3.12]; and in Hopcroft&Ullman'79 [19, §3.4] at the end of their second chapter on finite automata. The Myhill-Nerode theorem and Kleene's theorem are perhaps the most important results in formal language theory. Kleene's theorem says that the regular languages are the closure of the set $\{\emptyset\} \cup \{\{a\}|\ a \in \Sigma\}$ with respect to concatenation, union, and Kleene star. A nice, modern presentation of Kleene's theorem can be found in [32, Thm 2.1].

Now, the empirical claim is that, in the languages defined by the grammars of English speakers, no two elements of this sequence has the same good finals. For example, clearly *oysters eat* is a good sentence. And oysters are cannibalistic, so the sentence *oysters (that) oysters eat eat* is not only grammatical but sensible and true. Now the relevant facts for are claim are these:

> eat $\in$ goodFinals(oysters), but eat $\notin$ goodFinals(oysters oysters) or any other seq in the list
> eat eat $\notin$ goodFinals(oysters) or any other seq except goodFinals(oysters oysters)

and similarly for every other pair. The expressions oysters^{n}eat^n exhibit a kind of center-embedded dependency that cannot be defined with a regular grammar. This kind of argument is standard in the field (see for example [20]), but I do not regard it as persuasive by itself! It depends on judgements about center embeddings that become unacceptable already around depth 2! This argument is only persuasive when we add the consideration that HG$\notin$Reg, to which we turn now.

b. **Evidence for (HG$\notin$Reg).** In the 1960's various kinds of evidence were offered for this claim. The strongest kind of evidence comes from the idea that finite state grammars cannot define the kinds of constituents which are abundantly evidenced by semantic considerations (defining meaningful units), syntactic arguments (defining grammars in which discontinuous dependencies are recognized appropriately) and by various sorts of psycholinguistic evidence. For example, various studies of recalling substrings of sentences show that constituents are easier to remember than substrings that straddle constituent boundaries.[4] Miller'67 summarized the upshot of these studies, saying:

> ...constituent structure languages are more natural, easier to cope with, than regular languages...The hierarchical structure of strings generated by constituent-structure grammars is characteristic of much other behavior that is sequentially organized; it seems plausible that it would be easier for people than would the left-to-right organization characteristic of strings generated by regular grammars. [29]

Non-finite-state constituency is also evidenced by the perceptual effects demonstrated by the 'click' studies [24, 9], and many other phenomena.

Certain aspects of human languages appear to be finite state though:

- *The set of derivations defined by a CFG is a regular tree set (modulo renaming categories).* This was proven by Thatcher'67 [38]. We may return to this idea later, since we also have...
- *The set of derivations defined by a 'minimalist grammar' (MG) is a regular tree set (modulo renaming categories).* This idea follows directly from the results of Michaelis'98 [28]. Recently Kobele'11 and Graf'11 [23, 16] strengthened that insight by showing that the class of MG derivation trees is closed under intersection with regular tree languages.
- *Phonotactics is finite-state.* The SPE phonology of Chomsky&Halle'68 [5] used very powerful rewrite rules, but later analysis by Kaplan&Kay'94 [21] showed that finite state power suffices for almost everything, and this remains true in OT phonology [31, 2, 8] if reduplication is set aside.

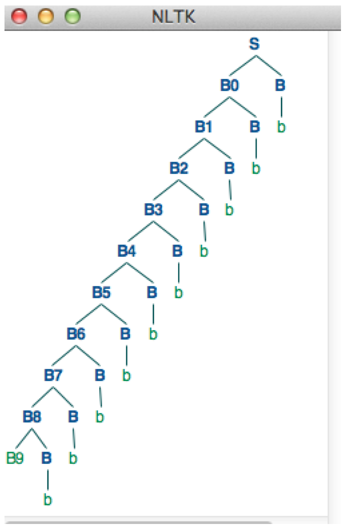
2.7.6 Left branching derivations, regularity, and space

Now let's consider this grammar, in which all branching is to the left:

```
""" file: g3.py
    a left branching grammar (no left recursion!)
"""
g3 = [('S',['B0','B']),
       ('B0',['B1','B']),
       ('B1',['B2','B']),
       ('B2',['B3','B']),
       ('B3',['B4','B']),
       ('B4',['B5','B']),
       ('B5',['B6','B']),
       ('B6',['B7','B']),
       ('B7',['B8','B']),
       ('B8',['B9','B']),
       ('B9',[]),
       ('B',['b'])]
```

[4]See e.g. Fodor, Bever and Garrett'74 [10] for a good review of this early work.

This grammar accepts b^{10} too, with this derivation:



If we look at the parser memory requirements for the 31 step derivation, though, we get a very different picture (running off the right edge of the page – but it should be clear what is happening!)

```
>>> from g3 import *
>>> from tdh import *
>>> parse(g3,[’b’,’b’,’b’,’b’,’b’,’b’,’b’,’b’,’b’,’b’])
expand S -> [’B0’, ’B’]
expand B0 -> [’B1’, ’B’]
...
0 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’S’])
1 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B0’, ’B’])
2 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B1’, ’B’, ’B’])
3 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B2’, ’B’, ’B’, ’B’])
4 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B3’, ’B’, ’B’, ’B’, ’B’])
5 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B4’, ’B’, ’B’, ’B’, ’B’, ’B’])
6 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B5’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’])
7 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B6’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’])
8 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B7’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’])
9 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B8’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’])
10 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B9’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’])
11 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’])
12 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’b’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’])
13 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’])
14 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’b’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’])
15 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’])
16 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’b’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’])
17 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’])
18 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’b’, ’B’, ’B’, ’B’, ’B’, ’B’, ’B’])
19 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’B’, ’B’, ’B’, ’B’, ’B’, ’B’])
20 ([’b’, ’b’, ’b’, ’b’, ’b’, ’b’], [’b’, ’B’, ’B’, ’B’, ’B’, ’B’])
21 ([’b’, ’b’, ’b’, ’b’, ’b’], [’B’, ’B’, ’B’, ’B’, ’B’])
22 ([’b’, ’b’, ’b’, ’b’, ’b’], [’b’, ’B’, ’B’, ’B’, ’B’])
23 ([’b’, ’b’, ’b’, ’b’], [’B’, ’B’, ’B’, ’B’])
24 ([’b’, ’b’, ’b’, ’b’], [’b’, ’B’, ’B’, ’B’])
25 ([’b’, ’b’, ’b’], [’B’, ’B’, ’B’])
26 ([’b’, ’b’, ’b’], [’b’, ’B’, ’B’])
27 ([’b’, ’b’], [’B’, ’B’])
28 ([’b’, ’b’], [’b’, ’B’])
29 ([’b’], [’B’])
30 ([’b’], [’b’])
31 ([], [])
```

Notice that the number of elements in the parser memory increases as we go down the left branch. The longer the left branch is, using any left branching grammar, the more parser memory required, without bound. So although left branching grammars define only finite state languages, the TD recognizer requires unbounded memory to handle them. Do humans have great difficulty with left-branching? It seems not [15].

2.7.7 Determinism: time, space, nontermination

As we mentioned in the previous chapter, the TD recognizer will fail to terminate if the grammar has left recursion, and even when it doesn't, the number of steps the parser takes can be on the order of k^n for some $k > 1$, as shown in [1, p.299]. This is because of nondeterminism! What can we do?

'Lookahead' can reduce local ambiguity, but English is not LL(k) for any k.

Consider the following simplistic grammar for sentences like *the dog [that I told you about] barks*:

$$\begin{aligned}
&\text{S} \to \text{DP VP}\\
&\text{DP} \to \text{D NP}\\
&\text{D} \to \text{the}\\
&\text{NP} \to \text{NP CP}\\
&\text{NP} \to \text{N}\\
&\text{N} \to \text{dog}\\
&\text{CP} \to \text{that I told you about}\\
&\text{VP} \to \text{barks}
\end{aligned}$$

A TD recognizer processing this example gets to the point:

(dog that I told you about barks, NP VP)

Here, by looking 2 words ahead, we see *that*, and so we might think that this lets us know that we should use the first NP rule:

(dog that I told you about barks, NP CP VP)

But here again, looking 2 words ahead, we still see *that*, and now it would be a mistake to use the first NP rule. Clearly, if there is a finite limit k on how many words ahead we can look, we will not be able to decide which rule to use. A language is said to be LL(k) if we can always decide which rule to use with k symbols of lookahead, so this and many other constructions show that English is not LL(k) for any k.

Quickly surveying some of the other cases of local ambiguity which could present problems:

Subject-verb agreement. In simple English clauses, the subject and verb agree, even though the subject and verb can be arbitrarily far apart:

a. The deer {are, is} in the field

b. The deer, the guide claims, {are, is} in the field

c. The deer who prance about are/*is in the field

d. The deer who prances about *are/is in the field

Bound pronoun agreement. The number of the embedded subject is unspecified in the following sentence:

a. I expect the deer to admire the pond.

But in similar sentences it can be specified by binding relations:

b. I expect the deer to admire {itself,themselves} in the reflections of the pond.

Head movement can move a head away from disambiguating context:

a. i. Have the students take the exam!

 ii. Have the students taken the exam?

b. i. Is the block sitting in the box?

 ii. Is the block sitting in the box red?

A' movement can create 'doubtful gaps', separated from disambiguating context:

a. who$_i$ did you expect t_i to make a potholder

b. who$_i$ did you$_j$ expect PRO$_j$ to make a potholder for t_i

English has intractable global ambiguity, like other human languages.

A context free grammar can have infinitely many derivations for one string. For example, this grammar has one string but infinitely many derivations:

$$S \to S \qquad S \to a.$$

How much structural ambiguity do sentences of human languages really have? We can get a first impression of how serious the structural ambiguity problem is by looking at simple artificial grammars for these constructions.

1. **PP attachment in $[_{VP}$ V D N PP1 PP2 ...].** Consider a grammar with these rules:

   $$\begin{aligned} &VP \to VP\ PP \\ &NP \to NP\ PP \end{aligned}$$

   If we calculate how many structures are allowed in strings when there are n PPs, we find:[5]

$n =$	1	2	3	4	5	6	7	8	9	10
#trees =	2	5	14	132	469	1430	4862	16796	1053686	...

2. **N compounds $[_N$ N N].** These can be generated by the rule

   $$N \to N\ N$$

   If we calculate how many structures are allowed in strings when there are n Ns, we find the same series as for PPs,

n	1	2	3	4	5	6	7	8	9	10
#trees	1	1	2	5	14	42	132	429	1420	4862

3. **Coordination $[_X$ X and X].** If coordination is always binary, the ambiguities are similar to the previous cases. To make things interesting, suppose we assume that in addition to binary coordination, English allows arbitrary lists of coordinates, with a rule like this:

   $$NP \to NP\ (\text{and}\ NP)^*$$

   Note that this is not a standard context-free rule. It is equivalent to a grammar with infinitely many rules:

   $$\begin{aligned} &NP \to NP \text{ and } NP \\ &NP \to NP \text{ and } NP \text{ and } NP \\ &NP \to NP \text{ and } NP \text{ and } NP \text{ and } NP \\ &\ldots \end{aligned}$$

   Then with n NPs, the number of structures allowed grows like this:

n	1	2	3	4	5	6	7	8	9	10
#trees	1	1	3	11	45	197	903	4279	20793	103049

Do these kinds of ambiguity-creating elements really crop up in substantial numbers? Consider the 9 PPs and other modifiers in this phrase that is engraved in stone at the UBC Museum of Anthropology:

> The government of the province of British Columbia has contributed to the building of this museum to honour the centenary of the province's entry into confederation in the conviction that it will serve and bring pleasure to the people of the nation.

Or consider the following example

> Enriched with minerals and vitamins, the purified soybean meal is colored, flavored, pressed, shaped and cut into bits that look and taste like bacon chips or strips, pork sausage, ground beef, sliced ham or chicken and are cheaper and just as nourishing as the real thing. (from the American Publishing House for the Blind (APHB) corpus) [25]

[5]This is the Catalan series $Cat(n) = \binom{2}{n} - \binom{2n}{n-1}$. Its growth is not bounded by any polynomial function of n. Classic discussions appear in [6, 26]. See also §6.2 in chapter 8 of the NLTK book on "pernicious ambiguity" -JTH

2.7.8 Implementing a timer, counting number of steps

To explore the number of steps a grammar takes, it is easy to add a ‘timer’, a count of the number of steps, to any of our previous programs. One way to do this in python is with a ‘global variable’, that is, a variable that scopes over the entire computation. Here we add a timer and print it out with lines 6,13,17,24,25,31:

```
1  """ file: tdt.py
2      a simple top-down backtrack CF recognizer,
3       with a step-counter (’time’, measured in steps)
4  """
5  def tdstep(g,(i,cs)): # compute all possible next steps from (i,cs)
6      global steps
7      if len(cs)>0:
8          cs1=cs[1:] # copy of predicted categories except cs[0]
9          nextsteps=[]
10         for (lhs,rhs) in g:
11             if lhs == cs[0]:
12                 #print ’expand’,lhs,’->’,rhs  # for trace
13                 steps = steps+1
14                 nextsteps.append((i,rhs+cs1))
15         if len(i)>0 and i[0] == cs[0]:
16             #print ’scan’,i[0] # for trace
17             steps = steps+1
18             nextsteps.append((i[1:],cs1))
19         return nextsteps
20     else:
21         return []
22
23 def recognize(g,i):
24     global steps
25     steps = 0
26     ds = [(i,[’S’])]
27     while ds != [] and ds[-1] != ([],[]):
28         #showDerivations(ds)  # for trace
29         d = ds.pop()
30         ds.extend(tdstep(g,d))
31     print ’steps=’,steps
32     if ds == []:
33         return False
34     else:
35         #showDerivations(ds)  # for trace
36         return True
37
38 # Examples:
39 # recognize(g1,[’Sue’,’laughs’])
40 # recognize(g1,[’Bill’,’knows’,’that’,’Sue’,’laughs’])
41 # recognize(g1,[’Sue’,’laughed’])
42 # recognize(g1,[’the’,’student’,’from’,’the’,’university’,’praises’,’the’,’beer’,’on’,’Tuesday’])
43 # recognize(g1,[’the’,’student’,’from’,’the’,’university’,’praises’,’the’])
44 # recognize(g1,[’Sue’,’knows’,’that’,’Maria’,’laughs’])
```

Adding a timer to our basic recognizer makes it clear that backtracking to find alternative parsers also takes many steps! The changes to `tdp.py` are just the lines 6,14,18,31,32,36:

```
1  """ file: tdpt.py  stabler@ucla.edu
2      return the rules used in succsesful derivation
3       with a step-counter (’time’, measured in steps)
4  """
5  def tdpstep(g,(i,cs,p)): # compute all possible next steps from (i,cs)
6      global steps
7      if len(cs)>0:
8          cs1=cs[1:] # copy of predicted categories except cs[0]
9          p1 = p[:]  # copy of rewrites so far
10         nextsteps=[]
11         for (lhs,rhs) in g:
12             if lhs == cs[0]:
13                 #print ’expand’,lhs,’->’,rhs  # for trace
14                 steps = steps+1
15                 nextsteps.append((i,rhs+cs1,p1+[[lhs]+rhs]))
16         if len(i)>0 and i[0] == cs[0]:
17             #print ’scan’,i[0] # for trace
18             steps = steps+1
19             i1=i[1:]
20             nextsteps.append((i1,cs1,p1))
```

```
        return nextsteps
    else:
        return []

def derive(g,ds):
    while ds != [] and not (ds[-1][0] == [] and ds[-1][1] == []):
        d = ds.pop()
        ds.extend(tdpstep(g,d))

def parse(g,i):
    global steps
    steps = 0
    ds = [(i,['S'],[])]
    while ds != []:
        derive(g,ds)
        print 'steps=',steps
        if ds == []:
            return 'False'
        else:
            d=ds.pop()
            print 'll=',d[2]
            print 'parse length=',len(d[2])
            ans = raw_input('another? ')
            if len(ans)>0 and ans[0]=='n':
                return d[2]

# Examples:
# parse(g1,['Sue','laughs'])
# parse(g1,['the','student','laughs'])
# parse(g1,['the','student','praises','the','beer'])
# parse(g1,['Bill','knows','Sue','laughs'])
```

2.8 Conclusions: We have to revise our goal!

The previous section established the following things:

- Computing a list of all parses is not possible for CF grammars in general, since a single string can have infinitely many parses!
- Even when every string has finitely many parses, computing a list of all parses is not tractable for CF grammars in general, since there can be exponentially many of them! In fact, this happens in human languages…
- In human languages, given standard assumptions about constituency, no polynomial function of n bounds the number of parses of strings of length n.

These facts require that we change our idea about our main question, from page 12. It is not reasonable to ask how people compute (all) grammatical structures from orthographic or phonetic representations, since it is not plausible that they do! There are a number of possible responses to this worry. Making our assumptions explicit, consider these alternatives:

Q1a. Since humans cannot map orthographic or phonetic input to complete, explicit analyses, they must use some more compact representation of (all) those structures. What algorithms can compute those?[6]

Q1b. It is unreasonable to assume that we compute all the structures of the sentences we hear, and it is well known that people systematically fail to notice many ambiguities in sentences they hear. SO language users must somehow rank candidate analyses, implicitly (and perhaps probabilistically), and then restrict their attention to the most probable one(s), in context. What algorithms can do that?

Q1c. Obviously, each language user has limited memory. There must be some particular bound k on the number of elements that can be remembered at any time in analyzing a sentence – a bound on the parser memory. When

[6]When computer scientists say that context free grammars can be efficiently parsed, they usually have this kind of idea in mind. They do not mean that all the parse structures can be listed efficiently. Instead, their parsers output something other than explicit trees, or explicit lists of rules used in each parse. We will consider such alternatives below, when we study chart parsers and consider their plausibility as models of human sentence processing.

the parser reaches that bound, the derivation must just crash, ceasing to be available for consideration.[7] So we might ask: what algorithms do this in a human-like way?

There are many other responses to the basic facts about human sentence recognition, each of which poses slightly different questions. We will not try to decide among them here. Certain core assumptions are shared by all of them:

(i) parsing involves finding derivations from the grammar (i.e. from some finite memory of linguistic structure),

(ii) to a first approximation, derivations are built up incrementally from left to right

(iii) to a first approximation, meaning is calculated incrementally from left to right

We can focus on achieving these in various ways, postponing decisions about which of Q1a-Q1c is the right perspective.

[7] In fact, there are reasons to think that the number of elements that can be remembered depends in part on what those elements are. Remembering 3 clearly distinct things may be easier than remembering 3 very similar things. This may be one reason that sentences like these are so difficult:

- Buffalo buffalo buffalo Buffalo buffalo
  with the syntax of: [California girls] like [Pacific waves]
- Dogs dogs dog dog dogs
  with the syntax of: [Mice (that) cats chase eat cheese]

There is a big literature exploring memory and interference effects of various kinds in parsing [].

Exercises: Memory requirements for our TD recognizer

Get `tdh.py` and copy it to `tdh⟨your-initials⟩.py`. Then modify the program as follows (small changes!)...

1. Instead of just the steps in the parse, modify the program to also print out the number of states in parser memory, like this:

```
0  1  (['Sue', 'laughs'], ['S'])
1  2  (['Sue', 'laughs'], ['DP', 'VP'])
2  2  (['Sue', 'laughs'], ['Name', 'VP'])
3  2  (['Sue', 'laughs'], ['Sue', 'VP'])
4  1  (['laughs'], ['VP'])
5  1  (['laughs'], ['V'])
6  1  (['laughs'], ['laughs'])
7  0  ([], [])
```

   Here each line has the number of the step and then the number of elements in parser memory at that step.

2. Now make a second change. Instead of printing out just the steps in the proof, print out the number of states in parser memory and the number of derivations in the backtrack stack just before the current step was taken, like this:

```
0  1  1  (['Sue', 'laughs'], ['S'])
1  2  1  (['Sue', 'laughs'], ['DP', 'VP'])
2  2  1  (['Sue', 'laughs'], ['Name', 'VP'])
3  2  3  (['Sue', 'laughs'], ['Sue', 'VP'])
4  1  4  (['laughs'], ['VP'])
5  1  4  (['laughs'], ['V'])
6  1  4  (['laughs'], ['laughs'])
7  0  4  ([], [])
```

   Here each line has the number of the step, the number of elements in parser memory at that step (as before), and then the number of elements in the backtrack stack at each point.

   We can check this with `td.py`, since that program lists all the derivations at every point. For example, at the last step in the parse of `Sue laughs`, we can see that there are, in fact, 4 derivations in the backtrack stack:

```
...
———
scan laughs
0 ( , )
1 ( Sue laughs , Bill VP )
2 ( Sue laughs , NP VP )
3 ( Sue laughs , D NP VP )
———
True
```

3. Using g1.py and your answer to the previous question:
   a. What sentence of 10 words or less requires the most parser memory? That is, find a sentence that, at some point in calculating one of its parses, has n elements in parser memory for n as high as you can get it. Tell me what the sentence is and what n is in a comment at the bottom of your file `tdh⟨your-initials⟩.py`. (I will check your answer!)
   b. In a sentence or two, explain why your answer to the previous question maximizes parser memory use – again put this in a comment at the bottom of your file `tdh⟨your-initials⟩.py`.
   c. What sentence of 10 words or less requires the most backtrack memory? That is, find a sentence that, at some point in calculating one of its parses, has n derivations elements in its backtrack stack for n as high as you can get it. Tell me what the sentence is, and what n is in a comment at the bottom of your file `tdh⟨your-initials⟩.py`. (I will check your answer!)
   d. In a sentence or two, explain why your answer to the previous question maximizes backtrack stack use – again put this in a comment at the bottom of your file `tdh⟨your-initials⟩.py`.

So you will have a modified parser, with 4 short answers to problem 3 at the bottom of the file. Turn these in via eLC in the usual way, along with your answer to exercise 1, 2, 3 & 4 on page 50 (§3.6)

More exercises: Time requirements for our TD recognizer

*** Not assigned this year! but you can do these for fun if you want ***

4. Modify `tdp.py` to produce the program `tdpt.py` that is listed in §2.7.8 above, name it `tdpt⟨YOURINITIALS⟩.py`, and put your name at the top of the file too.
5. At the top of this file, add a grammar with the name `g4` with the following rules:

$$\text{S} \rightarrow \text{aSS}$$
$$\text{S} \rightarrow \epsilon$$

6. Use your `tdpt.py` to see how many steps it takes to find <u>all</u> parses of each of these sentences:

a, aa, aaa, aaaa, aaaaa, aaaaaa, aaaaaaa

Cut and paste the timer counts for each of these sentences into a comment at the bottom of your file.

7. Also, in the same comment at the bottom of your file, answer this question: For which values of n does the number of steps required to find <u>all</u> parses exceed 2^n? Briefly defend your answer as persuasively as you can.
8. Optionally: sketch a proof that establishes the correctness of your answer to 7.

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] Albro, D. M. *Studies in Computational Optimality Theory, with Special Reference to the Phonological System of Malagasy.* PhD thesis, UCLA, 2005.

[3] Bever, T. G. The cognitive basis for linguistic structures. In *Cognition and the Development of Language*, J. Hayes, Ed. Wiley, NY, 1970.

[4] Chambers, C. G., Tanenhaus, M. K., Eberhard, K. M., Filip, H., and Carlson, G. N. Actions and affordances in syntactic ambiguity resolution. *Journal of Experimental Psychology: Learning, Memory and Cognition 30*, 3 (2004), 687–696.

[5] Chomsky, N., and Halle, M. *The Sound Pattern of English.* MIT Press, Cambridge, Massachusetts, 1968.

[6] Church, K., and Patil, R. How to put the block in the box on the table. *Computational Linguistics 8* (1982), 139–149.

[7] Crain, S., and Steedman, M. On not being led up the garden path. In *Natural Language Parsing*, D. Dowty, L. Karttunen, and A. Zwicky, Eds. Cambridge University Press, NY, 1985.

[8] Eisner, J. Doing OT in a straightjacket. Presented at UCLA, 1999.

[9] Fodor, J. A., and Bever, T. G. The psychological reality of linguistic segments. *Journal of Verbal Learning and Verbal Behavior 4* (1965), 414–420.

[10] Fodor, J. A., Bever, T. G., and Garrett, M. F. *The Psychology of Language: An Introduction to Psycholinguistics and Generative Grammar.* McGraw-Hill, NY, 1974.

[11] Ford, M., Bresnan, J., and Kaplan, R. M. A competence-based theory of syntactic closure. In *The Mental Representation of Grammatical Relations*, J. Bresnan, Ed. MIT Press, Cambridge, Massachusetts, 1982.

[12] Frazier, L. *On Comprehending Sentences: Syntactic Parsing Strategies.* PhD thesis, University of Massachusetts, Amherst, 1978.

[13] Frazier, L., and Clifton, C. *Construal.* MIT Press, Cambridge, Massachusetts, 1996.

[14] Frazier, L., and Rayner, K. Making and correcting errors during sentence comprehension. *Cognitive Psychology 14* (1982), 178–210.

[15] Frazier, L., and Rayner, K. Parameterizing the language system: Left- vs. right-branching within and across languages. In *Explaining Linguistic Universals*, J. A. Hawkins, Ed. Blackwell, NY, 1988, pp. 247–279.

[16] Graf, T. Closure properties of minimalist derivation tree languages. In *Logical Aspects of Computational Linguistics, LACL'11* (2011).

[17] Hartmanis, J. On the succinctness of different representations of languages. *SIAM Journal on Computing 9* (1980), 114–120.

[18] Hopcroft, J. E., Motwani, R., and Ullman, J. D. *Introduction to Automata Theory, Languages and Computation (2nd Edition).* Addison-Wesley, Reading, Massachusetts, 2000.

[19] Hopcroft, J. E., and Ullman, J. D. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, Reading, Massachusetts, 1979.

[20] Jäger, G., and Rogers, J. Formal language theory: Refining the Chomsky hierarchy. *Philosophical Transactions of the Royal Society B 367* (2012), 1956–1970.

[21] Kaplan, R., and Kay, M. Regular models of phonological rule systems. *Computational Linguistics 20* (1994), 331–378.

[22] Khoussainov, B., and Nerode, A. *Automata Theory and Its Applications.* Birkhäuser, Boston, 2001.

[23] Kobele, G. M. Minimalist tree languages are closed under intersection with recognizable tree languages. In *Logical Aspects of Computational Linguistics, LACL'11* (2011), S. Pogodalla and J.-P. Prost, Eds., pp. 129–144.

[24] Ladefoged, P., and Broadbent, D. Perception of sequence in auditory events. *Quarterly Journal of Experimental Psychology 13* (1960), 162–170.

[25] Langendoen, D. T. Limitations on embedding in coordinate structures. *Journal of Psycholinguistic Research 27* (1998), 235–259.

[26] Langendoen, D. T., McDaniel, D., and Langsam, Y. Preposition-phrase attachment in noun phrases. *Journal of Psycholinguistic Research 18* (1989), 533–548.

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

[28] Michaelis, J. Derivational minimalism is mildly context-sensitive. In *Proceedings, Logical Aspects of Computational Linguistics, LACL'98* (NY, 1998), Springer, pp. 179–198.

[29] Miller, G. A. Progject grammarama. In *Psychology of Communication.* Basic Books, NY, 1967.

[30] Pritchett, B. L. *Grammatical Competence and Parsing Performance.* University of Chicago Press, Chicago, 1992.

[31] Riggle, J. *Generation, Recognition, and Learning in Finite State Optimality Theory.* PhD thesis, University of California, Los Angeles, 2004.

[32] Sakarovitch, J. *Elements of Automata Theory.* Cambridge University Press, NY, 2009.

[33] Salomaa, A. *The Theory of Automata.* Pergamon, NY, 1969.

[34] Salomaa, A. *Formal Languages.* Academic, NY, 1973.

[35] Shieber, S., and Johnson, M. Variations on incremental interpretation. *Journal of Psycholinguistic Research 22* (1994), 287–318.

[36] Sipser, M. *Introduction to the Theory of Computation.* PWS Publishing, Boston, 1997.

[37] Stabler, E. P. The finite connectivity of linguistic structure. In *Perspectives on Sentence Processing*, C. Clifton, L. Frazier, and K. Rayner, Eds. Lawrence Erlbaum, Hillsdale, New Jersey, 1994, pp. 245–266.

[38] Thatcher, J. W. Characterizing derivation trees of context-free grammars through a generalization of finite automata theory. *Journal of Computer and System Sciences 1*, 4 (1967), 317 – 322.

Chapter 3 Beam instead of backtrack, and more alternatives

A beam parser is one where the search is restricted to a finite collection, a finite 'beam' of options at each point [11, 10, 7, 6, 13, 12, 8]. One variety of these strategies is called 'k-best': the search is limited to the k most probable parses. A simple beam parser is introduced here. It provides a kind of top-down analysis that can 'handle' left recursion, in a sense, though it does so in a manner that we will see is less than satisfactory. The basic idea is simply to always evaluate the simplest alternative parse (or the parse that is 'most probable' in some other sense), within a bound set by a parameter k. Any parses with probability less than k are discarded.

3.1 The TD beam recognizer

The data structure we need for the beam can be thought of as a sorted list, sorted by probability, from which we always pop one of the elements with highest probability. Since this list must be sorted, it is not a stack, but is sometimes called a *priority queue* or *heap*.[1] The other ingredient we need is some way of determining the probability of each parse. We will return to this question later, but for now let's just say that each option is equally likely. That is, whenever a parse with probability p can expand in n ways, we assign each of the n possible parses probability p/n. And in order to make sure that the beam stays a reasonable size, we will set a finite bound k on the minimum probability parse that we want to consider.

```
Top-down beam CF recognition($G, i, k$)
0  beam=[(1.,i,S)], a priority queue, where S is the start category
1  while beam$\neq$ [] and max(beam)$\neq$(p,[],[])  (any p):
2      (p0,i0,cs0)=pop(beam), the maximum element
3      nextsteps=[d| (i0,cs0) $\Rightarrow_{td}$ d]
4      p1=p0*$\frac{1}{\text{len(nextsteps)}}$
5      if p > $k$:
6          for each (i1,cs1) in nextsteps, push (p1,i1,cs1) onto beam
7  if beam==[] then False else True
```

where the steps $\Rightarrow_{td}$ are unchanged:

$$\text{(expand)}\ \frac{\text{input}, \mathrm{X}\alpha}{\text{input},\ \beta\alpha}\ \text{ if } \mathrm{X} \to \beta \qquad \text{(scan)}\ \frac{\text{w input}, \text{w}\alpha}{\text{input}, \alpha}.$$

If k is negative, then all derivations will be kept in the beam, as was done in top-down recognition. But the beam recognizer always expands one of the maximum probability parses, so unlike the top-down recognizer, the beam recognizer will always terminate on a grammatical input. Like the top-down parser, though, if k is negative, the recognizer can fail to terminate on ungrammatical inputs if the grammar has left recursion. (And even with positive k, with some unusual left recursive grammars, the recognizer fail to terminate – In what cases can this happen?)

3.2 Implementing the TD beam recognizer

Python provides a priority queue, a heap. To use this data structure, we load the basic capabilities with the command:

```
>>> import heapq
```

That command adds the heapq library, which allows us to create heaps. Any list can be converted into a heap:

[1]The implementation of these structures is interesting, a standard topic in classes on data structures. See, for example, [1, §6].

```
>>> l1=[4,1,3,2]
>>> heapq.heapify(l1)
```

Now we can pop the minimum element, and push new elements:

```
>>> x = heapq.heappop(l1)
>>> x
1
>>> l1
[2, 4, 3]
>>> y = heapq.heappop(l1)
>>> y
2
>>> x = heapq.heappop(l1)
>>> x
3
>>> l1
[4]
heapq.heappush(l1,-3)
>>> l1
[-3, 4]
heapq.heappush(l1,5)
>>> l1
[-3, 4, 5]
```

We can use this for our probabilities, but since we want to pop the maximum probability elements, we keep the probability negated, and we let represent our derivations with triples

$$(-\text{probability}, \text{remaining input}, \text{predicted categories}).$$

The *expand* and *scan* steps can remain unchanged, but whenever there are n possible next steps from a parse with probability p, we assign each possibility p/n.

In the following implementation, the function `tdstep` is unchanged from the top-down parser `td.py`. The only changes are in `recognize`, especially in the lines 25, 27 and 31-36:

```
1  """ file: tdb.py
2      A simple top-down beam CF recognizer.
3      The minor changes from td.py are labeled "for beam"
4  """
5  import heapq
6
7  def tdstep(g,(i,cs)): # compute all possible next steps from (i,cs)
8      if len(cs)>0:
9          cs1=cs[1:] # copy of predicted categories except cs[0]
10         nextsteps=[]
11         for (lhs,rhs) in g:
12             if lhs == cs[0]:
13                 print 'expand',lhs,'->',rhs   # for trace
14                 nextsteps.append((i,rhs+cs1))
15         if len(i)>0 and i[0] == cs[0]:
16             print 'scan',i[0] # for trace
17             nextsteps.append((i[1:],cs1))
18         return nextsteps
19     else:
20         return []
21
22 def recognize(g,i,k):
23     beam = [(-1.,i,['S'])] # probability 1., negated
24     heapq.heapify(beam) # make list into a "min-heap"
25     while beam != [] and not(min(beam)[1]==[] and min(beam)[2]==[]):
26         (prob0,i0,cs0) = heapq.heappop(beam)
27         print 'popped',(prob0,i0,cs0) # for trace
28         nextsteps = tdstep(g,(i0,cs0))
29         print 'next steps=',nextsteps
30         if len(nextsteps) > 0:
31             prob1 = prob0/float(len(nextsteps))
32             if -(prob1) > k:
33                 for (i1,cs1) in nextsteps:
34                     heapq.heappush(beam,(prob1,i1,cs1))
35                     print 'pushed',(prob1,i1,cs1) # for trace
36         print '|beam|=',len(beam) # for trace
37     if beam == []:
38         return False
```

```
    else:
        return True

# Examples:
# recognize(g1,['Sue','laughs'],0.05)
# recognize(g1,['Sue','laughs'],0.5)
# recognize(g1,['Bill','knows','that','Sue','laughs'],0.10)
# recognize(g1,['Bill','knows','that','Sue','laughs'],0.01)
```

3.3 Implementing the TD beam parser

It is easy to make the same changes to the top-down parser `tdp.py` to get this top-down beam parser:

```
""" file: tdbp.py  stabler@ucla.edu
    beam parser
"""
import heapq

def tdpstep(g,(ws,cs,p)): # compute all possible next steps from (ws,cs)
    if len(cs)>0:
        cs1=cs[1:] # copy of predicted categories except cs[0]
        p1 = p[:]  # copy of rules used so far
        nextsteps=[]
        for (lhs,rhs) in g:
            if lhs == cs[0]:
                #print 'expand',lhs,'->',rhs  # for trace
                nextsteps.append((ws,rhs+cs1,p1+[[lhs]+rhs]))
        if len(ws)>0 and ws[0] == cs[0]:
            #print 'scan',ws[0] # for trace
            ws1=ws[1:]
            nextsteps.append((ws1,cs1,p1))
        return nextsteps
    else:
        return []

def derive(g,beam,k):
    while beam != [] and not (min(beam)[1] == [] and min(beam)[2] == []):
        (prob0,ws0,cs0,p0) = heapq.heappop(beam)
        nextsteps = tdpstep(g,(ws0,cs0,p0))
        #print 'nextsteps=',nextsteps
        if len(nextsteps) > 0:
            prob1 = prob0/float(len(nextsteps))
            if -(prob1) > k:
                for (ws1,cs1,p1) in nextsteps:
                    heapq.heappush(beam,(prob1,ws1,cs1,p1))
                    #print 'pushed',(prob1,ws1,cs1) # for trace
        #print '|beam|=',len(beam) # for trace

def parse(g,ws,k):
    beam = [(-1.,ws,['S'],[])]
    heapq.heapify(beam) # make list of derivations into a "min-heap"
    while beam != []:
        derive(g,beam,k)
        if beam == []:
            return 'False'
        else:
            d=heapq.heappop(beam)
            print 'll=',d[3]
            ans = raw_input('another? ')
            if len(ans)>0 and ans[0]=='n':
                return d[3]

# Examples:
# parse(g1,['Sue','laughs'],-1.)
# parse(g1,['Bill','knows','that','Sue','laughs'],-1.)
# parse(g0,['Sue','laughs'],0.01)
# parse(g0,['Sue','laughs'],0.0001)
# parse(g0,['the','student','laughs'],0.0001)
# parse(g0,['the','student','laughs'],0.000001)
# parse(g0min,['the','kind','student','laughs'],0.0000001)
```

3.4 Implementing the original grammar, with left recursion

Since the top-down beam parser can now handle left recursion, we can go back to the original grammar from Fromkin, which we listed on page 11:

```
""" file: g0.py
    our first grammar. It has left recursion and empty productions.
"""
g0 = [('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']),
      ('NP',['NP','PP']), # left rec
      ('NP',['NP','CP']), # left rec
      ('VP',['AdvP','VP']),
      ('VP',['VP','PP']), # left rec
      ('AP',['AdvP','AP']),
      ('D',['D','Coord','D']), # left rec
      ('V',['V','Coord','V']), # left rec
      ('N',['N','Coord','N']), # left rec
      ('A',['A','Coord','A']), # left rec
      ('P',['P','Coord','P']), # left rec
      ('C',['C','Coord','C']), # left rec
      ('Adv',['Adv','Coord','Adv']), # left rec
      ('VP',['VP','Coord','VP']), # left rec
      ('NP',['NP','Coord','NP']), # left rec
      ('DP',['DP','Coord','DP']), # left rec
      ('AP',['AP','Coord','AP']), # left rec
      ('PP',['PP','Coord','PP']), # left rec
      ('AdvP',['AdvP','Coord','AdvP']), # left rec
      ('S',['S','Coord','S']), # left rec
      ('CP',['CP','Coord','CP']), # left rec
      ('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',[]), # empty production!
      ('C',['whether']),
      ('Coord',['and']),
      ('Coord',['or']),
      ('Coord',['but'])]
```

With this grammar, we get sessions like this:

```
>>> from tdbp import *
>>> from g0 import *
>>> parse(g0,['Sue','laughs'],0.0001)
1l= [['S', 'DP', 'VP'], ['DP', 'Name'], ['Name', 'Sue'], ['VP', 'V'], ['V', 'laughs']]
another? y
'False'
>>> parse(g0,['the','student','laughs'],0.0001)
'False'
>>> parse(g0,['the','student','laughs'],0.00001)
'False'
>>> parse(g0,['the','student','laughs'],0.000001)
1l= [['S', 'DP', 'VP'], ['DP', 'D', 'NP'], ['D', 'the'], ['NP', 'N'], ['N', 'student'], ['VP', 'V'], ['V', 'laughs']]
another? n
```

3.5 Assessment: Time, space, and nondeterminism

The situation is only slightly changed from the TD parser, so we list the basic properties quickly here.

1. The rules $\Rightarrow_{td}$ are sound and complete, as before – see §2.7.1.
2. The structures parsed at each step are connected, allowing for incremental interpretation, as before – see §2.7.2.
3. This method develops one parse at a time – the maximum probability parse in the beam, at each step – and so is similar to the top-down recognizer in suggesting a treatment of 'garden path' sentences – see 2.7.3.
4. Right branching derivations only require a finite amount of parser memory, no matter how long the input is – see §2.7.6
5. Left branching derivations can require unbounded amounts of parser memory, increasing with the length of the input – see §2.7.4
6. Lookahead can reduce indeterminacy, but not effectively for languages like English. English is not LL(k) for any k – see §2.7.7.
7. And of course, English has intractable ambiguity, no matter what recognizer you use – see §2.7.7.

The new thing is this:

8. Given a fixed, positive $0 < k \leq 1$, most left recursive grammars will not cause non-termination. (But see exercise 6 on page 50 below.

3.6 More alternatives, and conclusions

TD parsing has lots of nice properties, and the beam parser shares many of them. And certain cases of non-termination and intractability could be avoided with this approach.

But the beam parser is still exploring many parses which a 'smarter' parser might reject, e.g. parses that could have been avoided by looking one word ahead, or parses that could have been avoided by better judging plausibility in context. To take just one example: Do we really predict all the nouns (within suitably generous probability bounds) and then check our predictions? Not likely! And do we really want a parser to have to predict how many modifiers a phrase has, before seeing any of the phrase? That seems unlikely too. There are many other search methods: simple breadth-first, A* [9], Hale's adaptive methods [5], and more. And any of these algorithms could be implemented in 'hardware' that provides more or less parallelism, complicating the relation between the number of steps required ('time' in the computer scientists' sens) and the real time required by the hardware (and in particular, by any neurophysiological implementation). We should keep the basic questions about search, the questions about how to handle indeterminacy, in mind as we consider other parsing strategies.

Exercises: Time requirements for our TDB recognizer

Due by the end of the day on Monday (i.e. by midnight) January 28.

1. Modify `tdbp.py` to produce the program `tdbpt.py` in the same way that we modified the simple top-down parser in §2.7.8 (in the updated version of the week 2 notes). Name your file `tdbpt⟨YOURINITIALS⟩.py`, and put your name at the top of the file too.
2. At the top of this file, add a grammar with the name `g4` with the following rules:

$$\begin{array}{l} \text{S} \to \text{aSS} \\ \text{S} \to \epsilon \end{array}$$

3. Use your `tdbpt.py` to see how many steps it takes to find <u>all</u> parses of each of these sentences, using `g4`:

   a, aa, aaa, aaaa, aaaaa, aaaaaa, aaaaaaa

   Cut and paste the step counts for each of these sentences into a comment at the bottom of your file.
4. In the same comment at the bottom of your file, answer this question: For which values of n does the number of steps required to find <u>all</u> parses of the sentence a^n exceed 2^n?
5. Optionally: Sketch a proof that establishes the correctness of your answer to 4.
6. Optionally: Can you design a grammar which can cause this parser to be nonterminating when $k < 0$? Present the grammar and explain the problem.
7. A prominent tradition in psycholinguistics holds that typical sentence processing involves development of a single analysis, where that parse is subjected to 'reanalysis' when required. For example, Frazier and Clifton say:

   > Serial theories of sentence processing specify that the processor pursues just a single analysis of a sentence until that analysis becomes implausible or untenable, at which point revision of the first analysis occurs... Here it is argued that revision cost cannot be calculated in purely structural terms... It is also argued that a theory of revisions must include a Minimal Revisions principle... [3, p.193]

   In beam search, the most probable parse is developed at each So then, is a beam processor a serial model with reanalysis? Compare this idea to Frazier and Clifton's or other psycholinguistic proposals about reanalysis [2, 4, etc]. Staub's [14] proposal is especially interesting: he suggests that we see the influence of discarded parses at later stages of processing. Is there an alternative interpretation of his results?

References

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to Algorithms, 2nd Edition.* MIT Press, Cambridge, Massachusetts, 2001.

[2] Fodor, J. D., and Inoue, A. Garden path reanalysis: Attach (anyway) and revision as last resort. In *Cross-linguistic Perspectives on Language Processing*, M. De Vincenzi and V. Lombardo, Eds. Kluwer, Boston, 2000, pp. 21–61.

[3] Frazier, L., and Clifton, C. Sentence reanalysis and visibility. In *Reanalysis in Sentence Processing*, J. D. Fodor and F. Ferreira, Eds. Kluwer, Boston, 1998, pp. 143–176.

[4] Grodner, D., Gibson, E., Argaman, V., and Babyonshev, M. Against repair-based reanalysis in sentence comprehension. *Journal of Psycholinguistic Research 32*, 2 (2003), 141–166.

[5] Hale, J. T. What a rational parser would do. *Cognitive Science 35*, 3 (2011), 399–443.

[6] Huang, L., and Chiang, D. Better k-best parsing. In *Proceedings of the International Workshop on Parsing Technologies (IWPT)* (2005), pp. 53–64.

[7] Jiménez, V. M., and Marzal, A. Computation of the n best parse trees for weighted and stochastic context-free grammars. In *Proceedings of the Joint IAPR International Workshops on Advances in Pattern Recognition* (London, UK, 2000), Springer-Verlag, pp. 183–192.

[8] Jurafsky, D. A probabilistic model of lexical and syntactic access and disambiguation. *Cognitive Science 20* (1996), 137–194.

[9] Klein, D., and Manning, C. D. A* parsing: Fast exact Viterbi parse selection. In *Human Language Technology Conference - North American chapter of the Association for Computational Linguistics annual meeting (HLT-NAACL)* (2003).

[10] Pauls, A., and Klein, D. K-best A* parsing. In *Proceedings of the Association for Computational Linguistics* (2009).

[11] Pauls, A., Klein, D., and Quirk, C. Top-down k-best A* parsing. In *Proceedings of the Association for Computational Linguistics* (2010), pp. 275–298.

[12] Roark, B. Probabilistic top-down parsing and language modeling. *Computational Linguistics 27*, 2 (2001), 249–276.

[13] Roark, B. Robust garden path parsing. *Natural Language Engineering 10*, 1 (2004), 1–24.

[14] Staub, A. The return of the repressed: Abandoned parses facilitate syntactic reanalysis. *Journal of Memory and Language 57*, 2 (2007), 299 – 323.