CS61A Lecture 37

Monday, December 2, 2019

Announcements

- There is no new content in CS61A for the next week. Today, we will go over Prof. DeNero's research in natural language.
- Wednesday will be a review of the course.
- And Friday will be our final lecture.
- CSM will be running review sessions at the usual times and lecture halls next week.

Ambiguity

Syntactic Ambiguity in English

Harold Abelson, Gerald Sussman and Julie Sussman, Structure and Interpretation of Computer Programs:

"Programs must be written for people to read"

```
|-noun-||-------verb phrase--------|
|-noun-||----verb---||-subordinate-|
|-noun-||--------verb--------||verb|
```

Above are the three ways you could read the same sentence, thanks to the ambiguous rules of English. Today, we will build a syntactic parser, which takes in a phrase and outputs a tree of the possible clauses.

Syntax Trees

First, we will try to figure out what that tree, a syntax tree, is.

Representing Syntactic Structure

We will use a `Tree` class to represent a phrase, but it is not the same `Tree` class we've used all semester. It will be implemented as such:

A `Tree` represents a phrase, with:

- `tag` – What kind of phrase (e.g. S, NP, VP)
- `branches` – Sequence of `Tree` or `Leaf` components

A `Leaf` represents a single word;

- `tag` – What kind of word (e.g. N, V)
- `leaf` – The word itself

Here's an example of this in action:

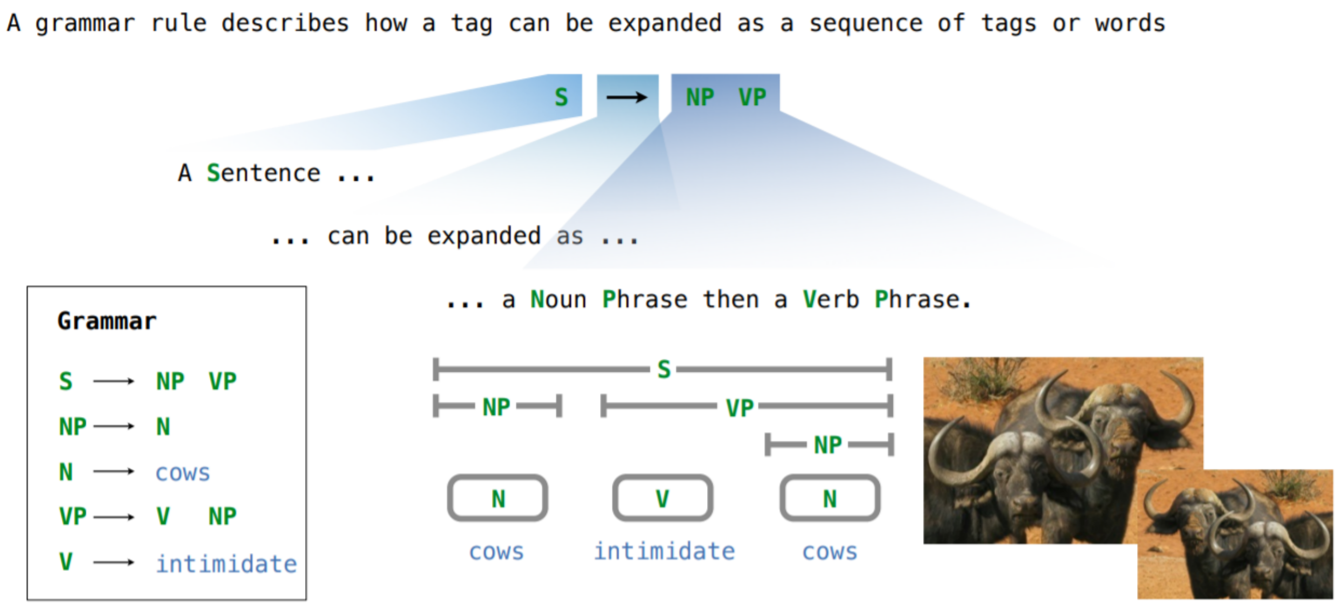
"cows intimidate cows"

```
cows = Leaf('N', cows)
intimidate = Leaf('V', `intimidate`)
S, NP, VP =  'S', 'NP', 'VP'

s = Tree(S, [Tree(NP, [cows]),
         Tree(VP, [intimidate,
                   Tree(NP, [cows])])])
```

And we can write a `print_tree` function like before:

```
def print_tree(t, indent=0, end='\n'):
    if isinstance(t, Leaf):
        print(t, end='')
    else:
        s = '(' + t.tag + ' '
        indent += len(s)
        print(s, end='')
        print_tree(t.branches[0], indent, '')
        for b in t.branches[1:]:
            print('\n' + ' '*indent, end='')
            print_tree(b, indent, '')
        print(')', end=end)

>>> print_tree(s)
(S (NP (N cows))
   (VP (V intimidate)
       (NP (N cows))))
```

We can now write very basic rules describing the grammar, just for this sentence.


A grammar rule describes how a tag can be expanded as a sequence of tags or words
S → NP VP
A Sentence ...
... can be expanded as ...
... a Noun Phrase then a Verb Phrase.
Grammar
S → NP VP
NP → N
N → cows
VP → V NP
V → intimidate
S
NP
VP
NP
N
V
N
cows
intimidate
cows

Here's how we can start:

```
lexicon = {
    Leaf('N', 'buffalo'), # beasts
    Leaf('V', 'buffalo'), # intimidate
    }

grammar = {
    'S':  [['NP', 'VP']],
    'NP': [['N']],
    'VP': [['V', 'NP']],
    }
```

In order to expand your tags, we need to write a generator function called `expand` :

```
def expand(tag):
    """Yield all trees rooted by tag."""
    for leaf in lexicon:
        if leaf.tag == tag:
            yield leaf
    if tag in grammar:
        for tags in grammar[tag]:
            for branches in expand_all(tags):
                yield Tree(tag,branches)

def expand_all(tags):
    """Yield all sequences of branches for a sequence of tags."""
    if len(tags)==1:
        for branch in expand(tags[0]):
            yield [branch]
        else:
            first, rest = tags[0], tags[1:]
            for first_branch in expand(first):
                for rest_branches in expand_all(rest):
                    yield [first_branch]+rest_branches
```

And now, let's open this parser program:

```
>>> for tree in expand('S'):
...     print_tree(tree)
(S (NP (N cows))
   (VP (V intimidate)
       (NP (N cows))))
```

The parser **built** this sentence, on its own! If we expanded the grammar rules, it could build a list of all possible sentences, as the grammar rules allow.

Exhaustive Parsing

We want to expand all tags recursively, but we constrain words to match input. Let's say we want to expand the sentence "buffalo buffalo buffalo buffalo"

```
[0] buffalo [1] buffalo [2] buffalo [3] buffalo [4]
```

In order to do so, we need to keep track of where we are. We split them up between words. We then build a constraint that a `Leaf` must match the input, so that we only expand the tags that matter to the current context.

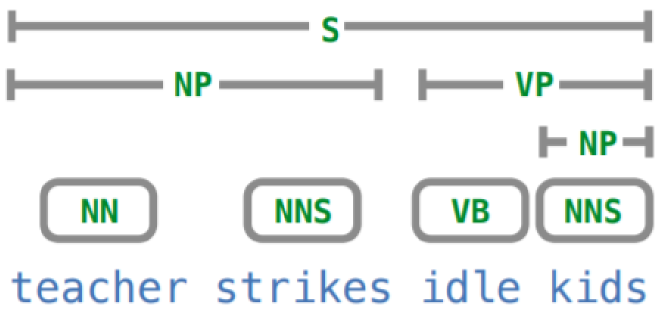
We can do this by fixing our implementation of `expand` and `expand_all` :

```
def parse(line):
    words = line.split()

    def expand(start,end,tag):
        """Yield all trees rooted by tag."""
        if end-start==1:
            word = words[start]
            for leaf in lexicon:
                if leaf.tag == tag and leaf.word==word:
                    yield leaf
            if tag in grammar:
                for tags in grammar[tag]:
                    for branches in expand_all(tags,start,end):
                        yield Tree(tag,branches)

    def expand_all(start,end,tags):
        """Yield all sequences of branches for a sequence of tags."""
        if len(tags)==1:
            for branch in expand(tags[0]):
                yield [branch]
            else:
                first, rest = tags[0], tags[1:]
                for middle in range(start+1,end+1-len(rest))
                for first_branch in expand(first,middle,first):`
                    for rest_branches in expand_all(middle,end,rest):
                        yield [first_branch]+rest_branches`

    >>> for tree in expand(0,len(words),'S'):
    ...     print_tree(tree)
```

Learning

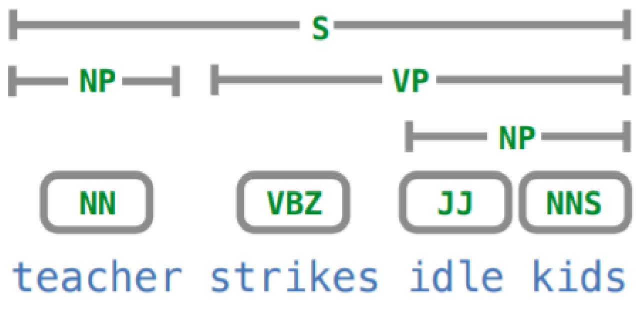
The parser learns by feeding in data. The University of Pennsylvania, for example, hired undergrads to write the rules of sentences written in the Wall Street Journal.

With all these rules written, the parser can now compare sentence trees with those in the rules, and build new sentences on its own. More importantly, it can also predict which sentence interpretation is the “correct” one based on its relative frequency in the examples: how often does that structure appear?



Rule frequency per 100,000 tags

S ⟶ NP VP	25372	NN ⟶	teacher	5
NP ⟶ NN NNS	1335	NNS ⟶	strikes	25
VP ⟶ VB NP	6679	VB ⟶	idle	26
NP ⟶ NNS	4282	NNS ⟶	kids	32



Rule frequency per 100,000 tags

S ⟶ NP VP	25372		NN ⟶	teacher	5	
NP ⟶ NN	~~1335~~	4358	VBZ ⟶	strikes	~~25~~	19
VP ⟶ VBZ NP	~~6679~~	3160	JJ ⟶	idle	~~26~~	18
NP ⟶ JJ NNS	~~4282~~	2526	NNS ⟶	kids	32	