Text Mining
Project/Lab

Behrang QasemiZadeh
behrangatoffice@gmail.com

Information Extraction

Motivation

- With the invention of WWW, the amount of accessible electronic text is soaring.
- If you have a question, it is highly likely that someone has written its answer somewhere.
- The goal of information extraction is to help you find information you are looking for from this gigantic amount of text.
- But considering the complexity and ambiguity of text, how can we achieve this goal?

Motivation

- One way is come up with a general framework for the representation of the meaning in natural language text:
  - Is it really possible, considering the complexity and ambiguity of natural language?
- Another way is to focus on limited set of questions:
  - **Who** is married with **whom**?
  - **Where** is a company located?
  - **What** is the capital of Bavaria?
- The latter seems more feasible, does not it?

Goal

- How can we extract structured data, such as tables, from unstructured text?
- What are the common methods for identifying entities and their relationships in text?
- Which corpora are available for this task, and how they can be used for training and evaluating classifiers to perform information extraction?

Introduction

- We are often interested in information represented as structured data

Organization	Location
Omnicom	New York
DDB Needham	New York
Kaplan Thaler Group	New York
BBDO South	Atlanta
Georgia-Pacific	Atlanta

- The structured data in the table above can be simply presented as a list of tuples: **`(entity, relation, entity)`**

Introduction

- If this data is presented as a list of tuples, then it is easy to answer questions such as

  “Which organizations operate in Atlanta?”

```
>>> locs = [('Omnicom', 'IN', 'New York'),
        ('DDB Needham', 'IN', 'New York'),
       ('Kaplan Thaler Group', 'IN', 'New York'),
        ('BBDO South', 'IN', 'Atlanta'),
       ('Georgia-Pacific', 'IN', 'Atlanta')]
>>> query = [e1 for (e1, rel, e2) in locs if e2=='Atlanta']
>>> print(query)
['BBDO South', 'Georgia-Pacific']
```

Introduction

- Now assume that instead of the previous table we have this text:

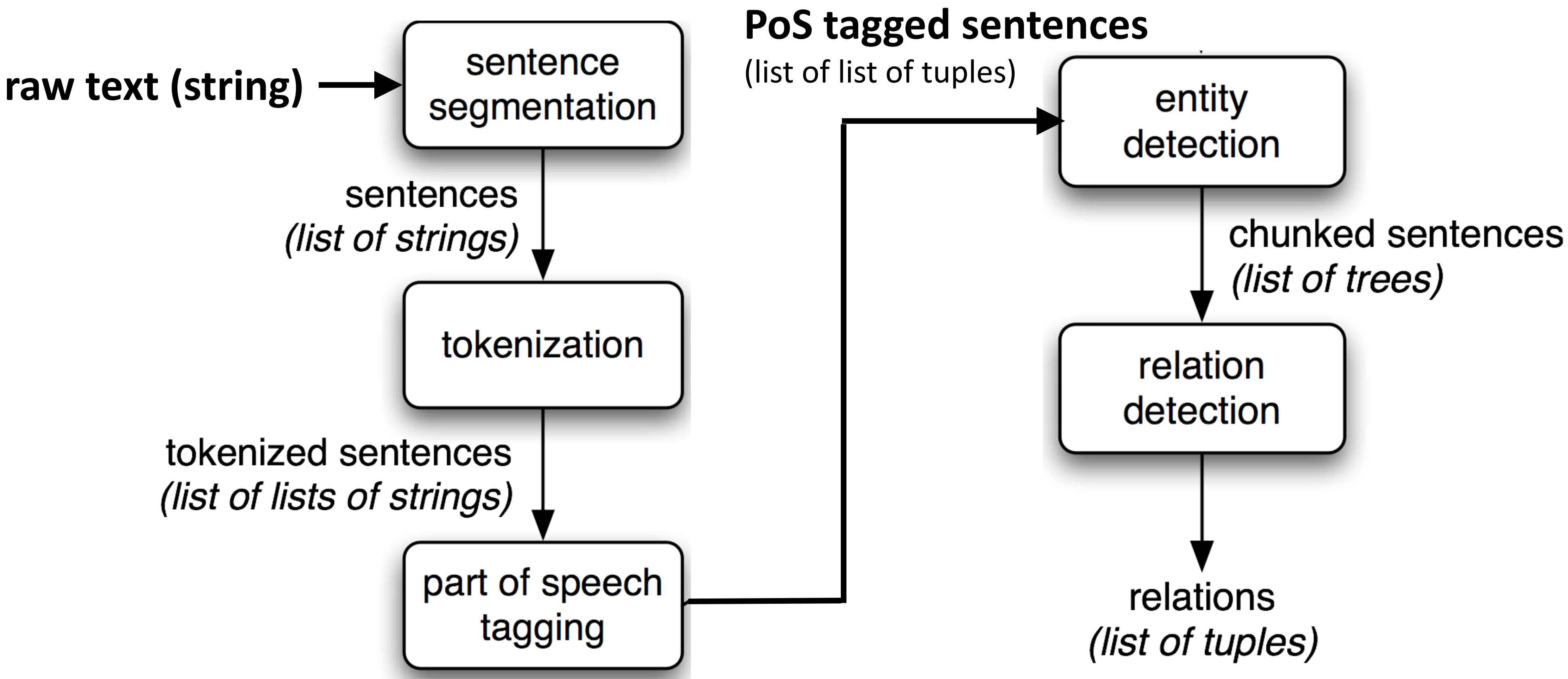
> The fourth Wells account moving to another agency is the packaged paper-products division of Georgia-Pacific Corp., which arrived at Wells only last fall. Like Hertz and the History Channel, it is also leaving for an Omnicom-owned agency, the BBDO South unit of BBDO Worldwide. BBDO South in Atlanta, which handles corporate advertising for Georgia-Pacific, will assume additional duties for brands like Angel Soft toilet tissue and Sparkle paper towels, said Ken Haldin, a spokesman for Georgia-Pacific in Atlanta.

- How to make a computer understand the text above to answer the query “which organizations operate in Atlanta”?

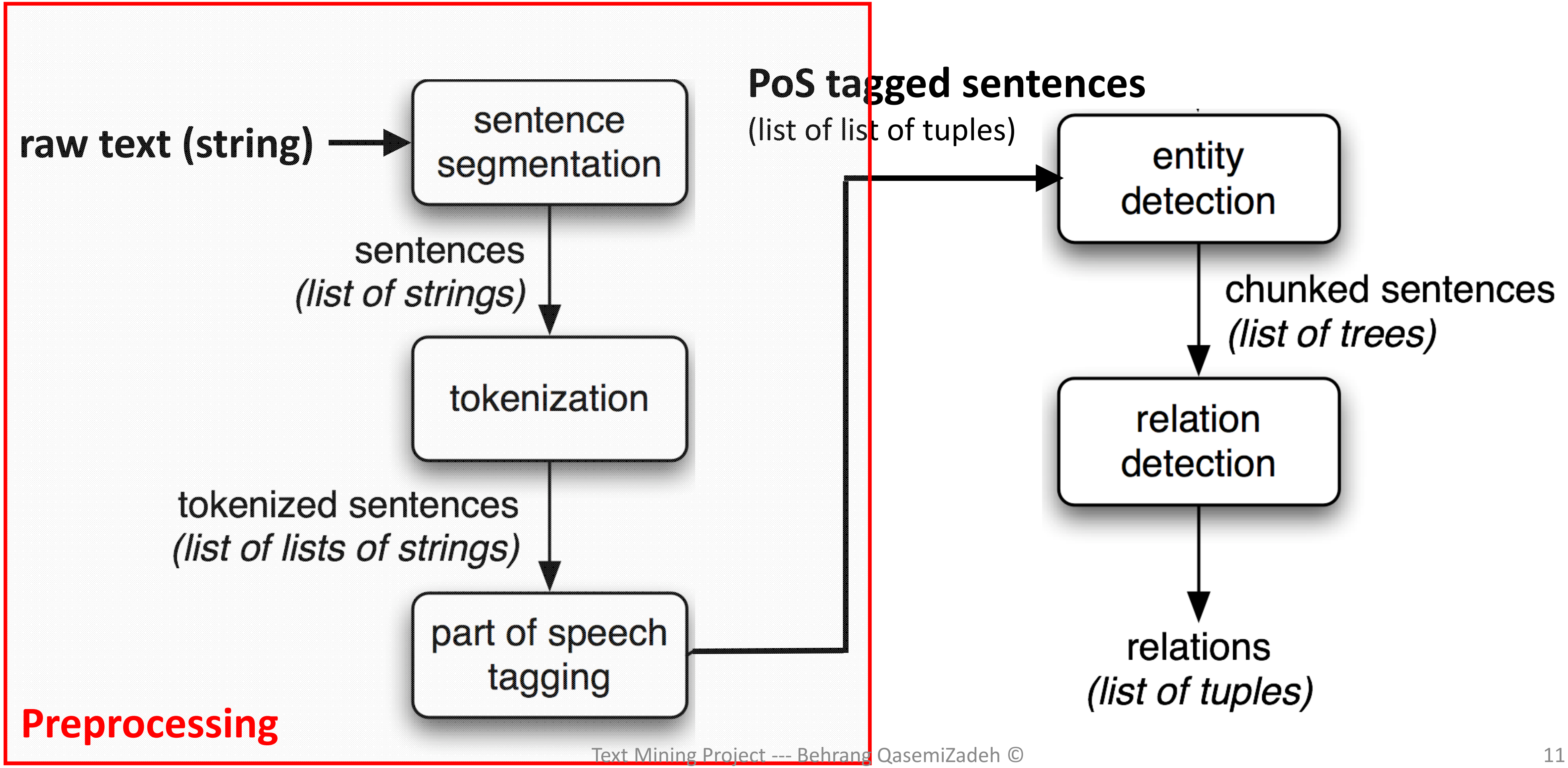
A Practical Solution

- First, convert the unstructured data of natural language sentences into the structured data.
- Second, use powerful tools for querying structured data, e.g. SQL, to retrieve this data.
- The steps listed above is the core of information extraction.

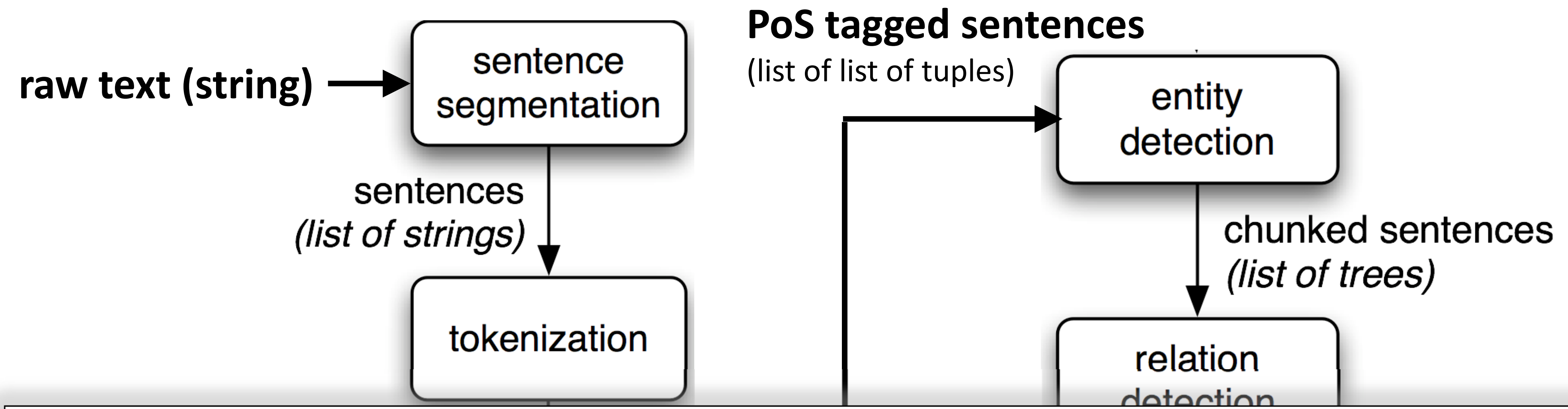
Information Extraction Architecture



Information Extraction Architecture

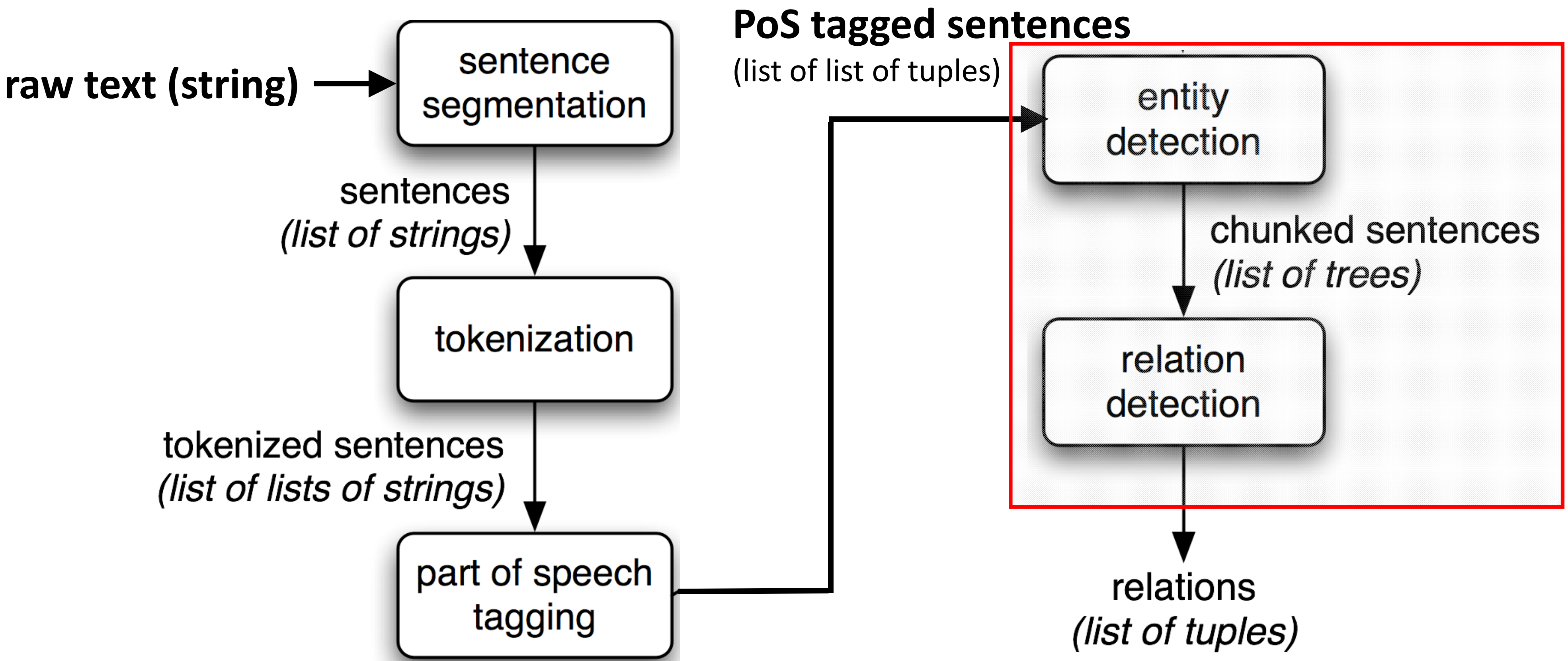


Information Extraction Architecture



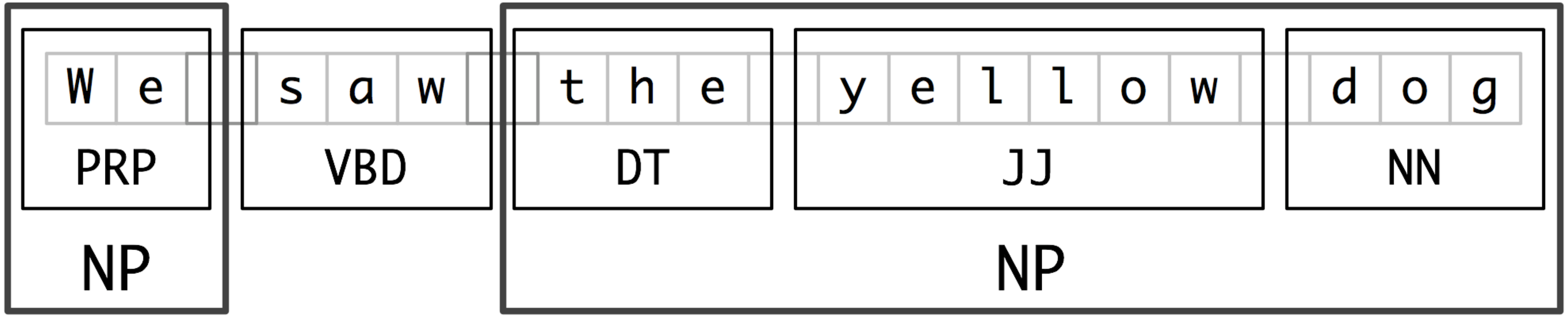

```
>>> import nltk, re, pprint
>>> def ie_preprocess(document):
      sentences = nltk.sent_tokenize(document)
      sentences = [nltk.word_tokenize(sent) for sent in sentences]
      sentences = [nltk.pos_tag(sent) for sent in sentences]
      return sentences
```

Information Extraction Architecture



Chunking

- A **chunker** segments and labels multi-token sequences as one group.
- Each of these multi-token sequences are called a **chunk.**
- Each chunk has a particular grammatical function.



Noun Phrase Chunking

- **NP-chunking (noun phrase chunking)** is the process of finding *smallest* chunks that form a noun phrase:

The market for system-management software for Digital's hardware is fragmented enough that a giant such as Computer Associates should do well there.

[The/DT market/NN **]** for/IN **[** system-management/NN software/NN **]** for/IN
[Digital/NNP **]** **[** 's/POS hardware/NN **]** is/VBZ fragmented/JJ enough/RB that/IN
[a/DT giant/NN **]** such/JJ as/IN **[** Computer/NNP Associates/NNPS **]** should/MD do/VB well/RB there/RB ./.

Building an NP-Chunker

- To perform the NP-Chunking task, we can use PoS tags and a set of chunking rules (**tag patterns)**:

```
>>> grammar = "NP: {<DT>?<JJ>*<NN>}“
>>> cp = nltk.RegexpParser(grammar)
```

Building an NP-Chunker

- To perform the NP-Chunking task, we can use PoS tags and a set of chunking rules (**tag patterns**):

```
>>> grammar = "NP: {<DT>?<JJ>*<NN>}“
>>> cp = nltk.RegexpParser(grammar)

>>> sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"), \
  ("dog", "NN"), ("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]
>>> result = cp.parse(sentence)
>>> print(result)
(S (NP the/DT little/JJ yellow/JJ dog/NN) barked/VBD at/IN (NP the/DT cat/NN))
```

Building an NP-Chunker

- To perform the NP-Chunking task, we can use PoS tags and a set of chunking rules (**tag patterns**):

```
>>> grammar = "NP: {<DT>?<JJ>*<NN>}“
>>> cp = nltk.RegexpParser(grammar)

>>> sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"), \
  ("dog", "NN"), ("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]
>>> result = cp.parse(sentence)
>>> print(result)
(S (NP the/DT little/JJ yellow/JJ dog/NN) barked/VBD at/IN (NP the/DT cat/NN))

>>> result.draw()
```

Building an NP-Chunker

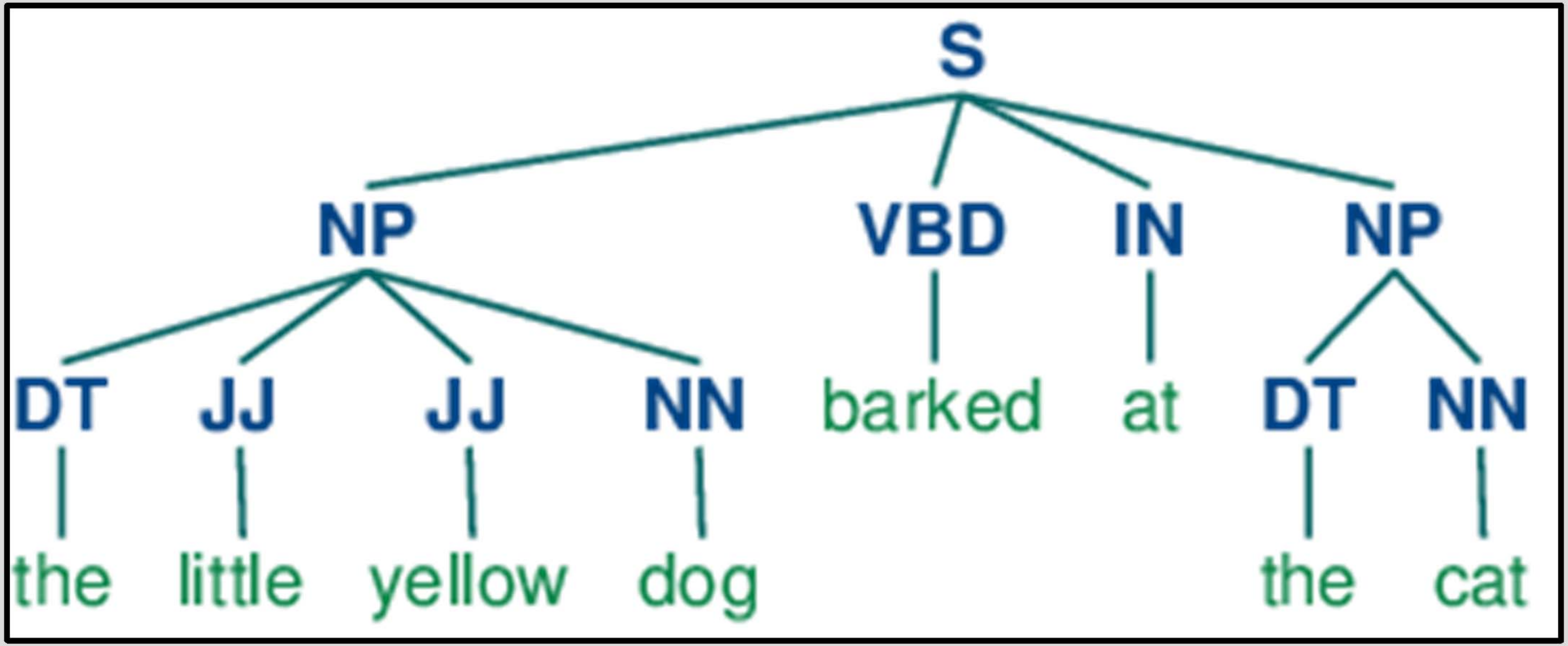
- To perform the NP-Chunking task, we can use PoS tags and a set of chunking rules (**tag patterns)**:

```
>>> grammar = "NP: {<DT>?<JJ>*<NN>}“
>>> cp = nltk.RegexpParser(grammar)

>>> sentence = [("the", "DT"), ("little", "JJ"), ("yellow", "JJ"), \
  ("dog", "NN"), ("barked", "VBD"), ("at", "IN"), ("the", "DT"), ("cat", "NN")]
>>> result = cp.parse(sentence)
>>> print(result)
(S (NP the/DT little/JJ yellow/JJ dog/NN) barked/VBD at/IN (NP the/DT cat/NN))

>>> result.draw()
```

Building an NP-Chunker




```
>>> result.draw()
```

Exercise

- Refine the employed tag pattern in the previous example to cover other patterns such as:

another/DT sharp/JJ dive/NN
trade/NN figures/NNS
any/DT new/JJ policy/NN measures/NNS
earlier/JJR stages/NNS
Panamanian/JJ dictator/NN Manuel/NNP Noriega/NNP
his/PRP$ Mansion/NNP House/NNP speech/NN
the/DT price/NN cutting/VBG

Exercise

- Refine the employed tag pattern in the previous example to cover other patterns such as:

another/DT sharp/JJ dive/NN
trade/NN figures/NNS
any/DT new/JJ policy/NN measures/NNS
earlier/JJR stages/NNS
Panamanian/JJ dictator/NN Manuel/NNP Noriega/NNP
his/PRP$ Mansion/NNP House/NNP speech/NN
the/DT price/NN cutting/VBG

Similar to the practice we did for PoS taggin

Chinking instead of Chunking

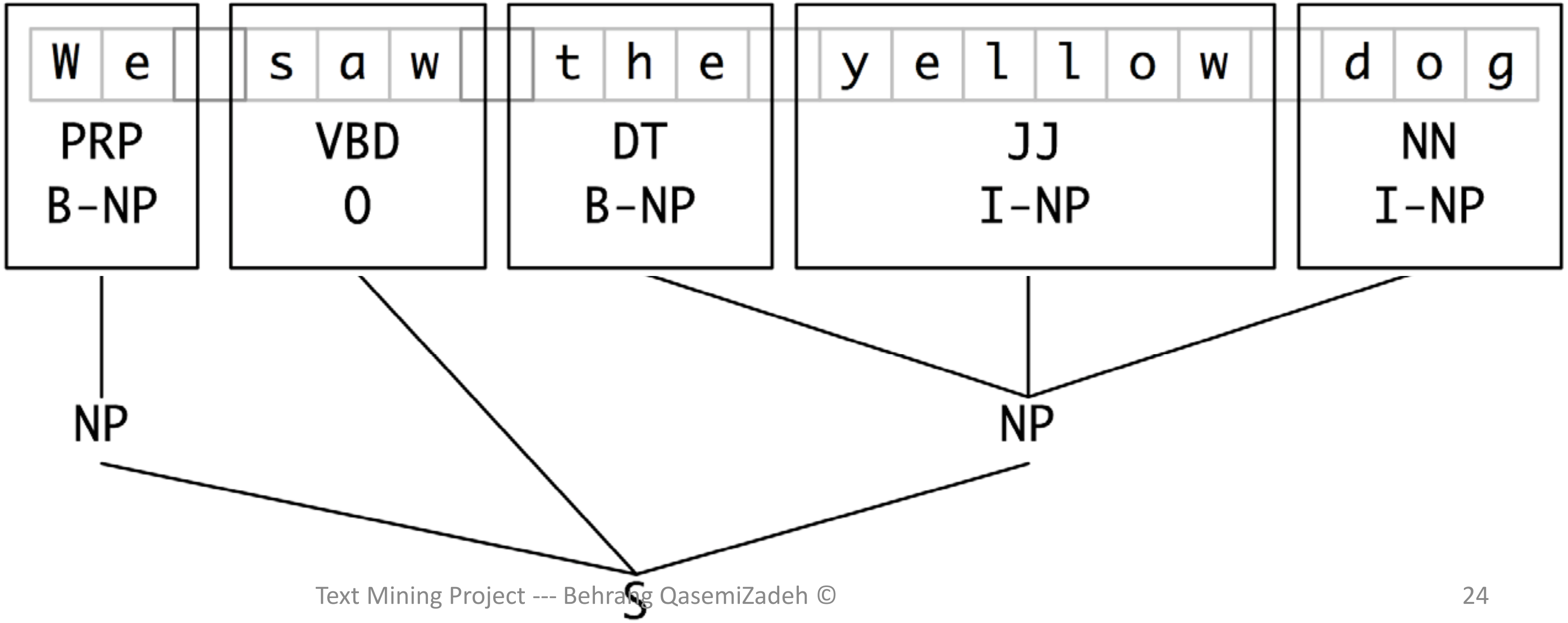
- Sometimes, it is easier to say what we do not want, instead of stating what we want!
- Chinking is the process of removing a sequence of tokens from a chunk:
  - We can alter the definition of chunk patterns to get rid of what we do not want.

```
grammar = r"""
      NP: {<.*>+} # Chunk everything
      }<VBD|IN>+{ # Chink sequences of VBD and IN """
```

Chunk Representation

- Chunks can be presented/seen both using tags and trees.
- However, the **IOB tags** are most common representation:
  - I (inside), O (outside), or B (begin).

We PRP B-NP
saw VBD O
the DT B-NP
yellow JJ I-NP
dog NN I-NP



Reading IOB Format in CoNLL 2000 Corpus

- The CoNLL 2000 corpus contains 270k words of Wall Street Journal text.
- The corpus is divided into "train" and "test" portions.
- Each part is annotated with part-of-speech tags and chunk tags in the IOB format.

Reading IOB Format in CoNLL 2000 Corpus

```
>>> from nltk.corpus import conll2000
>>> print(conll2000.chunked_sents('train.txt')[99])
(S
      (PP Over/IN)
      (NP a/DT cup/NN)
      (PP of/IN) (NP coffee/NN)
      ,/,
      (NP Mr./NNP Stone/NNP)
      (VP told/VBD)
      (NP his/PRP$ story/NN)
      ./.)
```

Reading IOB Format in CoNLL 2000 Corpus

```
>>> print(conll2000.chunked_sents('train.txt', chunk_types=['NP'])[99])
(S
      Over/IN
      (NP a/DT cup/NN)
      of/IN
      (NP coffee/NN)
      ,/,
      (NP Mr./NNP Stone/NNP)
      told/VBD
      (NP his/PRP$ story/NN)
      ./.)
```

Simple Evaluation and Baselines

- Let’s use CoNLL2000 … tion of the rule-based chunker we devel…

```
>>> from nltk.corp…
>>> cp = nltk.Regexp…
>>> test_sents = conll2…                      chunk_types=['NP'])
>>> print(cp.evaluate(te…
ChunkParse score:
      IOB Accuracy: 43.4%
      Precision: 0.0%
      Recall: 0.0%
      F-Measure: 0.0%
```

This means that 43% of words are tagged with O, i.e. not in an NP chunk!

But the tagger could not find any NP chunk so the precision and recall is 0.0!

Simple Evaluation and Baselines

- Let's try a simple regular expression pattern

```
>>> grammar = r"NP: {<[CDJNP].*>+}"
>>> cp = nltk.RegexpParser(grammar)
>>> print(cp.evaluate(test_sents))
ChunkParse score:
      IOB Accuracy: 87.7%
      Precision: 70.6%
      Recall: 67.8%
      F-Measure: 69.2%
```

Not too bad for a simple pattern, ha?!

Training Classifier-Based Chunkers

- Even if we define an elaborated set of pattern
still may not be the best method for chunking

Similar PoS sequence but different chunks!

Joey/NN sold/VBD the/DT farmer/NN rice/NN ./.

Nick/NN broke/VBD my/DT computer/NN monitor/NN ./.

- We can use data-driven techniques (similar to what we used for PoS tagger development) to develop chunkers.

```python
class ConsecutiveNPChunkTagger(nltk.TaggerI):
   def __init__(self, train_sents):
      train_set = []
      for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                  featureset = npchunk_features(untagged_sent, i, history)
                  train_set.append( (featureset, tag) )
                  history.append(tag)
            self.classifier = nltk.MaxentClassifier.train(
                  train_set, algorithm='megam', trace=0)

   def tag(self, sentence):
      history = []
      for i, word in enumerate(sentence):
            featureset = npchunk_features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
      return zip(sentence, history)
```

```
class ConsecutiveNPChunkTagger(nltk.TaggerI):
    def __init__(self, train_sents):
        train_set = []
        for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                featureset = npchunk_features(untagged_sent, i, history)
                train_set.append( (featureset, tag) )
                history.append(tag)
            self.classifier = nltk.MaxentClassifier.train(
                train_set, algorithm='megam', trace=0)

    def tag(self, sentence):
        history = []
        for
        
        retu
```

http://www.nltk.org/book/pylisting/code_classifier_chunker.py

```
class ConsecutiveNPChunkTagger(nltk.TaggerI):
   def __init__(self, train_sents):
      train_set = []
      for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                  featureset = npchunk_features(untagged_sent, i, history)
                  train_set.append( (featureset, tag) )
                  history.append(tag)
            self.classifier = nltk.MaxentClassifier.train(
                  train_set, algorithm='megam', trace=0)

   def tag(self, sentence):
      history = []
      for i, word in enumerate(sentence):
            featureset = npchunk_features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
      return zip(sentence, history)
```

indication of inheritance

```
class ConsecutiveNPChunkTagger(nltk.TaggerI):
   def __init__(self, train_sents):
      train_set = []
      for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                  featureset = npchunk_features(untagged_sent, i, history)
                  train_set.append( (featureset, tag) )
                  history.append(tag)
            self.classifier = nltk.MaxentClassifier.train(
                  train_set, algorithm='megam', trace=0)

   def tag(self, sentence):
      history = []
      for i, word in enumerate(sentence):
            featureset = npchunk_features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
      return zip(sentence, history)
```

class instantiation invokes __init__

```python
class ConsecutiveNPChunkTagger(nltk.TaggerI):
    def __init__(self, train_sents):
        train_set = []
        for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                featureset = npchunk_features(untagged_sent, i, history)
                train_set.append( (featureset, tag) )
                history.append(tag)
            self.classifier = nltk.MaxentClassifier.train(
                train_set, algorithm='megam', trace=0)

    def tag(self, sentence):
        history = []
        for i, word in enumerate(sentence):
            featureset = npchunk_features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
        return zip(sentence, history)
```

to provide the appropriate history to the feature extractor

```python
class ConsecutiveNPChunkTagger(nltk.TaggerI):
    def __init__(self, train_sents):
        train_set = []
        for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                featureset = npchunk_features(untagged_sent, i, history)
                train_set.append( (featureset, tag) )
                history.append(tag)
            self.classifier = nltk.MaxentClassifier.train(
                train_set, algorithm='megam', trace=0)

    def tag(self, sentence):
        history = []
        for i, word in enumerate(sentence):
            featureset = npchunk_features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
        return zip(sentence, history)
```

To indelicate instance of the object itself

```python
class ConsecutiveNPChunkTagger(nltk.TaggerI):
   def __init__(self, train_sents):
      train_set = []
      for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                  featureset = npchunk_features(untagged_sent, i, history)
                  train_set.append( (featureset, tag) )
                  history.append(tag)
            self.classifier = nltk.MaxentClassifier.train(
                  train_set, trace=0)

   def tag(self, sentence):
      history = []
      for i, word in enumerate(sentence):
            featureset = npchunk_features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
      return zip(sentence, history)
```

Use Maximum Entropy classifier

```python
class ConsecutiveNPChunkTagger(nltk.TaggerI):
    def __init__(self, train_sents):
        train_set = []
        for tagged_sent in train_sents:
            untagged_sent = nltk.tag.untag(tagged_sent)
            history = []
            for i, (word, tag) in enumerate(tagged_sent):
                featureset = npchunk_features(untagged_sent, i, history)
                train_set.append( (featureset, tag) )
                history.append(tag)
            self.classifier = nltk.MaxentClassifier.train(
                train_set, algorithm='megam', trace=0)

    def tag(self, sentence):
        history = []
        for i, word in enumerate(sentence):
            featureset = npchunk_features(sentence, i, history)
            tag = self.classifier.classify(featureset)
            history.append(tag)
        return zip(sentence, history)
```

And, this is the method to be used for chunking

Simple Feature Extraction

```
>>> def npchunk_features(sentence, i, history):
      word, pos = sentence[i]
      return {"pos": pos}

>>> chunker = ConsecutiveNPChunker(train_sents)

>>> print(chunker.evaluate(test_sents))
ChunkParse score:
      IOB Accuracy: 92.9%
      Precision: 79.9%
      Recall: 86.8%
      F-Measure: 83.2%
```

Only the part-of-speech tag of the current token

Simple Feature Extraction

```
>>> def npchunk_features(sentence, i, history):
      word, pos = sentence[i]
      if i == 0:
            prevword, prevpos = "<START>", "<START>"
      else:
            prevword, prevpos = sentence[i-1]
      return {"pos": pos, "word": word, "prevpos": prevpos}

>>> chunker = ConsecutiveNPChunker(train_sents)
>>> print(chunker.evaluate(test_sents))
ChunkParse score:
      IOB Accuracy: 94.5%
      Precision: 84.2%
      Recall: 89.4%
      F-Measure: 86.7%
```

Also, add the current word and the PoS of the previous wor

```
>>> def npchunk_features(sentence, i, history):
      word, pos = sentence[i]
       if i == 0:
            prevword, prevpos = "<START>", "<START>"
      else:
            prevword, prevpos = sentence[i-1]
       if i == len(sentence)-1:
            nextword, nextpos = "<END>", "<END>“
      else:
            nextword, nextpos = sentence[i+1]
      return {
      "pos": pos, "word": word,"prevpos": prevpos,
      "nextpos": nextpos, "prevpos+pos": "%s+%s" %(prevpos, pos),
      "pos+nextpos": "%s+%s" % (pos, nextpos),
      "tags-since-dt": tags_since_dt(sentence, i)}
```

Also, include more complex context features!

```
ChunkParse score:
      IOB Accuracy: 96.0%
      Precision: 88.6%
      Recall: 91.0%
      F-Measure: 89.8%
```

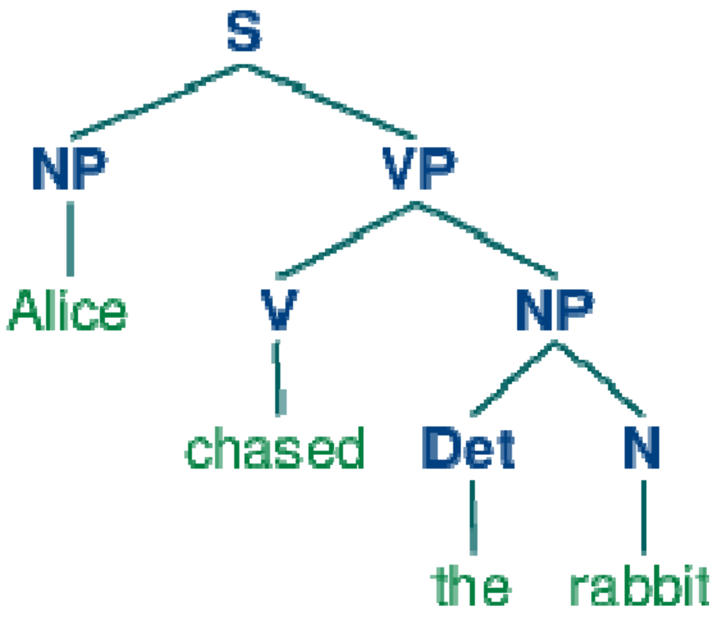
Nested Structure with Cascaded Chunkers

- It is possible to build chunk structures of arbitrary depth.
- To do so, we can use a multi-stage chunk grammar containing recursive rules:
  - For a chunker based on regular expressions, this means that we need to change our RegEx pattern:

```
grammar = r"""
      NP: {<DT|JJ|NN.*>+}     # Chunk sequences of DT, JJ, NN
      PP: {<IN><NP>} # Chunk prepositions followed by NP
      VP: {<VB.*><NP|PP|CLAUSE>+$} # Chunk verbs and their arguments
      CLAUSE: {<NP><VP>} # Chunk NP, VP
      """
```

Trees

- **Tree** a set of connected labelled nodes each reachable by a unique path from a distinguished root node, i.e. is an acyclic graph.
- Nodes are often referred to by terms **parent**, **child** and **sibling**.



Trees

- In NLTK, a tree is created using a node label and a list of children (list):

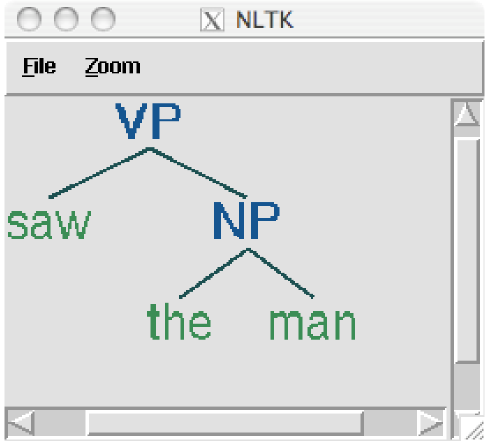
```
>>> tree1 = nltk.Tree('NP', ['Alice'])
>>> print(tree1)
(NP Alice)
>>> tree2 = nltk.Tree('NP', ['the', 'rabbit'])
>>> print(tree2)
(NP the rabbit)
```

Trees

- A tree of an arbitrary depth can then be created in a recursively:

```
>>> tree3 = nltk.Tree('VP', ['chased', tree2])
>>> tree4 = nltk.Tree('S', [tree1, tree3])
>>> print(tree4)
(S (NP Alice) (VP chased (NP the rabbit)))

>>> tree3.draw()
```



Tree Traversal

- An easy way to traverse a tree is to use a recursive function:

```
def traverse(t):
        try:
                t.label()
        except AttributeError:
                print(t, end=" ")
        else: # Now we know that t.node is defined
                print('(', t.label(), end=" ")
                for child in t:
                        traverse(child) print(')', end=" ")
>>> t = nltk.Tree('(S (NP Alice) (VP chased (NP the rabbit)))')
>>> traverse(t)
( S ( NP Alice ) ( VP chased ( NP the rabbit ) ) )
```

See how we check **t** is tree and encode the termination of the function with handling the exception !