University of Malta
BSc IT (Hons.) Year III

CSA305: Natural Language Processing

Lecture 6

Grammars and Algorithms

- Derivation
- Recognition as Search
- Top Down/Bottom Up Strategies
- Grammars and Logic
- Definite Clause Grammars

Department of Computer Science and AI
2000/01

Grammars and Algorithms

Any grammar G assigns structures to all and only the sentences in the language generated by G.

However, G does not tell us **how** to go about assigning such structures, or even how to generate the language of G. Each of these tasks is essentially procedural and require algorithmic rather than mathematical concepts for their description. Practical tasks of interest in NLP are

- **Recognition:** deciding whether a given sentence belongs to a language.
- **Parsing:** discovering the syntactic structure assigned by a grammar to a sentence.
- **Synthesis:** generating all the sentences of a language or having a given syntactic structure.

Derivation - Mathematical Definition

Let X be a non terminal symbol and let α, β, Y be (possibly empty) strings of terminal and non terminal symbols of the grammar.

With respect to a grammar G, the string $\alpha X\beta$ *immediately derives* the string $\alpha Y\beta$, written

$\alpha X\beta \Rightarrow_i^G \alpha Y\beta$

if $X \rightarrow Y \in$ G

With respect to a grammar G, the string $\alpha X\beta$ *derives* the string $\alpha Z\beta$, written

$\alpha X\beta \Rightarrow_*^G \alpha Z\beta$

if either $\alpha X\beta \Rightarrow_i^G \alpha Z\beta$

or if $\alpha X\beta \Rightarrow_i^G Y$ and $Y \Rightarrow_*^G \alpha Z\beta$

Recognition

- A recogniser must establish whether or not a given sentence has a derivation.
- We can regard the discovery of such a derivation as a classic search problem characterised by (a) a well-defined start state (b) a well defined goal state and (c) operators (grammar rules and lexical entries).
- Various recognition strategies can be adopted.
  - **Top Down**: in which we attempt to synthesise the sentence by progressively rewriting category symbols until we reach words
  - **Bottom Up**: in which we start with the words and attempt to reduce the sequence of symbols to S – this time using the grammar rules backwards.'

Examples of Recognition Strategies

Top Down Strategy	Bottom Up Strategy
S	John ate the cat
NP VP	n ate the cat
n VP	n v the cat
John VP	n v d cat
John v NP	n v d n
John ate NP	NP v d n
John ate d n	NP v NP
John ate the N	NP VP
John ate the cat	S

The stategy chosen changes the search problem:

Stategy	Start State	Goal State
Top Down	S	Words
Bottom Up	Words	S

Bottom up v. Top Down

- Top down never explores derivations that could never lead to an S, unlike bottom up.
- Top down can explore derivations that are not consistent with the input, unlike bottom up.
- Top down and bottom up are extremes. It is possible to envisage mixed strategies which begin by operating bottom up to get the categories of all lexical items, and then top down to get the higher categories.
- Indexing of rules depends upon strategy:
  - Top down, index rules by their LHS category
  - Bottom up, index rules by leftmost RHS category (for left to right parsing)

Bottom Up v. Top Down

In general

- We prefer to parse a category bottom up if there are many rules that rewrite that category, and the indexing categories on the RHS of those rules index few rules.
- We prefer to parse a category top down if there are few rules that rewrite that category and the RHS of those rules contain indexing categories that index many rules.

Top Down Strategy: Parse States

- Starts with the S symbol and rewrites it into a sequence of terminal symbols matching the word classes of the input (in our example we have gone further and selected words).
- The **parse state** can be represented as a pair comprising a **symbol list** and a **text position**. For instance given the sentence

  0 The 1 dogs 2 cried 3
  ↑

  the initial state would be

  ((S) 0)

  indicating that in order to recognise a sentence, then starting at position 0, is necessary to find an S. Since S is not a lexical category, we rewrite it S according to the grammar rules to obtain

  ((NP VP) 0)

  and subsequently (assuming NP $\rightarrow$ d n $\in$ G)

  ((d n VP) 0)

Top Down Recognition

((d n VP) 0)

At this point the leftmost symbol is a **lexical** category, so we can check against the next input word via the lexicon.

If the the next word has a matching lexical category, the next word in the input is consumed

$$\begin{array}{ccccccc} 0 & \text{The} & 1 & \text{dogs} & 2 & \text{cried} & 3 \\ & & \uparrow & & & & \end{array}$$

and the parse state is updated as follows:

((n VP) 1)

i.e. to succeed we must find an n followed by a VP starting at position 1. In this case the next state is as follows

((VP) 2)

$$\begin{array}{ccccccc} 0 & \text{The} & 1 & \text{dogs} & 2 & \text{cried} & 3 \\ & & & & \uparrow & & \end{array}$$

Backtracking

When the leftmost symbol of the symbol list is **non-lexical**, there may be several applicable grammar rules, and therefore several possible new states, all of which might have to be explored to effect successful recognition. One way to achieve this is through **backtracking**.

- Generate all possible successor states from a given state.
- One of these is chosen; the rest are kept as backup states.
- Whenever a given non-success state has no successors, a new choice is made from the backup states.

N.B. A state is successful when it consists of the pair ((),m) where m is the length of the input sentence.

Top Down Parsing Algorithm

Algorithm manipulates a queue Q of possible states with the following shape

$$(C, B_1, B_2, \ldots)$$

where C is the current state and the B_i are backup states. In the following description we assume a sentence containing n words.

1. INITIALIZE :

- Q ← (((S),0))

2.1 CHOOSE CURRENT :

- IF empty(Q) **EXIT WITH FAILURE**
- ELSE **C ← pop(Q)**

2.2. TEST-SUCCESS :

- IF C = (((),n)) **EXIT WITH SUCCESS**
- ELSE
  - **EXPAND C**
  - **ENQUEUE**

REPEAT step 2

How to Expand and Enqueue a State

- **Expansion**

  State has shape (σ, P) where σ is a symbol list $\sigma_1 \ldots \sigma_{n-1}, \sigma_n$ and P is a text position. Assume that w(P) is the word at that text position.

  - IF σ_1 is lexical AND w(P) can be of that lexical category, new state is

    $$(pop(\sigma), P+1)$$

  - IF σ_1 is phrasal, for each rule in the grammar of the form

    $$\sigma_1 \rightarrow \psi$$

    that can rewrite it, generate a new state

    $$(\psi + pop(\sigma), P)$$

    where "+" is the concatenation operation

- **Enqueuing**
  - To achieve depth-first behaviour, all states are added **at the front** of the queue.
  - To achieve breadth-first behaviour, all states are **at the back** of the queue.

Depth First Recognition of *0 John 1 hit 2 Bill 3*

Step	C	Backups	Comment
1.	((S) 0)		
2.	((NP VP) 0)		Expand S
3.	((n VP)) 0)		Expand NP
		((d n VP) 0)	
		((NP PP VP) 0)	
4.	((VP) 1)		Consume n (John)
5.	((v) 1)		Expand VP
		((v NP) 1)	
		((v NP NP) 1)	
		((VP PP) 1)	
		((d n VP) 0)	
		((NP PP VP) 0)	
6.	(() 2)		Consume v (hit)
7.	((v NP) 1)		Trying next poss.
		((v NP NP) 1)	
		((VP PP) 1)	
		((d n VP) 0)	
		((NP PP VP) 0)	
8.	((NP) 2)		Consume v
9.	((n) 2)		Expand NP
		((d n) 2)	
		((NP PP) 2)	
		((v NP NP) 1)	
		((VP PP) 1)	
		((d n VP) 0)	
		((NP PP VP) 0)	
10.	(() 3)		Consume n; succee

Depth First vs. Breadth First Strategies

- **Depth First**
  - Advantages: Space Efficient
  - Disadvantages: Prone to looping on left recursive rules, eg *NP* → *NP PP*
  - Cure: Prohibit L. recursive rules
- **Breadth First**
  - Advantages: Handles left recursion.
  - Disadvantages: Space inefficient. Can loop on ungrammatical sentences since number of parse states may be infinite if recursive rules are allowed.
  - Cure: Prohibit L. recursive rules. Impose depth limit. Use more information to avoid generating useless states.

Grammars and Logic Programming: From Grammar Rules To Clauses

Any rewrite rule containing only phrasal categories on the RHS such as

S → *NP VP*

can be regarded as kind of axiom stating that a sequence of words is an S if it consists of a legal NP followed by a legal VP.

Using the language of **text positions**:

```
p0  W  ...  W   p1  W  ...  W  p
↑               ↑              ↑
```

there is an S between p0 and p if there is an NP between p0 and p1 and a VP between p1 and p

$$\forall\ p_0\ p_1\ p\quad NP(p_0,p_1) \wedge VP(p_1,p) \supset S(p_0,p)$$

s(P0,P) :- np(P0,P1), vp(P1,P).

Converting a Complete Grammar

Grammar

S	→	*NP VP*
NP	→	*n*
NP	→	*d n*
NP	→	*NP PP*
VP	→	*v*
VP	→	*v NP*
VP	→	*v NP NP*
VP	→	*VP PP*
PP	→	*p NP*

Clauses

```
s(P0,P)    :-  np(P0,P1), vp(P1,P).
np(P0,P)   :-  n(P0,P).
np(P0,P)   :-  d(P0,P1), n(P1,P).
np(P0,P)   :-  np(P0,P1), pp(P1,P).
vp(P0,P)   :-  v(P0,P).
vp(P0,P)   :-  v(P0,P1), np(P1,P).
vp(P0,P)   :-  v(P0,P1), np(P1,P2),
               np(P2,P).
vp(P0,P)   :-  vp(P0,P1), pp(P1,P).
pp(P0,P)   :-  p(P0,P1), np(P0,P).
```

Rules for Lexical Categories: The Lexicon

Rules for lexical categories always derive exactly one token in the input (normally a word, but could be punctuation, whitespace etc.). Such a rule must express the following:

There is an n (d,v,p ...) between positions p0 and p if some word of syntactic category n spans those positions.

One way to deal with this is first to define (for a given input) the predicate `input(P1,P2,L)` such that P1 and P2 are positions and L is a list containing the words spanning those positions. So for the sentence *John ate the cat* we could expect

```
input(0,1,['John']).
input(1,2,[ate]).
etc
```


Rules for Lexical Categories

Then each lexical category is defined in terms of the words it can rewrite.

```
d(P1,P)  :-  input(P1,P,[the]).
n(P1,P)  :-  input(P1,P,[cat]).
n(P1,P)  :-  input(P1,P,['John']).
v(P1,P)  :-  input(P1,P,[ate]).
```

This corresponds to an implementation of the lexicon for which for every word w of category c we have a rule of the form.

c(P1,P) :- input(P1,P,[w]).

A Complete Program

Grammar

```
s(P0,P)   :-  np(P0,P1), vp(P1,P).
np(P0,P)  :-  n(P0,P).
np(P0,P)  :-  d(P0,P1), n(P1,P).
vp(P0,P)  :-  v(P0,P).
vp(P0,P)  :-  v(P0,P1), np(P1,P).
vp(P0,P)  :-  v(P0,P1), np(P1,P2),
              np(P2,P).
```

Lexicon

```
d(P1,P)  :-  input(P1,P,[the]).
n(P1,P)  :-  input(P1,P,[cat]).
n(P1,P)  :-  input(P1,P,['John']).
v(P1,P)  :-  input(P1,P,[ate]).
```

input

```
input(0,1,['John']).
input(1,2,[ate]).
input(2,3,[the]).
input(3,4,[cat]).
```

Query

```
?- s(0,4).
```


Execution Trace

```
?- trace.
?- vp(1,4).
  N  R  P     G                          N = goal i/d
  1  1  Call: vp(1,4) ?                  R = recursion depth
  2  2  Call: v(1,4) ?                   P = execution state
  3  3  Call: input(1,4,[ate]) ?         G = goal
  3  3  Fail: input(1,4,[ate]) ?
  2  2  Fail: v(1,4) ?
  2  2  Call: v(1,_349) ?
  3  3  Call: input(1,_349,[ate]) ?
  3  3  Exit: input(1,2,[ate]) ?
  2  2  Exit: v(1,2) ?
  4  2  Call: np(2,4) ?
  5  3  Call: n(2,4) ?
  6  4  Call: input(2,4,[cat]) ?
  6  4  Fail: input(2,4,[cat]) ?
  6  4  Call: input(2,4,[John]) ?
  6  4  Fail: input(2,4,[John]) ?
  5  3  Fail: n(2,4) ?
  5  3  Call: d(2,_1338) ?
  6  4  Call: input(2,_1338,[the]) ?
  6  4  Exit: input(2,3,[the]) ?
  5  3  Exit: d(2,3) ?
  7  3  Call: n(3,4) ?
  8  4  Call: input(3,4,[cat]) ?
  8  4  Exit: input(3,4,[cat]) ?
  7  3  Exit: n(3,4) ?
  4  2  Exit: np(2,4) ?
  1  1  Exit: vp(1,4) ?
```

DCG Notation

1. **Grammar:**

```
s    -->  np, vp.
np   -->  n.
np   -->  d, n.
vp   -->  v.
vp   -->  v, np.
vp   -->  v, np, np.
```

– Special rewrite arrow causes automatic insertion of position arguments.

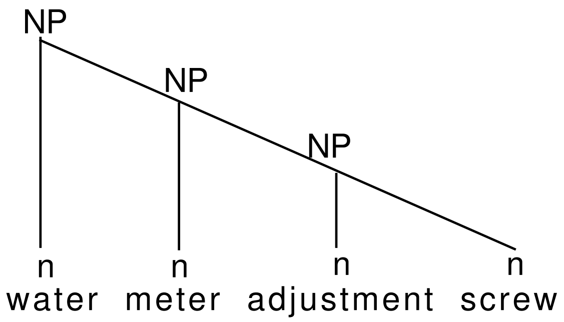
– To handle left recursion either:

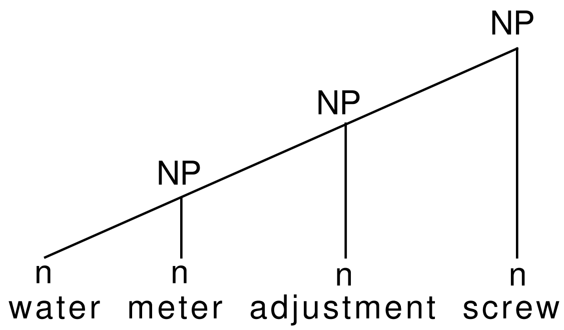
  * Use a different parsing algorithm (Prolog is top down depth first).

  * Use a different grammar.

```
np --> n              np -> n
np --> np n   versus  np -> n np
```


Left and Right Branching Structures





DCG Notation

- **2. Lexicon**

  The lexicon is represented as a series of rewrites with the following shape:

```
n  -->  ['John'].
n  -->  [cat].
d  -->  [the].
v  -->  [ate].
```

- **3. Query**

  – The text is represented as a list of words, e.g. ['John',ate,the,cat].

  – Instead of representing positions using a pair of integers, we use a pair of **lists** whose **difference** represents the segment of text in question,

  Hence the initial query takes the form

  s(['John',ate,the,cat],[]).

Difference Lists

We can represent a list as a pair of pointers:

- The first pointer (P1) points to the entire list.
- The second (P2) points to a suffix of the list, including the degenerate case of the list itself.
- The represented list is the difference between the two lists.

So the list `[a,b,c]`

can be represented using any of the following pairs where P2 is a suffix of P1.

P1	P2
[a,b,c,d,e]	[d,e]
[a,b,c,d]	[d]
[a,b,c]	[]

The Lexicon Revisited

Currently information about words takes the form of rewrite rules for lexical categories, e.g.

```
n --> [cat].
```

It is often convenient to separate the descriptions of individual words from the grammar rules as follows.

1. Introduce a new 1-place predicate, eg `noun(X)` that will hold of all nouns.
2. Replace all rules of the above form with the general rule

   `n --> [X], { noun(X) }.`
3. Provide a definition for the newly introduced predicate, e.g.

   `noun(X) :- lex(X,noun).`

   where `lex` is the user interface to the lexicon.
4. Define the lexicon using assertions about words.

   ```
   lex(cat,noun).
   lex('John',noun).
   ```


The Complete DCG

Grammar

```
s   -->  np, vp.
np  -->  n.
np  -->  d, n.
vp  -->  v.
vp  -->  v, np.
vp  -->  v, np, np.
n   -->  [X], { noun(X) }.
d   -->  [X], { det(X). }.
v   -->  [X], { verb(X).}.
p   -->  [X], { prep(X) }.
```

Lexicon

```
lex(the,det).
lex(cat,noun).
lex('John',noun).
lex(ate,verb).
```

Query

```
?- s(['John',ate,the,cat],[]).
```

Lexicon Interface

```
noun(X)  :-  lex(X,noun).
verb(X)  :-  lex(X,verb).
det(X)   :-  lex(X,det).
prep(X)  :-  lex(X,prep).
```