Top-down parsing

- We start from the root non-terminal for the grammar
- Successively pick and apply derivation rules for non-terminals that still need to be expanded
- End when we have expanded all non-terminals and the derived token sequence matches the input sequence
- If no valid expansion rule exists or the token sequences do not match then back up to the most recent choice point where an alternate choice is still possible, and try that derivation path
- Fails if we run out of derivation paths without finding a match for the token sequence

Generic top down algorithm

```
current = root nonterminal
stack = <NONE>
repeat
    if current is a nonterminal
        pick one of the derivation rules current->x1 x2 ... xn
        push xn..x2 onto stack (right-to-left) and set current = x1
    else if current matches next input token
        pop top of stack into current and read next input token
    else backtrack
if stack empty and input has all been read then accept, otherwise reject
```

Example derivation sequence

- Permitted tokens are: a b c d e
- Grammar rules are:
- S --> XY
- X --> abc | eXe
- Y --> cY | d
- String we are trying to match is: eabceccd

Derivation: part 1

current	nexttoken	rest of input	stack (top on left)	notes
S	e	abceccd	-	pick S-->XY
X	e	abceccd	Y-	pick X-->eXe
e	e	abceccd	XeY-	matches
X	a	bceccd	eY-	Pick X-->abc
a	a	bceccd	bceY-	matches
b	b	ceccd	ceY-	matches
c	c	eccd	eY-	matches

Derivation: part 2

e	e	ccd	Y-	matches
Y	c	cd	-	pick Y-->cY
c	c	cd	Y-	matches
Y	c	d	-	pick Y-->cY
c	c	d	Y-	matches
Y	d	-	-	pick Y-->d
d	d	-	-	matches
-	-	-	-	ACCEPT

Backtrack-free grammars

- In fact, many grammars can be parsed without the need for backtracking, particularly if we allow the parser to look at the upcoming token in the input sequence (lookahead)
- Sometimes we can rewrite our existing grammar to put it in a backtrack-free form
- For grammars that cannot eliminate backtracking, we can still consider implementations to reduce its cost

Left recursion

- Left recursive rules have form X --> X*something*
- Parser can get locked into infinite loop of accepting this same rule, trying to expand leftmost nonterminal
- Can also have indirect left recursion
  - W-->X
  - X-->Y
  - Y-->WZ

Consider
W-->X-->Y-->WZ-->XZ-->YZ-->WWZ--> etc

- Want to eliminate both direct and indirect left recursion from our grammar

Eliminate direct left recursion

- Done by introducing extra nonterminal, here N
- X --> X b | a
- (i.e. X describes ab*)
- X --> a N
- N --> b N | nil
- (here using nil to represent empty sequence, which gets rid of the final trailing N in each derivation)

Eliminating indirect left recursion

- Works on grammars that don't have cycles or nil rules
- Give all the non-terminals an arbitrary ordering, 1..k
- For i = 1..k
  - For j = 1..(i-1)
    - For each rule rule Ni->NjX (remember j<i) replace it with
      - Ni->Y1X | Y2X | Y3X ... where Y1, Y2, Y3 are RHS of Nj rules,
      - i.e. The grammar has rules Nj->Y1 | Y2 | Y3 | ...
  - After inner for loop, eliminate any direct left recursion on Ni
- Done! Essentially inserting rules to bypass the recursive rules, replacing them with ones to generate the results

Predictive grammars

- Used for backtrack-free parsing
- When picking which rule to apply next you get to look at next input token, and select from just those rules whose RHS can generate that token
- If a predictive grammar is backtrack free then it can be efficiently parsed using recursive parsers (which we'll look at in our next session)
- Unfortunately, not all grammars are backtrack free