Introduction to Parsing

Data Structures and Algorithms for Computational Linguistics III
(ISCL-BA-07)

Çağrı Çöltekin
ccoltekin@sfs.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2025/26

version: f7b8ee1 @2026-01-19

What is parsing?

- *Parsing* is the task of analyzing a string of symbols to discover its (inherent) structure
- Typically, the structure (and the valid strings in the language) is defined by a *grammar*
- The output of a parser is a structured representation of the input string, often a tree
- *Recognition* is an intimately related task which determines whether a given string is in a language

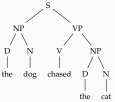
Ingredients of a parser

(for natural language parsing)

- A formal grammar defining a language of interest
- An algorithm that (efficiently) verifies whether a given string is in the language (recognizer) and enumerates the grammar rules used for verification (parser)
- A system for ambiguity resolution (not in this course)

Grammars

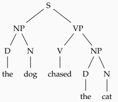
- A grammar is a finite specification of a possibly infinite language
- The most commonly studied type of grammars are *phrase structure grammars*
- Analysis using context-free grammars result in *constituency* or *phrase structure* trees

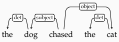


S → NP VP   NP → D N   VP → V NP
V → chased   D → the   N → cat   N → dog

Why study parsing?

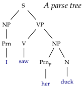
- In general, it is an intermediate step for interpreting sentences
- Applications include:
  - Compiler construction
  - Grammar checking
  - Sentiment analysis
  - Information (e.g., relation) extraction
  - Argument mining
  - ...





Different ways to represent a context-free parse

A parse tree



A history of derivations

Sentential form	derivation
S	(start)
NP VP	S ⇒ → NP VP
Prn VP	NP ⇒ → Prn
I VP	Prn ⇒ → I
I V NP	VP ⇒ → V NP
I saw NP	V ⇒ → saw
I saw Prn_p N	NP ⇒ → Prn_p N
I saw her N	Prn_p ⇒ → her
I saw her duck	N ⇒ → duck

(*Labeled*) *brackets*: $\left[_{S}\left[_{NP}[_{Prn}\text{ I}]\right]\left[_{VP}[_{V}\text{ saw}]\,[_{NP}[_{Prn_p}\text{ her}]\,[_{N}\text{ duck}]]\right]\right]$

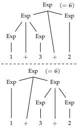
Relation between different representations

- The parse tree and the bracket representation is equivalent
  - parse trees are easier to read by humans
  - brackets are easier for computers
  - brackets are the typical representation for treebanks
- A parse tree (or bracket representation) can be obtained with a different order of production rules

Grammars and ambiguity

Exp → n
Exp → Exp + Exp
(terminal symbol 'n' stands for any number)

- If a grammar is ambiguous, some sentences produce multiple analyses
- If the resulting analysis lead to the same semantics, the ambiguity is *spurious*

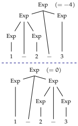


Grammars and ambiguity

Exp → n
Exp → Exp − Exp

(terminal symbol 'n' stands for any number)

- Is this ambiguity spurious?
- If different structures yield different semantics, the ambiguity is *essential*



Ambiguity can be removed from a grammar

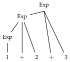
if the language is not ambiguous

Exp → n
Exp → Exp + n
(terminal symbol 'n' stands for any number)

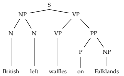
- The grammar above does not have the ambiguity of

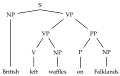
  Exp → n
  Exp → Exp + Exp

- Both grammars define the same language



Natural languages are ambiguous





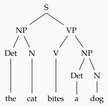
- The grammars we define have to distinguish between two different structures
- We need methods for ranking analyses

Top-down parsing

general idea

- Start from S, find a sequence of derivations that yield the sentence
- This is simply the same as the generation procedure we discussed earlier
- Attempt to generate all strings from a grammar, but allow only the productions that 'produce' the input string

Top-down: demonstration



S	→	NP VP
NP	→	Det N
VP	→	V NP
VP	→	V
Det	→	a
Det	→	the
N	→	cat
N	→	dog
V	→	bites

From demonstration to parsing

- There may be multiple productions applicable
- We need an automatic mechanism to select the correct productions
- We have two actions:
  - predict generate a hypothesis based on the grammar
  - match when a terminal symbol is produced, check if it matches with the one in the expected position
    - if matched, continue
    - otherwise, backtrack
- if we eliminate all non terminals from the sentential form, and the complete input string is matched (produced), then parsing successful

Top-down parsing: another demonstration

the grammar

S	→	NP VP
NP	→	Det N
VP	→	V NP
VP	→	V
Det	→	a
Det	→	the
N	→	cat
N	→	dog
V	→	bites

parse: *the cat bites a dog*

matched	goal	production
	S	S ⇒ NP VP
	NP VP	NP ⇒ Det N
	Det N VP	Det ⇒ a ✗
	Det N VP	Det ⇒ the ✓
the	N VP	N ⇒ dog ✗
the cat	N VP	N ⇒ cat ✓
the cat	VP	VP ⇒ V
the cat bites	V	V ⇒ bites ✓
the cat bites		(not at the end) ✗
the cat	VP	VP ⇒ V NP
the cat	V NP	V ⇒ bites ✓
the cat bites	Det N	NP ⇒ Det N
the cat bites a	N	Det ⇒ a ✓
the cat bites a dog		Det ⇒ dog ✓

Note that the valid productions yield the parse tree.

Top-down parsing: problems and possible solutions

- The trial-and-error procedure leads to exponential time parsing
- But lots of repeated work: dynamic programming may help avoid it
- What happens if we had a rule like

  NP → NP PP

  some rules may cause infinite loops
- Notice that if we knew which terminals are possible as the initial part of a non-terminal symbol, we can eliminate the unsuccessful matches earlier

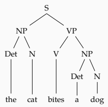
Bottom-up parsing

general idea

- Start from from the input symbols, and try to *reduce* the input to start symbol
- We need to match parts of the sentential form (starting from the input) to the RHS of the grammar rules
- While top-down process relies on *productions* the bottom-up process relies on *reductions*

production ↓						reduction ↑
	NP		V	NP		
	Det	N	V	Det	N	
	the	cat	bites	a	dog	

Bottom-up: demonstration



S	→	NP VP
NP	→	Det N
VP	→	V NP
VP	→	V
Det	→	a
Det	→	the
N	→	cat
N	→	dog
V	→	bites

A (first) introduction to shift-reduce parsing

- We keep two data structures:
  - a stack for the (partially) reduced sentential form
  - an input queue that contains only terminal symbols

  [NP V] [a dog]
- We use two operations:
  - shift shifts a terminal to stack

    [NP V] [a dog] —shift→ [NP V a] [dog]
  - reduce when top symbols on stack mach a RHS, replace them with the LHS of the rule

    [NP V] [a dog] —reduce→ [NP VP] [a dog]

Shift-reduce (bottom-up) parsing a demonstration

stack	input	rule
	the cat bites a dog	shift
the	cat bites a dog	Det ⇒ the
Det	cat bites a dog	shift
Det cat	bites a dog	N ⇒ cat
Det N	bites a dog	NP ⇒ Det N
NP	bites a dog	shift
NP bites	a dog	V ⇒ bites
NP V	a dog	VP ⇒ V
NP VP	a dog	S ⇒ NP VP
S	a dog	shift
S a	dog	Det ⇒ A
S Det dog		N ⇒ dog
S Det N		NP ⇒ Det N
S NP		(stuck)

stack	input	rule
NP V	a dog	shift
NP V a	dog	Det ⇒ a
NP V Det	dog	shift
NP V Det dog		N ⇒ dog
NP V Det N		NP ⇒ Det N
NP V NP		VP ⇒ V NP
NP VP		S ⇒ NP VP
S		(done)

- All input reduced to S, accept
- Rules form the parse tree

Summary

- Parsing can be formulated as a top-down or bottom-up search (the search may also be depth-first or breadth first)
- Naive parsing algorithms are inefficient (exponential time complexity)
- There are some directions: dynamic programming, filtering
- Suggested reading (for constituency parsing): Jurafsky and Martin (2009, draft 3rd ed, chapters 12 & 13)
- A general reference for parsing: Grune and Jacobs (2007)

Next:

- Bottom-up chart parsing: CKY algorithm
- Suggested reading: Jurafsky and Martin (2009, draft 3rd ed, section 13.2)

blank

blank

blank

blank