Introduction to Parsing

Detmar Meurers: Intro to Computational Linguistics I
OSU, LING 684.01, February 5., 10. and 12, 2003

Overview

- What is a parser?
- Under what criteria can they be evaluated?
- Parsing strategies
  - top-down vs. bottom-up
  - left-right vs. right-left
  - depth-first vs. breadth-first
- Implementing different types of parsers:
  - Basic top-down and bottom-up
  - More efficient algorithms

Parsers and criteria to evaluate them

- Function of a parser:
  - grammar + string $\longrightarrow$ analysis trees
- Main criteria for evaluating parsers:
  - correctness
  - completeness
  - efficiency

Correctness

A parser is **correct** iff for every grammar and for every string, every analysis returned by parser is an actual analysis.

Correctness is nearly always required (unless simple post-processor could eliminate wrong analyses)

Completeness

A parser is **complete** iff for every grammar and for every string, every correct analysis is found by the parser.

- In theory, always desirable.
- In practice, essential to find the 'relevant' analysis first (possibly using heuristics).
- For grammars licensing an infinite number of analyses this means: there is no analysis that the parser could not find.

Efficiency

- One can reason about complexity of (parsing) algorithms by considering how it will deal with bigger and bigger examples.

- For practical purposes, the factors ignored by such analyses are at least as important.
  - profiling using typical examples important
  - finding the (relevant) first parse vs. all parse

- Memoization of complete or partial results is essential to obtain efficient parsing algorithms.

Complexity classes

If n is the length of the string to be parsed, one can distinguish the following complexity classes:

- **constant**: amount of work does not depend on n
- **logarithmic**: amount of work behaves like $log_k(n)$ for some constant k
- **polynomial**: amount of work behaves like n^k, for some constant k. This is sometimes subdivided into the cases
  - **linear** $(k = 1)$
  - **quadratic** $(k = 2)$
  - **cubic** $(k = 3)$
  - ...
- **exponential**: amount of work behaves like k^n, for some constant k.

Complexity and the Chomsky hierarchy

Grammar type	Worst-case complexity of recognition
regular (3)	linear
context-free (2)	cubic (n^3)
context-sensitive (1)	exponential
general rewrite (0)	undecidable

Recognition with type 0 grammars is **recursively enumerable**: if a string x is in the language, the recognition algorithm will succeed, but it will not return if x is not in the language.

Parsing strategies

1. What do we start from?
   - top-down vs. bottom-up
2. In what order is the string or the RHS of a rule looked at?
   - left-to-right, right-to-left, island-driven, . . .
3. How are alternatives explored?
   - depth-first vs. breadth-first

Direction of processing I: Top-down

Goal-driven processing is Top-down:

- Start with the start symbol
- Derive sentential forms.
- If the string is among the sentences derived this way, it is part of the language.

Direction of processing II: Bottom-up

Data-driven processing is Bottom-up:

- Start with the sentence.
- For each substring σ of each sentential form $\alpha\sigma\beta$, find each grammar rule $N \rightarrow \omega$ to obtain all sentential forms $\alpha N \beta$.
- If the start symbol is among the sentential forms obtained, the sentence is part of the language.

Problem: Epsilon rules ($N \rightarrow \epsilon$).

The order in which one looks at a RHS

Left-to-Right

- Use the leftmost symbol first, continuing with the next to its right

Problem for top-down, left-to-right processing: left-recursion

For example, a rule like N' $\rightarrow$ N' PP leads to non-termination.

How are alternatives explored? I. Depth-first

- At every choice point: Pursue a single alternative completely before trying another alternative.
- State of affairs at the choice points needs to be remembered. Choices can be discarded after unsuccessful exploration.
- Depth-first search is not necessarily complete.

How are alternatives explored? II. Breadth-first

- At every choice point: Pursue every alternative for one step at a time.
- Requires serious bookkeeping since each alternative computation needs to be remembered at the same time.
- Search is guaranteed to be complete.

Compiling and executing DCGs in Prolog

- DCGs are a grammar formalism supporting any kind of parsing regime.
- The standard translation of DCGs to Prolog plus the proof procedure of Prolog results in a parsing strategy which is
  - top-down
  - left-to-right
  - depth-first

Implementing parsers

- Data structures: a parser configuration
- Top-down parsing
  - formal characterization
  - Prolog implementation
- Bottom-up parsing
  - formal characterization
  - Prolog implementation
- Towards more efficient parsers:
  - Left-corner
  - Remembering subresults

An example grammar (parser/simple/grammar.pl)

```
% defining grammar rule operator
:- op(1100,xfx,'--->').

% lexicon:
vt  ---> [saw].
det ---> [the].
det ---> [a].
n   ---> [dragon].
n   ---> [boy].
adj ---> [young].
```

```
% syntactic rules:
s  ---> [np, vp].
vp ---> [vt, np].
np ---> [det, n].
n  ---> [adj, n].
```

A parser configuration

Assuming a left-to-right order of processing, a **configuration** of a parser can be encoded by a pair of

- a stack as auxiliary memory
- the string remaining to be recognized

More formally, for a grammar $G = (N, \Sigma, S, P)$, a parser configuration is a pair $< \alpha, \tau >$ with $\alpha \in (N \cup \Sigma)^*$ and $\tau \in \Sigma^*$

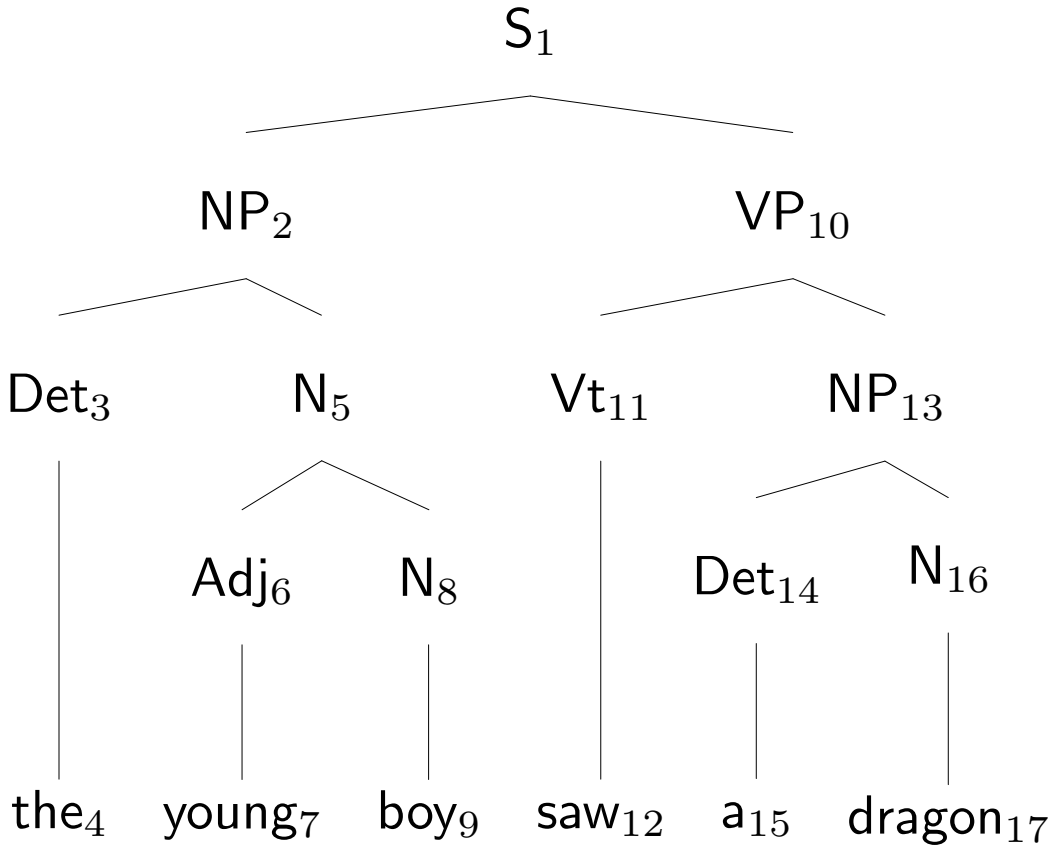
Top-down parsing

- **Start configuration** for recognizing a string ω: $< S, \omega >$
- **Available actions**:
  - **consume**: remove an expected terminal a from the string
    $< a\alpha, a\tau > \mapsto < \alpha, \tau >$
  - **expand**: apply a phrase structure rule
    $< A\beta, \tau > \mapsto < \alpha\beta, \tau >$ if $A \rightarrow \alpha \ \in P$
- **Success configuration**: $< \epsilon, \epsilon >$

A top-down parser in Prolog (parser/simple/td_parser.pl)

```
:- op(1100,xfx,'--->').

% Start
td_parse(String) :- td_parse([s],String).

% Success
td_parse([],[]).

% Consume
td_parse([H|T],[H|R]) :-
        td_parse(T,R).

% Expand
td_parse([A|Beta],String) :-
        (A ---> Alpha),
        append(Alpha,Beta,Stack),
        td_parse(Stack,String).
```

Top-Down, left-right, depth-first tree traversal



S → NP VP
VP → Vt NP
NP → Det N
N → Adj N

Vt → saw
Det → the
Det → a
N → dragon
N → boy
Adj → young

Bottom-up parsing

- **Start configuration** for recognizing a string ω: $< \epsilon, \omega >$
- **Available actions**:
  - **shift**: turn to the next terminal a of the string
    $< \alpha, a\tau > \mapsto < \alpha a, \tau >$
  - **reduce**: apply a phrase structure rule
    $< \beta\alpha, \tau > \mapsto < \beta A, \tau >$ if $A \rightarrow \alpha \ \in P$
- **Success configuration**: $< S, \epsilon >$

A shift-reduce parser in Prolog (parser/simple/sr_parser.pl)

```
:- op(1100,xfx,'--->').

sr_parse(String) :- sr_parse([],String).    % Start

sr_parse([s],[]).                           % Success

sr_parse(Stack,String) :-                   % Reduce
        append(Beta,Alpha,Stack),
        (A ---> Alpha),
        append(Beta,[A],NewStack),
        sr_parse(NewStack,String).

sr_parse(Stack,[Word|String]) :-            % Shift
        append(Stack,[Word],NewStack),
        sr_parse(NewStack,String).
```

A trace (parser/simple/grammar.pl, parser/simple/sr_parser_trace.pl)

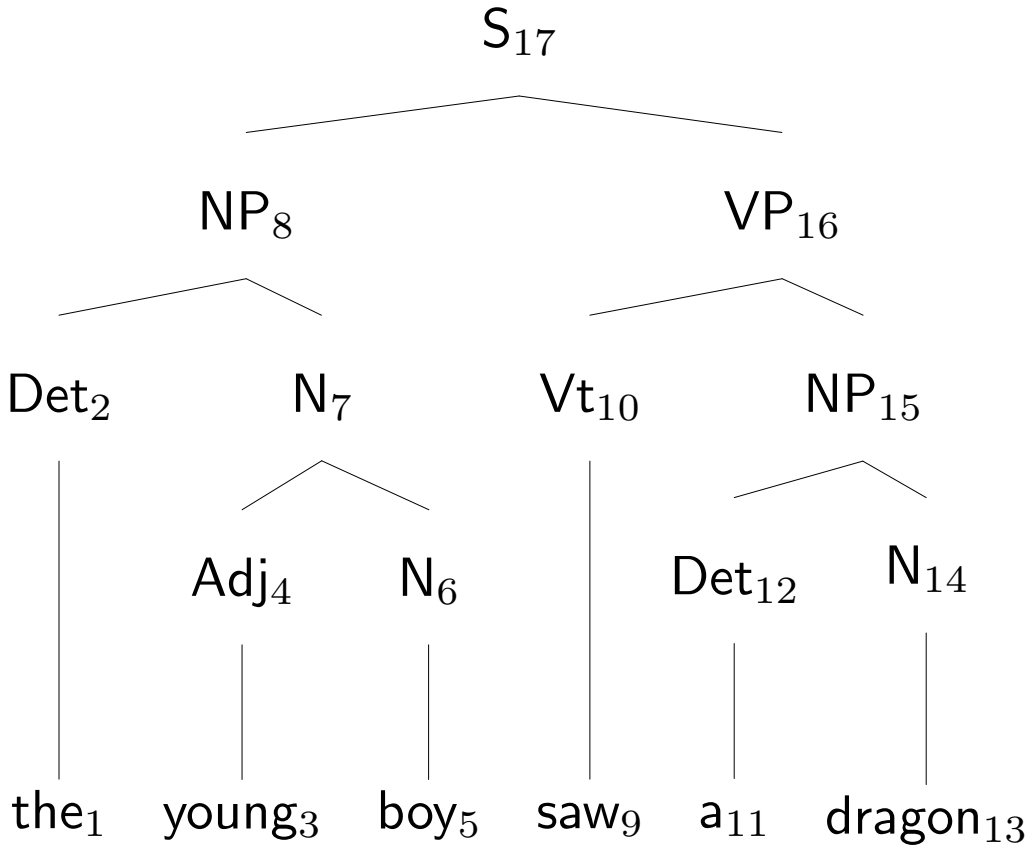
```
| ?- sr_parse([the,young,boy,saw,the,dragon]).
START: <[],[the,young,boy,saw,the,dragon]>
     Reduce []? no
     Shift "the"
<[the],[young,boy,saw,the,dragon]>
     Reduce [the] => det
<[det],[young,boy,saw,the,dragon]>
     Reduce [det]? no
     Reduce []? no
     Shift "young"
<[det,young],[boy,saw,the,dragon]>
     Reduce [det,young]? no
     Reduce [young] => adj
```

```
<[det,adj],[boy,saw,the,dragon]>
     Reduce [det,adj]? no
     Reduce [adj]? no
     Reduce []? no
     Shift "boy"
<[det,adj,boy],[saw,the,dragon]>
     Reduce [det,adj,boy]? no
     Reduce [adj,boy]? no
     Reduce [boy] => n
<[det,adj,n],[saw,the,dragon]>
     Reduce [det,adj,n]? no
     Reduce [adj,n] => n
<[det,n],[saw,the,dragon]>
     Reduce [det,n] => np
<[np],[saw,the,dragon]>
     Reduce [np]? no
     Reduce []? no
     Shift "saw"
```

```
<[np,saw],[the,dragon]>
     Reduce [np,saw]? no
     Reduce [saw] => vt
<[np,vt],[the,dragon]>
     Reduce [np,vt]? no
     Reduce [vt]? no
     Reduce []? no
     Shift "the"
<[np,vt,the],[dragon]>
     Reduce [np,vt,the]? no
     Reduce [vt,the]? no
     Reduce [the] => det
<[np,vt,det],[dragon]>
     Reduce [np,vt,det]? no
     Reduce [vt,det]? no
     Reduce [det]? no
     Reduce []? no
     Shift "dragon"
```

```
<[np,vt,det,dragon],[]>
     Reduce [np,vt,det,dragon]? no
     Reduce [vt,det,dragon]? no
     Reduce [det,dragon]? no
     Reduce [dragon] => n
<[np,vt,det,n],[]>
     Reduce [np,vt,det,n]? no
     Reduce [vt,det,n]? no
     Reduce [det,n] => np
<[np,vt,np],[]>
     Reduce [np,vt,np]? no
     Reduce [vt,np] => vp
<[np,vp],[]>
     Reduce [np,vp] => s
<[s],[]>
SUCCESS!
```

Bottom-up, left-right, depth-first tree traversal



S → NP VP
VP → Vt NP
NP → Det N
N → Adj N

Vt → saw
Det → the
Det → a
N → dragon
N → boy
Adj → young

A shift-reduce parser for grammars in CNF using difference lists to encode the string

(parser/simple/cnf_sr_diff_list.pl)

```
:- op(1100,xfx,'--->').

recognise(String) :- recognise([],String,[]) % Start

recognise([s],[],[]).              % Success

recognise([Y,X|Rest],S0,S) :-      % Reduce
    (LHS ---> [X,Y]),
    recognise([LHS|Rest],S0,S).

recognise(Stack,[Word|S0],S) :- % Shift
    Cat ---> [Word],
    recognise([Cat|Stack],S0,S).
```

A shift-reduce parser for grammars in CNF using DCG notation to encode the string

(parser/simple/cnf_sr_dcg.pl)

```
:- op(1100,xfx,'--->').

recognise(String) :- recognise([],String,[]) % Start

recognise([s],[],[]).                % Success

recognise([Y,X|Rest]) -->            % Reduce
    {LHS ---> [X,Y]},
    recognise([LHS|Rest]).

recognise(Stack) -->                 % Shift
    [Word],
    {Cat ---> [Word]},
    recognise([Cat|Stack]).
```

A trace (parser/simple/grammar.pl, parser/simple/cnf_sr_trace.pl)

```
| ?- recognise([the,young,boy,saw,the,dragon]).
START: <[],[the,young,boy,saw,the,dragon]-[]>
     Shift "the" as "det"
<[det],[young,boy,saw,the,dragon]-[]>
     Shift "young" as "adj"
<[adj,det],[boy,saw,the,dragon]-[]>
     Reduce [det,adj]? no
     Shift "boy" as "n"
<[n,adj,det],[saw,the,dragon]-[]>
     Reduce [adj,n] => n
<[n,det],[saw,the,dragon]-[]>
     Reduce [det,n] => np
<[np],[saw,the,dragon]-[]>
     Shift "saw" as "vt"
```

```
<[vt,np],[the,dragon]-[]>
     Reduce [np,vt]? no
     Shift "the" as "det"
<[det,vt,np],[dragon]-[]>
     Reduce [vt,det]? no
     Shift "dragon" as "n"
<[n,det,vt,np],[]-[]>
     Reduce [det,n] => np
<[np,vt,np],[]-[]>
     Reduce [vt,np] => vp
<[vp,np],[]-[]>
     Reduce [np,vp] => s
<[s],[]-[]>
SUCCESS!
```

Towards more efficient parsers

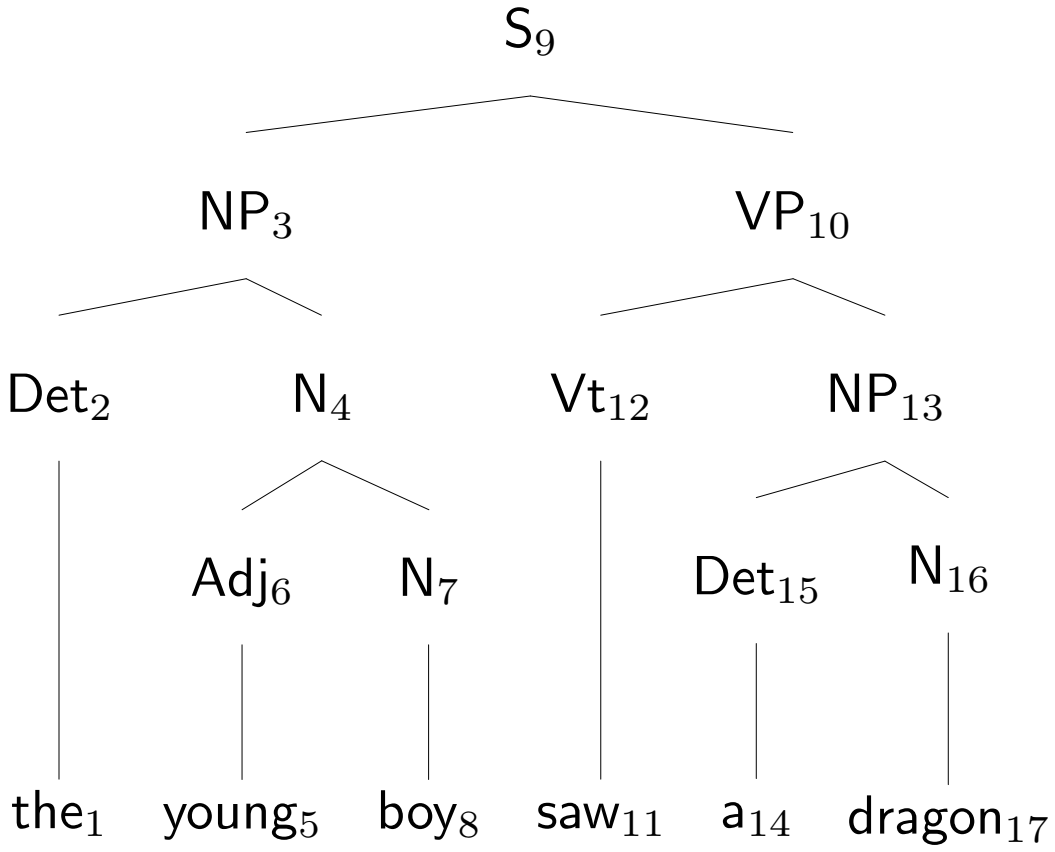
- Combining bottom-up parsing with top-down prediction
  - From shift-reduce to left-corner parsing
  - Adding more top-down filtering: link tables
- Memoization of partial results
  - well-formed substring tables
  - active charts

From shift-reduce to left-corner parsing

- Shift-reduce parsing is not goal directed at all:
  - Reduction of every possible substring,
  - obtaining every possible analysis for it.

- Idea to revise shift-reduce strategy:
  - Take a particular element x (here: the leftmost).
  - x triggers those rules it can occur in, to make predictions about the material occurring around x.

Left-corner, left-right, depth-first tree traversal



S → NP VP
VP → Vt NP
NP → Det N
N → Adj N

Vt → saw
Det → the
Det → a
N → dragon
N → boy
Adj → young

In the figure above, we numbered the mother in the tree at the time the rule is looked up of which it is the left-hand side category. Alternatively, one could number the mother only at the time when the parser tries to prove it's the left corner of something.

A left-corner parser for grammars in CNF using ordinary strings (parser/simple/cnf_lc.pl)

```
:- op(1100,xfx,'--->').

recognise(Phrase, [Word|Rest]) :-
        (Cat ---> [Word]),
        lc(Cat, Phrase, Rest).

lc(Phrase, Phrase, _).

lc(SubPhrase, SuperPhrase, String) :-
        (Phrase ---> [SubPhrase,Right]),
        append(SubString,Rest,String),
        recognise(Right, SubString),
        lc(Phrase, SuperPharse, Rest).
```

A left-corner parser for grammars in CNF using difference lists to encode the string

(parser/simple/cnf_lc_diff_list.pl)

```
:- op(1100,xfx,'--->').

recognise(Phrase, [Word|S0], S) :-
        (Cat ---> [Word]),
        lc(Cat, Phrase, S0, S).

lc(Phrase,Phrase, S, S).

lc(SubPhrase, SuperPhrase, S0, S) :-
        (Phrase ---> [SubPhrase,Right]),
        recognise(Right, S0, S1),
        lc(Phrase, SuperPharse, S1, S).
```

A left-corner parser for grammars in CNF using DCG notation to encode the string

(parser/simple/cnf_lc_dcg.pl)

```
:- op(1100,xfx,'--->').

% ?- recognise(s,<list(word)>,[]).

recognise(Phrase) --> [Word],
                      {Cat ---> [Word]},
                      lc(Cat,Phrase).

lc(Phrase,Phrase) --> [].

lc(SubPhrase,SuperPhrase) -->
    {Phrase ---> [SubPhrase,Right]},
    recognise(Right),
    lc(Phrase,SuperPhrase).
```

Problems of basic left-corner approach

- There can be a choice involved in picking a rule which
  - projects a particular word
  - projects a particular phrase

- How do we make sure we only pick a category which is on our path up to the goal?
  - Define a **link table** encoding the transitive closure of the left-corner relation. This is always a finite table!
  - Use it as an **oracle** guiding us to pick a reasonable candidate.

Example for a link table

For a grammar with the following non-terminal rules

```
:- op(1100,xfx,'--->').

  s  ---> [np, vp].          vp ---> [v, np].
  np ---> [det, n].          n  ---> [n, pp].
  pp ---> [p, np].
```

one can define or automatically deduce the link table

```
  link(s,s).      link(np,np).    link(pp,pp).
  link(det,det).  link(n,n).      link(p,p).
  link(np,s).     link(det,np).   link(p,pp).     link(v,vp).
  link(det,s).
```

Using a link table in a left-corner parser

```
:- op(1100,xfx,'--->').

recognise(Phrase) --> [Word],
                      {Cat ---> [Word]},
                      {link(Cat,Phrase)},
                      lc(Cat,Phrase).

lc(Phrase,Phrase) --> [].

lc(SubPhrase,SuperPhrase) -->
    {Phrase ---> [SubPhrase,Right]},
    {link(Phrase,SuperPhrase)},
    recognise(Right),
    lc(Phrase,SuperPhrase).
```

Observation: Inefficiency of backtracking

Two example sentences:

(1) He [gave [the young cat] [to Bill]].
(2) He [gave [the young cat] [some milk]].

The corresponding grammar rules:

```
vp ---> [v_ditrans, np, pp_to].
vp ---> [v_ditrans, np, np].
```

Solution: Memoization

- Store intermediate results:
  a) completely analyzed constituents:
     well-formed substring table or **(passive) chart**
  b) partial and complete analyses:
     (active) chart

- All intermediate results need to be stored for completeness.

- All possible solutions are explored in parallel.

CYK Parser

- Developed independently by Cocke, Younger, and Kasami
- Grammar has to be in Chomsky Normal Form (CNF), only
  - RHS with a single terminal: $A \rightarrow a$
  - RHS with two non-terminals: $A \rightarrow BC$
- Sentence representation showing position and word indices:

$$\cdot_0 \; \mathsf{w}_1 \; \cdot_1 \; \mathsf{w}_2 \; \cdot_2 \; \mathsf{w}_3 \; \cdot_3 \; \mathsf{w}_4 \; \cdot_4 \; \mathsf{w}_5 \; \cdot_5 \; \mathsf{w}_6 \; \cdot_6$$

  For example:

$$\cdot_0 \text{ the } \cdot_1 \text{ young } \cdot_2 \text{ boy } \cdot_3 \text{ saw } \cdot_4 \text{ the } \cdot_5 \text{ dragon } \cdot_6$$

The passive chart

- The well-formed substring table, henceforth (passive) chart, for a string of length n is an $n \times n$ matrix.
- An entry in a field (i, j) of the chart encodes the set of categories which spans the string from position i to j.
- More formally: chart(i,j) $= \{A \mid A \stackrel{*}{\Rightarrow} w_{i+1} \ldots w_j\}$

Coverage represented in the chart

An input sentence with 6 words:

$$\cdot_0 \; w_1 \; \cdot_1 \; w_2 \; \cdot_2 \; w_3 \; \cdot_3 \; w_4 \; \cdot_4 \; w_5 \; \cdot_5 \; w_6 \; \cdot_6$$

Coverage represented in the chart:

FROM: \ TO:	1	2	3	4	5	6
0	0–1	0–2	0–3	0–4	0–5	0–6
1		1–2	1–3	1–4	1–5	1–6
2			2–3	2–4	2–5	2–6
3				3–4	3–5	3–6
4					4–5	4–6
5						5–6

Example for coverage represented in chart

Example sentence:

$$\cdot_0 \text{ the } \cdot_1 \text{ young } \cdot_2 \text{ boy } \cdot_3 \text{ saw } \cdot_4 \text{ the } \cdot_5 \text{ dragon } \cdot_6$$

Coverage represented in chart:

	1	2	3	4	5	6
0	the	the young	the young boy	the young boy saw	the young boy saw the	the young boy saw the dragon
1		young	young boy	young boy saw	young boy saw the	young boy saw the dragon
2			boy	boy saw	boy saw the	boy saw the dragon
3				saw	saw the	saw the dragon
4					the	the dragon
5						dragon

An example for a filled-in chart

Input sentence:

$\cdot_0$ the $\cdot_1$ young $\cdot_2$ boy $\cdot_3$ saw $\cdot_4$ the $\cdot_5$ dragon $\cdot_6$

Chart:

	1	2	3	4	5	6
0	{Det}	{}	{NP}	{}	{}	{S}
1		{Adj}	{N}	{}	{}	{}
2			{N}	{}	{}	{}
3				{V}	{}	{VP}
4					{Det}	{NP}
5						{N}

Grammar:

S → NP VP
VP → Vt NP
NP → Det N
N → Adj N
Vt → saw
Det → the
Det → a
N → dragon
N → boy
Adj → young

Filling in the chart left-to-right, depth-first

	1	2	3	4	5	6
0	**1!**	**3**	**6**	**10**	**15**	**21**
1		**2!**	**5**	**9**	**14**	**20**
2			**4!**	**8**	**13**	**19**
3				**7!**	**12**	**18**
4					**11!**	**17**
5						**16!**

for $j := 1$ to 6
    lexical-chart-fill$(j-1, j)$
    for $i := j-2$ down to 0
        syntactic-chart-fill(i, j)

lexical-chart-fill(j-1,j)

- Idea: Lexical lookup. Fill the field $(j-1, j)$ in the chart with the preterminal category dominating word j.

- Realized as:

$$chart(j-1, j) := \{\mathsf{X} \mid \mathsf{X} \rightarrow \mathsf{word}_j \in \mathsf{P}\}$$

syntactic-chart-fill(i,j)

- Idea: Perform all reduction step using syntactic rules such that the reduced symbol covers the string from i to j.

- Realized as:

$$chart(i,j) = \left\{ A \;\middle|\; \begin{array}{l} A \rightarrow BC \in P, \\ i < k < j, \\ B \in chart(i,k), \\ C \in chart(k,j) \end{array} \right\}$$

Explicit version of syntactic-chart-fill(i,j)

- Needed: version making explicit enumerations of
  - every possible value of k and
  - every context free rule

- Code:
  $chart(i, j) := \{\}$.
  for $k := i + 1$ to $j - 1$ do
      for every $A \rightarrow BC \in P$ do
          if $B \in chart(i, k)$ and $C \in chart(k, j)$ then
              $chart(i, j) := chart(i, j) \cup \{\mathsf{A}\}$.

The complete CYK algorithm

for $j := 1$ to n do
    $chart(j-1, j) := \{\mathsf{X} \mid \mathsf{X} \rightarrow \mathsf{word}_j \in \mathsf{P}\}$
    for $i := j - 2$ down to 0 do
        $chart(i, j) := \{\}$
        for $k := i + 1$ to $j - 1$ do
            for every $A \rightarrow BC \in P$ do
                if $B \in chart(i, k)$ and $C \in chart(k, j)$ then
                    $chart(i, j) := chart(i, j) \cup \{\mathsf{A}\}$
if $\mathsf{S} \in chart(0, n)$ then accept else reject

The CYK algorithm in PROLOG (parser/cky/cky.pl)

```
% Data structures: chart(From,To,Category)
:- dynamic chart/3.

% Operator for grammar rules
:- op(1100,xfx,'--->').

% recognize(+WordList,?Startsymbol)
% top-level predicate for CYK recognizer

recognize(S,Cat) :-
   retractall(chart(_,_,_)),
   length(S,N),
   fill(0,N,S),
   chart(0,N,Cat).
```

```
% fill(+Current minus one,+Last,+WordList)
% Main j-loop from 1 to number of words in string.

fill(N,N,[]).
fill(JminOne,N,[W|Ws]) :-
   J is JminOne + 1,
   lexical_chart_fill(J,JminOne,W),
   %
   I is J - 2,
   syntactic_chart_fill(I,J),
   %
   fill(J,N,Ws).
```

```
% lexical_chart_fill(+J,+JminOne,+Word)
% fill main diagonal with preterminal categories

lexical_chart_fill(J,JminOne,W) :-
   findall_unique(X,(X ---> [W]),Cats),
   add_all_to_chart(JminOne,J,Cats).

% syntactic_chart_fill(+I,+J)
% i-loop from J-2 down to 0

syntactic_chart_fill(-1,_) :- !.
syntactic_chart_fill(I,J) :-
   K is I+1,
   build_phrases_from_to(I,K,J),
   IminOne is I-1,
   syntactic_chart_fill(IminOne,J).
```

```
% build_phrases_from_to(+From,+Current,+To)

build_phrases_from_to(_,J,J) :- !.
build_phrases_from_to(I,K,J) :-
   findall_unique(A,(chart(I,K,B),
                     chart(K,J,C),
                     (A ---> [B,C])),
                  List),
   add_all_to_chart(I,J,List),
   KplusOne is K+1,
   build_phrases_from_to(I,KplusOne,J).
```

```
% add_one_to_chart(+FromIndex,+ToIndex,+Contents)
% a) only add if it does not yet exist:
add_one_to_chart(From,To,Cat) :- chart(From,To,Cat), !.

% b) add a chart entry
add_one_to_chart(From,To,Cat) :-
   assertz(chart(From,To,Cat)).

add_all_to_chart(_,_,[]).
add_all_to_chart(From,To,[Cat|Cats]) :-
   add_one_to_chart(From,To,Cat),
   add_all_to_chart(From,To,Cats).
```

```
% findall_unique(+Var,+CallWithVar,-ResultList)
% Obtain the list of all call results without duplicates.
% (uses builtin predicates findall/3 and sort/3)

findall_unique(Var,Goal,UniqueResults) :-
        findall(Var,Goal,Results),
        sort(Results,UniqueResults).
```