Einführung in die Computerlinguistik

Parsing

Laura Kallmeyer

Heinrich-Heine-Universität Düsseldorf

Summer 2018

Introduction (1)

A parser is a device that accepts a word w and a grammar G as input and that

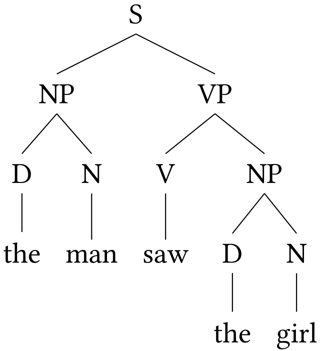
1. decides whether w is in the language generated by the grammar and
2. if so, it provides a syntactic analysis for w or, if w is ambiguous, a set of analyses, oftentimes represented in a compact way as a derivation forest.

A device that does only the first part of the task is called a recognizer.

Introduction (2)

Example for parsing:

Input: “the man saw the girl”.
Output:



Input: “the man saw saw the girl”. Output: no.

Top-Down Parsing (1)

CFG parser that is

- a top-down parser: we start with S and subsequently replace lefthand sides of productions with righthand sides.
- a directional parser: the expanding of non-terminals (with appropriate righthand sides) is ordered; we start with the leftmost non-terminal and go through the righthand sides of productions from left to right.
  In particular: we determine the start position of the span of the ith symbol in a rhs only after having processed the $i-1$ preceding symbols.
- a LL-parser: we process the input from left to right while constructing a leftmost derivation.

First proposed by Sheila Greibach (for CFGs in GNF).

Top-Down Parsing (2)

Assume a CFG without left recursion $A \stackrel{+}{\Rightarrow} A\alpha$ for $\alpha \in (N \cup T)^*$.

The parser goes through different pairs of remaining input and sentential form (a stack), starting with w and the start symbol S.
In each step, we

- either scan the next input symbol, provided it corresponds to the top of the sentential form
- or we non-deterministically predict a production that expands the top of the sentential form, provided this is a non-terminal. In this case we replace it with the rhs of a production.

Success, if we end with empty remaining input and empty sentential form.

Top-Down Parsing (3)

Top-down parsing

$S \to aSb \mid c$, input $aacbb$.

1.	$aacbb$	S	initial
2.	$aacbb$	aSb	predict from 1.
4.	$aacbb$	c	predict from 1.
5.	$acbb$	Sb	scan from 2.
6.	$acbb$	$aSbb$	predict from 5.
7.	$acbb$	cb	predict from 5.
8.	cbb	Sbb	scan from 6.
9.	cbb	$aSbbb$	predict from 8.
10.	cbb	cbb	predict from 8.
11.	bb	bb	scan from 10.
12.	b	b	scan from 11.
13.	ε	ε	scan from 12.

Top-Down Parsing (4)

Function `top-down` with arguments

- w: remaining input;
- α: remaining sentential form (a stack).

$\texttt{top-down}(w, \alpha)$ iff $\alpha \stackrel{*}{\Rightarrow} w$ (for $\alpha \in (N \cup T)^*$, $w \in T^*$)

Initial call:
$\texttt{top-down}(w, S)$

Top-Down Parsing (5)

```
function top-down(w,α):
   out = false;
   if w = α = ϵ, then out = true;
   else if w = aw′ and α = aα′,
      then out = top-down(w′,α′)            scan
   else if α = Xα′ with X ∈ N,
      then for all X → X₁...Xₖ:
         if top-down(w, X₁...Xₖα′)      predict
            then out = true;
   return out
```

Top-Down Parsing (6)

How to turn the recognizer into a parser:

Add an analysis stack to the parser that allows you to construct the parse tree.

Assume that for each $A \in N$, the righthand sides of A-productions are numbered (have indices).

Whenever

- a production is applied (prediction step), the lefthand side is pushed on the analysis stack together with the index of the righthand side;
- a terminal a is scanned, a is pushed on the analysis stack. (This is needed for backtracking in a depth-first strategy.)

Top-Down Parsing (7)

```
function top-down(w,α,Γ):
   out = false;
   if w = α = ϵ,
      then output Γ; out = true;
   else if w = aw′ and α = aα′,
      then out = top-down(w′,α′,aΓ)
   else if α = Xα′ with X ∈ N,
      then for all X → γ with rhs-index i:
         if top-down(w, γα′,⟨X,i⟩Γ)
            then out = true;
   return out
```

Shift-Reduce Parsing (1)

CFG parser that is

- a bottom-up parser: we start with the terminals and subsequently replace righthand sides of productions with lefthand sides.
- a directional parser: the replacing of righthand sides with lefthand sides is ordered corresponding to a rightmost derivation.
- a LR-parser: we process the input from left to right while constructing a rightmost derivation.
- a Shift-reduce-parser: the two operations of the parser are shift and reduce.

Shift-Reduce Parsing (2)

The parser consists of

- a stack (initially empty) $\Gamma \in (N \cup T)^*$
- the remaining input (initially w).

Idea:

- w is shifted on the stack while, whenever the top of the stack is the rhs of a production in reverse order, this is replaced with the lhs.
- Success if $\Gamma = S$ and remaining input ϵ.

Shift-Reduce Parsing (3)

For convenience we write the stack with its top on the right.

Shift-reduce parsing

$S \to ABC,\ A \to a \mid Aa,\ B \to b \mid Bb,\ C \to c$
$w = aabbbc.$

	$aabbbc$	
a	$abbbc$	shift
A	$abbbc$	reduce, $A \to a$
Aa	$bbbc$	shift
A	$bbbc$	reduce, $A \to Aa$
Ab	bbc	shift
AB	bbc	reduce, $B \to b$
ABb	bc	shift

Shift-Reduce Parsing (4)

Example continued

ABb	bc	
AB	bc	reduce, $B \rightarrow Bb$
ABb	c	shift
AB	c	reduce, $B \rightarrow Bb$
ABc		shift
ABC		reduce, $C \rightarrow c$
S		reduce, $S \rightarrow ABC$

If we apply the productions in reverse order we obtain a rightmost derivation:
$S \Rightarrow ABC \Rightarrow ABc \Rightarrow ABbc \Rightarrow ABbbc \Rightarrow Abbbc \Rightarrow Aabbbc \Rightarrow aabbbc$

Shift-Reduce Parsing (5)

Assume a grammar without ϵ-productions and without loops.

```
function bottom-up(w,Γ):
    if w = ϵ and Γ = S then true
    else reduce(w,Γ) or shift(w,Γ)
```

```
function shift(w,Γ):
    out = false
    if w = aw′ and a ∈ T
        then out = bottom-up(w′,Γa)
    return out
```

Shift-Reduce Parsing (6)

```
function reduce(w,Γ):
    out = false;
    for every A → α ∈ P:
        if Γ = Γ′α and bottom-up(w,Γ′A)
        then out = true;
    return out
```

Initial call: `bottom-up(w, ϵ)`

CYK

The CYK parser is

- a bottom-up parser: we start with the terminals in the input string and subsequently compute recognized parse trees by going from already recognized rhs of productions to the non-terminal on the lefthand side.
- a non-directional parser: the order of the completing of sub-trees is not necessarily from left to right.
- a chart parser: we store every intermediate result in a chart and can reuse it in different contexts. This avoids computing the same subtree several times. Particularly useful for ambiguous grammars such as natural language grammars.

Independently proposed by Cocke, Kasami and Younger in the 60s.

CYK recognizer for CNF (1)

A CFG is in Chomsky Normal Form iff all productions are either of the form $A \rightarrow a$ or $A \rightarrow B\ C$.

If the grammar has this form,

- we need to check only for two categories B, C, in order to construct an A with $A \rightarrow B\ C$.
- we can be sure that the spans always become longer when applying productions $A \rightarrow B\ C$. I.e., if l_1 and l_2 are the lengthes of B and C, then the length of the resulting A is $l_1 + l_2 > max(l_1, l_2)$.

Every CFG can be transformed into an equivalent CFG in CNF.

CYK recognizer for CNF (2)

The chart C is an $n \times n$-array. The first index is the index of the first terminal in the span and the second gives the length of a span.

$A \in C_{i,l}$ indicates that we have found an A with a span starting at index i and having length l.

Algorithm:

```
C_{i,1} = {A | A → w_i ∈ P}                                   scan
for all l ∈ [1..n]:
    for all i ∈ [1..n]:
        for every A → B C:
            if there is a l_1 ∈ [1..l − 1] such that
                B ∈ C_{i,l_1} and C ∈ C_{i+l_1,l−l_1},
            then C_{i,l} = C_{i,l} ∪ {A}                     complete
```

CYK recognizer for CNF (3)

CYK recognition for CNF grammars

$S \to C_aC_b \mid C_aS_B, S_B \to SC_b, C_a \to a, C_b \to b.$

(From $S \to aSb \mid ab$ with transformation into CNF.)

$w = aaabbb.$

l							
6	S						
5		S_B					
4		S					
3			S_B				
2			S				
1	C_a	C_a	C_a	C_b	C_b	C_b	
	1 a	2 a	3 a	4 b	5 b	6 b	i

CYK parsing (1)

We know that for every CFG G with $\epsilon \notin L(G)$ we can

- eliminate ϵ-productions,
- eliminate unary productions,
- eliminate useless symbols,
- transform into CNF,

and the resulting CFG G' is such that $L(G) = L(G')$.
Therefore, for every CFG, we can use the CNF recognizer after transformation.

How can we obtain a parser?

CYK parsing (2)

We need to do two things:

- turn the CNF recognizer into a parser, and
- if the original grammar was not in CNF, retrieve the original syntax from the CNF syntax.

CYK parsing (3)

To turn the CNF recognizer into a parser, we record not only non-terminal categories but whole productions with the positions and lenghts of the rhs symbols in the chart (i.e., with backpointers):

$C_{i,1} := \{A \to w_i \mid A \to w_i \in P\}$
for all $l \in [1..n]$:
  for all $i \in [1..n]$:
    for every $A \to B\ C$:
      if there is a $l_1 \in [1..l-1]$ such that
        $B \in C_{i,l_1}$ and $C \in C_{i+l_1,l-l_1}$,
      then $C_{i,l} := C_{i,l} \cup \{A \to [B, i, l_1][C, i+l_1, l-l_1]\}$

We can then obtain a parse tree by traversing the productions from left to right, starting with every S-production in $C_{1,n}$.

CYK parsing (4)

Example

$S \to C_a C_b \mid C_a S_B$, $S_B \to S C_b$, $C_a \to a$, $C_b \to b$, $w = aaabbb$. (We write $A_{i,l}$ for $[A, i, l]$.)

$S \to C_{a1,1} S_{B2,5}$					
	$S_B \to S_{2,4} C_{b6,1}$				
	$S \to C_{a2,1} S_{B3,3}$				
		$S_B \to S_{3,2} C_{b5,1}$			
		$S \to C_{a3,1} C_{b4,1}$			
$C_a \to a$	$C_a \to a$	$C_a \to a$	$C_b \to b$	$C_b \to b$	$C_b \to b$

CYK parsing (5)

From the CNF parse tree to the original parse tree:
First, we undo the CNF transformation:

- replace every $C_a \rightarrow a$ in the chart with a and replace every occurrence of C_a in a production with a.
- For all $l, i \in [1..n]$: If $A \rightarrow \alpha D_{i_D,l_D} \in C_{i,l}$ such that D is a new symbol introduced in the CNF transformation and $D \rightarrow \beta \in C_{i_D,l_D}$, then replace $A \rightarrow \alpha D_{i_D,l_D}$ with $A \rightarrow \alpha\beta$ in $C_{i,l}$.
- Finally remove all $D \rightarrow \gamma$ with D being a new symbol introduced in the CNF transformation from the chart.

CYK parsing (6)

Example

$S \to C_aC_b \mid C_aS_B$, $S_B \to SC_b$, $C_a \to a$, $C_b \to b$, $w = aaabbb$. New symbols: C_a, C_b, S_B. Elimination of C_a, C_b:

6	$S \to aS_{B2,5}$					
5		$S_B \to S_{2,4}b$				
4		$S \to aS_{B3,3}$				
3			$S_B \to S_{3,2}b$			
2			$S \to ab$			
1	a	a	a	b	b	b
	1	2	3	4	5	6

CYK parsing (7)

Example

$S \rightarrow C_a C_b \mid C_a S_B$, $S_B \rightarrow S C_b$, $C_a \rightarrow a$, $C_b \rightarrow b$, $w = aaabbb$.

Replacing of S_B in rhs:

6	$S \rightarrow aS_{2,4}b$					
5		$S_B \rightarrow S_{2,4}b$				
4		$S \rightarrow aS_{3,2}b$				
3			$S_B \rightarrow S_{3,2}b$			
2			$S \rightarrow ab$			
1	a	a	a	b	b	b
	1	2	3	4	5	6

CYK parsing (8)

Example

$S \rightarrow C_aC_b \mid C_aS_B, S_B \rightarrow SC_b, C_a \rightarrow a, C_b \rightarrow b, w = aaabbb.$

Elimination of S_B:

6	$S \rightarrow aS_{2,4}b$					
5						
4		$S \rightarrow aS_{3,2}b$				
3						
2			$S \rightarrow ab$			
1	a	a	a	b	b	b
	1	2	3	4	5	6

CYK parsing (9)

Undo the elimination of unary productions:

- For every $A \rightarrow \beta$ in $C_{i,l}$ that has been added in removing of the unary productions to replace $B \rightarrow \beta'$ (β' is β without chart indices): replace A with B in this entry in $C_{i,l}$.
- For every unary production $A \rightarrow B$ in the original grammar and for every $B \rightarrow \beta \in C_{i,l}$: add $A \rightarrow B_{i,l}$ to $C_{i,l}$. Repeat this until chart does not change any more.

CYK parsing (10)

Undo the elimination of ϵ-productions:

- Add a row with $l = 0$ and a column with $i = n + 1$ to the chart.
- Fill row 0 as in the general case using the original CFG grammar (tabulating productions).
- For every $A \rightarrow \beta$ in $C_{i,l}$ that has been added in removing the ϵ-productions: add the deleted nonterminals to β with the position of the preceding non-terminal as starting position (or i if it is the first in the rhs) and with length 0.