Top Down Parsing

Johannes Dellert   Aleksandar Dimitrov

Seminar für Sprachwissenschaft, Universität Tübingen

December 2006

Outline

Top-Down Parsing

- CYK and Unger parser are non-directional methods

Top-Down Parsing

- CYK and Unger parser are non-directional methods
- They need the whole input sentence before beginning to parse

Top-Down Parsing

- CYK and Unger parser are non-directional methods
- They need the whole input sentence before beginning to parse
- Today we introduce a directional top-down parsing method

Top-Down Parsing

- CYK and Unger parser are non-directional methods
- They need the whole input sentence before beginning to parse
- Today we introduce a directional top-down parsing method
- This is what the term 'Top-down Parsing' usually refers to

General approach

- Rederive the word starting at the input symbol

General approach

- Rederive the word starting at the input symbol
- Build the tree from the top

General approach

- Rederive the word starting at the input symbol
- Build the tree from the top
- Collect 'ideas' on how the tree might be continued

General approach

- Rederive the word starting at the input symbol
- Build the tree from the top
- Collect 'ideas' on how the tree might be continued
- If the tree is 'full' and all the input is in the tree, parsing was successful

Intuitive example

- Assume the following grammar:

$S \rightarrow NP\ VP$
$NP \rightarrow D\ N$
$VP \rightarrow VT\ NP \mid VI\ PP$
$PP \rightarrow P\ NP$
$D \rightarrow \text{der} \mid \text{die}$
$N \rightarrow \text{Mond} \mid \text{Wiese}$
$VI \rightarrow \text{scheint}$
$VT \rightarrow \text{bescheint}$
$P \rightarrow \text{auf}$

Intuitive example

- Now let us parse the sentence 'der Mond scheint auf die Wiese'
- First 'tree idea':

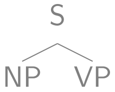
Intuitive example

- Now let us parse the sentence 'der Mond scheint auf die Wiese'
- First 'tree idea':

S

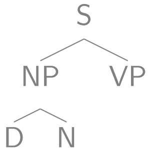
Intuitive example

- 'Tree idea' is expanded via leftmost derivations:



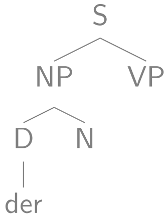
Intuitive example

- 'Tree idea' is expanded via leftmost derivations:



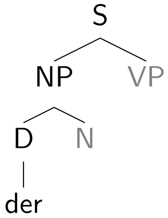
Intuitive example

- 'Tree idea' is expanded via leftmost derivations:



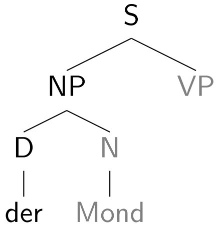
Intuitive example

- Tree begins to match input and is expanded:



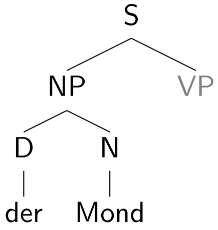
Intuitive example

- Tree begins to match input and is expanded:



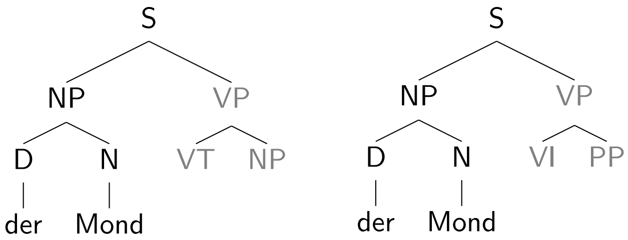
Intuitive example

- Tree begins to match input and is expanded:



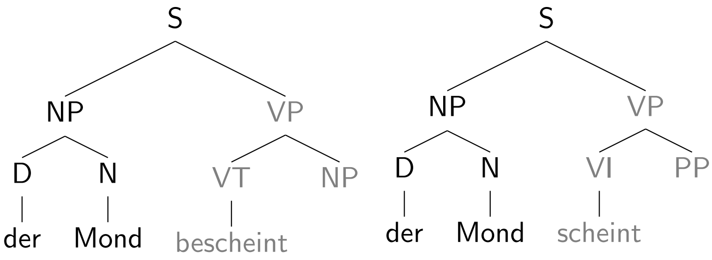
Intuitive example

- Nondeterminism: Two different possible trees.



Intuitive example

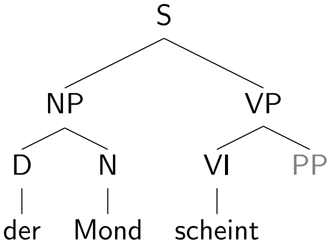
- Nondeterminism: Expanding both trees.



Johannes Dellert, Aleksandar Dimitrov

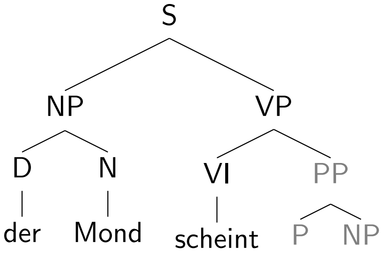
Intuitive example

- No scan possible for first tree; remaining tree gets expanded



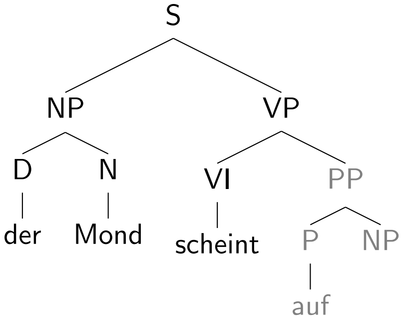
Intuitive example

- Expanding the predicted tree



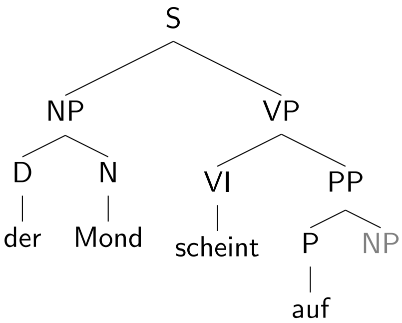
Intuitive example

- Expanding the predicted tree



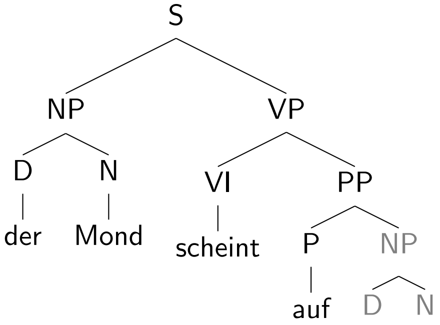
Intuitive example

- Expanding the predicted tree



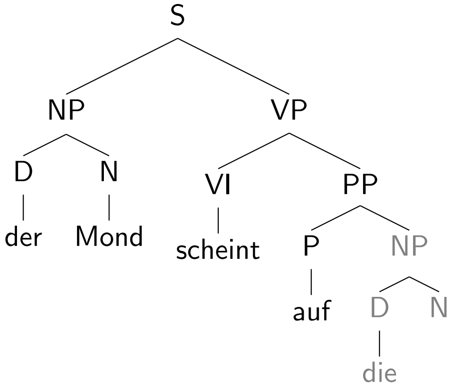
Intuitive example

- Expanding the predicted tree



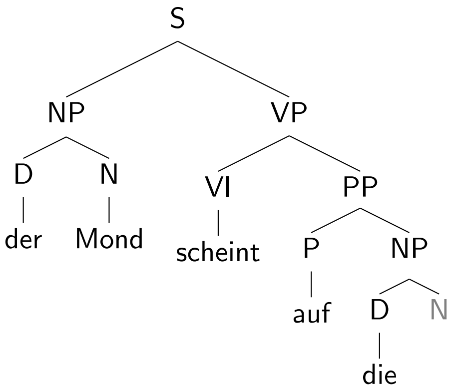
Intuitive example

- Expanding the predicted tree



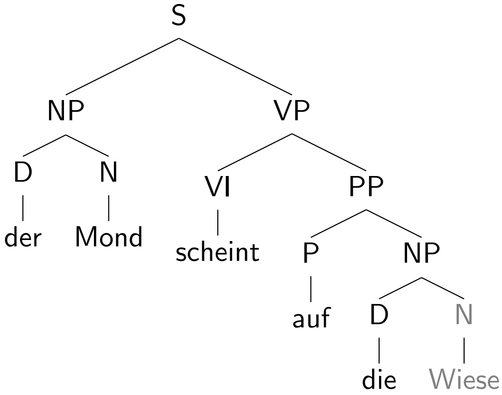
Intuitive example

- Expanding the predicted tree



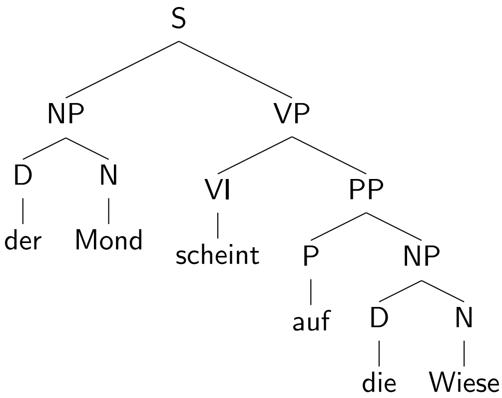
Intuitive example

- Expanding the predicted tree



Intuitive example

- Tree completed



General features of the top down method

- The parser makes predictions about the input.

General features of the top down method

- The parser makes predictions about the input.
- The left-most prediction is usually processed first.

General features of the top down method

- The parser makes predictions about the input.
- The left-most prediction is usually processed first.
- Terminals in the prediction are matched against the input.

General features of the top down method

- The parser makes predictions about the input.
- The left-most prediction is usually processed first.
- Terminals in the prediction are matched against the input.
- Non-Terminals are replaced by one of the right hand sides.

Operations

- For Bottom-Up-Parsing, we got to know SHIFT and REDUCE

Operations

- For Bottom-Up-Parsing, we got to know SHIFT and REDUCE
- The corresponding operations for Top-Down-Parsing are called PREDICT and SCAN

Operations

- For Bottom-Up-Parsing, we got to know SHIFT and REDUCE
- The corresponding operations for Top-Down-Parsing are called PREDICT and SCAN
- PREDICT replaces a non-terminal in the sentential form with the right hand side of a corresponding rule:

Operations

- For Bottom-Up-Parsing, we got to know SHIFT and REDUCE
- The corresponding operations for Top-Down-Parsing are called PREDICT and SCAN
- PREDICT replaces a non-terminal in the sentential form with the right hand side of a corresponding rule:
- e.g. der Mond VP $\rightarrow$ der Mond V PP for a rule VP $\rightarrow$ V PP

Operations

- For Bottom-Up-Parsing, we got to know SHIFT and REDUCE
- The corresponding operations for Top-Down-Parsing are called PREDICT and SCAN
- PREDICT replaces a non-terminal in the sentential form with the right hand side of a corresponding rule:
- e.g. der Mond VP → der Mond V PP for a rule VP → V PP
- SCAN matches a terminal in the sentential form with a symbol on the input string

Operations

- For Bottom-Up-Parsing, we got to know SHIFT and REDUCE
- The corresponding operations for Top-Down-Parsing are called PREDICT and SCAN
- PREDICT replaces a non-terminal in the sentential form with the right hand side of a corresponding rule:
- e.g. der Mond VP $\rightarrow$ der Mond V PP for a rule VP $\rightarrow$ V PP
- SCAN matches a terminal in the sentential form with a symbol on the input string
- e.g. der N VP $\rightarrow$ N VP, 'der' matched in the input string

Parsing Schema - Basics

- Parsing Schemata are a formal way of describing parsing methods

Parsing Schema - Basics

- Parsing Schemata are a formal way of describing parsing methods
- they are independent of the actual implementation

Parsing Schema - Basics

- Parsing Schemata are a formal way of describing parsing methods
- they are independent of the actual implementation
- Every recognized subtree (or tree hypothesis) is stored as an item

Parsing Schema - Basics

- Parsing Schemata are a formal way of describing parsing methods
- they are independent of the actual implementation
- Every recognized subtree (or tree hypothesis) is stored as an item
- Items look like this: $[\bullet\beta, j]$

Parsing Schema - Basics

- Parsing Schemata are a formal way of describing parsing methods
- they are independent of the actual implementation
- Every recognized subtree (or tree hypothesis) is stored as an item
- Items look like this: $[\bullet\beta, j]$
- Meaning: β parts of the tree to be 'filled', j current position in input string

Parsing Schema - Basics

- Parsing Schemata are a formal way of describing parsing methods
- they are independent of the actual implementation
- Every recognized subtree (or tree hypothesis) is stored as an item
- Items look like this: $[\bullet\beta, j]$
- Meaning: β parts of the tree to be 'filled', j current position in input string
- We start with $[\bullet S, 0]$ because the whole tree has to be built and we have not yet scanned anything from the input string

Parsing Schema - Basics

- Parsing Schemata are a formal way of describing parsing methods
- they are independent of the actual implementation
- Every recognized subtree (or tree hypothesis) is stored as an item
- Items look like this: $[\bullet\beta, j]$
- Meaning: β parts of the tree to be 'filled', j current position in input string
- We start with $[\bullet S, 0]$ because the whole tree has to be built and we have not yet scanned anything from the input string
- Our goal item will be $[\bullet, n]$ meaning that the tree is complete and the whole input of length n is scanned

Parsing Schema - Inference Rules

- How do we formalize the scanning step?

Parsing Schema - Inference Rules

- How do we formalize the scanning step?

$$\frac{[\bullet w_{j+1}\beta, j]}{[\bullet\beta, j+1]} \tag{1}$$

- How do we formalize the prediction step?

Parsing Schema - Inference Rules

- How do we formalize the scanning step?

$$\frac{[\bullet w_{j+1}\beta, j]}{[\bullet\beta, j+1]} \tag{1}$$

- How do we formalize the prediction step?

$$\frac{[\bullet B\beta, j]}{[\bullet\gamma\beta, j]} B \rightarrow \gamma \tag{2}$$

Now a complete derivation - with the Schema

1	$[\bullet S, 0]$	INITIALIZE

Now a complete derivation - with the Schema

1	$[\bullet S, 0]$	INITIALIZE
2	$[\bullet NP\ VP, 0]$	PREDICT from 1

Now a complete derivation - with the Schema

1	$[\bullet S, 0]$	INITIALIZE
2	$[\bullet NP\ VP, 0]$	PREDICT from 1
3	$[\bullet D\ N\ VP, 0]$	PREDICT from 2

Now a complete derivation - with the Schema

1	$[\bullet S, 0]$	INITIALIZE
2	$[\bullet NP\ VP, 0]$	PREDICT from 1
3	$[\bullet D\ N\ VP, 0]$	PREDICT from 2
4	$[\bullet der\ N\ VP, 0]$	PREDICT from 3

Now a complete derivation - with the Schema

1	$[\bullet S, 0]$	INITIALIZE
2	$[\bullet NP\ VP, 0]$	PREDICT from 1
3	$[\bullet D\ N\ VP, 0]$	PREDICT from 2
4	$[\bullet der\ N\ VP, 0]$	PREDICT from 3
5	$[\bullet die\ N\ VP, 0]$	PREDICT from 3

Now a complete derivation - with the Schema

1	$[\bullet S, 0]$	INITIALIZE
2	$[\bullet NP\ VP, 0]$	PREDICT from 1
3	$[\bullet D\ N\ VP, 0]$	PREDICT from 2
4	$[\bullet der\ N\ VP, 0]$	PREDICT from 3
5	$[\bullet die\ N\ VP, 0]$	PREDICT from 3
6	$[\bullet N\ VP, 1]$	SCAN from 4

Now a complete derivation - with the Schema

1	$[\bullet S, 0]$	INITIALIZE
2	$[\bullet NP\ VP, 0]$	PREDICT from 1
3	$[\bullet D\ N\ VP, 0]$	PREDICT from 2
4	$[\bullet der\ N\ VP, 0]$	PREDICT from 3
5	$[\bullet die\ N\ VP, 0]$	PREDICT from 3
6	$[\bullet N\ VP, 1]$	SCAN from 4
7	$[\bullet Mond\ VP, 1]$	PREDICT from 6

Now a complete derivation - with the Schema

1	$[\bullet S, 0]$	INITIALIZE
2	$[\bullet NP\ \ VP, 0]$	PREDICT from 1
3	$[\bullet D\ \ N\ \ VP, 0]$	PREDICT from 2
4	$[\bullet der\ \ N\ \ VP, 0]$	PREDICT from 3
5	$[\bullet die\ \ N\ \ VP, 0]$	PREDICT from 3
6	$[\bullet N\ \ VP, 1]$	SCAN from 4
7	$[\bullet Mond\ \ VP, 1]$	PREDICT from 6
8	$[\bullet Wiese\ \ VP, 1]$	PREDICT from 6

Now a complete derivation - with the Schema

1	$[\bullet S, 0]$	INITIALIZE
2	$[\bullet NP\ VP, 0]$	PREDICT from 1
3	$[\bullet D\ N\ VP, 0]$	PREDICT from 2
4	$[\bullet der\ N\ VP, 0]$	PREDICT from 3
5	$[\bullet die\ N\ VP, 0]$	PREDICT from 3
6	$[\bullet N\ VP, 1]$	SCAN from 4
7	$[\bullet Mond\ VP, 1]$	PREDICT from 6
8	$[\bullet Wiese\ VP, 1]$	PREDICT from 6
9	$[\bullet VP, 2]$	SCAN from 7

Now a complete derivation - with the Schema

9 $[\bullet VP, 2]$ SCAN from 7

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ NP, 2]$	PREDICT from 9

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ NP, 2]$	PREDICT from 9
11	$[\bullet VI\ PP, 2]$	PREDICT from 9

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ \ NP, 2]$	PREDICT from 9
11	$[\bullet VI\ \ PP, 2]$	PREDICT from 9
12	$[\bullet bescheint\ \ NP, 2]$	PREDICT from 10

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ NP, 2]$	PREDICT from 9
11	$[\bullet VI\ PP, 2]$	PREDICT from 9
12	$[\bullet bescheint\ NP, 2]$	PREDICT from 10
13	$[\bullet scheint\ PP, 2]$	PREDICT from 11

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ NP, 2]$	PREDICT from 9
11	$[\bullet VI\ PP, 2]$	PREDICT from 9
12	$[\bullet bescheint\ NP, 2]$	PREDICT from 10
13	$[\bullet scheint\ PP, 2]$	PREDICT from 11
14	$[\bullet PP, 3]$	SCAN from 13

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ NP, 2]$	PREDICT from 9
11	$[\bullet VI\ PP, 2]$	PREDICT from 9
12	$[\bullet bescheint\ NP, 2]$	PREDICT from 10
13	$[\bullet scheint\ PP, 2]$	PREDICT from 11
14	$[\bullet PP, 3]$	SCAN from 13
15	$[\bullet P\ NP, 3]$	PREDICT from 14

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ NP, 2]$	PREDICT from 9
11	$[\bullet VI\ PP, 2]$	PREDICT from 9
12	$[\bullet bescheint\ NP, 2]$	PREDICT from 10
13	$[\bullet scheint\ PP, 2]$	PREDICT from 11
14	$[\bullet PP, 3]$	SCAN from 13
15	$[\bullet P\ NP, 3]$	PREDICT from 14
16	$[\bullet auf\ NP, 3]$	PREDICT from 15

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ \ NP, 2]$	PREDICT from 9
11	$[\bullet VI\ \ PP, 2]$	PREDICT from 9
12	$[\bullet bescheint\ \ NP, 2]$	PREDICT from 10
13	$[\bullet scheint\ \ PP, 2]$	PREDICT from 11
14	$[\bullet PP, 3]$	SCAN from 13
15	$[\bullet P\ \ NP, 3]$	PREDICT from 14
16	$[\bullet auf\ \ NP, 3]$	PREDICT from 15
17	$[\bullet NP, 4]$	SCAN from 16

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ NP, 2]$	PREDICT from 9
11	$[\bullet VI\ PP, 2]$	PREDICT from 9
12	$[\bullet bescheint\ NP, 2]$	PREDICT from 10
13	$[\bullet scheint\ PP, 2]$	PREDICT from 11
14	$[\bullet PP, 3]$	SCAN from 13
15	$[\bullet P\ NP, 3]$	PREDICT from 14
16	$[\bullet auf\ NP, 3]$	PREDICT from 15
17	$[\bullet NP, 4]$	SCAN from 16
18	$[\bullet D\ N, 4]$	PREDICT from 17

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ \ NP, 2]$	PREDICT from 9
11	$[\bullet VI\ \ PP, 2]$	PREDICT from 9
12	$[\bullet bescheint\ \ NP, 2]$	PREDICT from 10
13	$[\bullet scheint\ \ PP, 2]$	PREDICT from 11
14	$[\bullet PP, 3]$	SCAN from 13
15	$[\bullet P\ \ NP, 3]$	PREDICT from 14
16	$[\bullet auf\ \ NP, 3]$	PREDICT from 15
17	$[\bullet NP, 4]$	SCAN from 16
18	$[\bullet D\ \ N, 4]$	PREDICT from 17
19	$[\bullet der\ \ N, 4]$	PREDICT from 18

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ \ NP, 2]$	PREDICT from 9
11	$[\bullet VI\ \ PP, 2]$	PREDICT from 9
12	$[\bullet bescheint\ \ NP, 2]$	PREDICT from 10
13	$[\bullet scheint\ \ PP, 2]$	PREDICT from 11
14	$[\bullet PP, 3]$	SCAN from 13
15	$[\bullet P\ \ NP, 3]$	PREDICT from 14
16	$[\bullet auf\ \ NP, 3]$	PREDICT from 15
17	$[\bullet NP, 4]$	SCAN from 16
18	$[\bullet D\ \ N, 4]$	PREDICT from 17
19	$[\bullet der\ \ N, 4]$	PREDICT from 18
20	$[\bullet die\ \ N, 4]$	PREDICT from 18

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ \ NP, 2]$	PREDICT from 9
11	$[\bullet VI\ \ PP, 2]$	PREDICT from 9
12	$[\bullet bescheint\ \ NP, 2]$	PREDICT from 10
13	$[\bullet scheint\ \ PP, 2]$	PREDICT from 11
14	$[\bullet PP, 3]$	SCAN from 13
15	$[\bullet P\ \ NP, 3]$	PREDICT from 14
16	$[\bullet auf\ \ NP, 3]$	PREDICT from 15
17	$[\bullet NP, 4]$	SCAN from 16
18	$[\bullet D\ \ N, 4]$	PREDICT from 17
19	$[\bullet der\ \ N, 4]$	PREDICT from 18
20	$[\bullet die\ \ N, 4]$	PREDICT from 18
21	$[\bullet N, 5]$	SCAN from 20

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ NP, 2]$	PREDICT from 9
11	$[\bullet VI\ PP, 2]$	PREDICT from 9
12	$[\bullet bescheint\ NP, 2]$	PREDICT from 10
13	$[\bullet scheint\ PP, 2]$	PREDICT from 11
14	$[\bullet PP, 3]$	SCAN from 13
15	$[\bullet P\ NP, 3]$	PREDICT from 14
16	$[\bullet auf\ NP, 3]$	PREDICT from 15
17	$[\bullet NP, 4]$	SCAN from 16
18	$[\bullet D\ N, 4]$	PREDICT from 17
19	$[\bullet der\ N, 4]$	PREDICT from 18
20	$[\bullet die\ N, 4]$	PREDICT from 18
21	$[\bullet N, 5]$	SCAN from 20
22	$[\bullet Mond, 5]$	PREDICT from 21

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ \ NP, 2]$	PREDICT from 9
11	$[\bullet VI\ \ PP, 2]$	PREDICT from 9
12	$[\bullet bescheint\ \ NP, 2]$	PREDICT from 10
13	$[\bullet scheint\ \ PP, 2]$	PREDICT from 11
14	$[\bullet PP, 3]$	SCAN from 13
15	$[\bullet P\ \ NP, 3]$	PREDICT from 14
16	$[\bullet auf\ \ NP, 3]$	PREDICT from 15
17	$[\bullet NP, 4]$	SCAN from 16
18	$[\bullet D\ \ N, 4]$	PREDICT from 17
19	$[\bullet der\ \ N, 4]$	PREDICT from 18
20	$[\bullet die\ \ N, 4]$	PREDICT from 18
21	$[\bullet N, 5]$	SCAN from 20
22	$[\bullet Mond, 5]$	PREDICT from 21
23	$[\bullet Wiese, 5]$	PREDICT from 21

Now a complete derivation - with the Schema

9	$[\bullet VP, 2]$	SCAN from 7
10	$[\bullet VT\ \ NP, 2]$	PREDICT from 9
11	$[\bullet VI\ \ PP, 2]$	PREDICT from 9
12	$[\bullet bescheint\ \ NP, 2]$	PREDICT from 10
13	$[\bullet scheint\ \ PP, 2]$	PREDICT from 11
14	$[\bullet PP, 3]$	SCAN from 13
15	$[\bullet P\ \ NP, 3]$	PREDICT from 14
16	$[\bullet auf\ \ NP, 3]$	PREDICT from 15
17	$[\bullet NP, 4]$	SCAN from 16
18	$[\bullet D\ \ N, 4]$	PREDICT from 17
19	$[\bullet der\ \ N, 4]$	PREDICT from 18
20	$[\bullet die\ \ N, 4]$	PREDICT from 18
21	$[\bullet N, 5]$	SCAN from 20
22	$[\bullet Mond, 5]$	PREDICT from 21
23	$[\bullet Wiese, 5]$	PREDICT from 21
24	$[\bullet, 6]$	SCAN from 23 - GOAL

Advantages

- Offers real-time processing of input

Advantages

- Offers real-time processing of input
- Resembles human real-time parsing

Advantages

- Offers real-time processing of input
- Resembles human real-time parsing
- Relatively easy to implement using stacks

Advantages

- Offers real-time processing of input
- Resembles human real-time parsing
- Relatively easy to implement using stacks
- Efficient in comparison with Unger parser

Advantages

- Offers real-time processing of input
- Resembles human real-time parsing
- Relatively easy to implement using stacks
- Efficient in comparison with Unger parser
- Not less efficient than CYK

Disadvantages

- LEFT RECURSION really is a problem (cf. implementation)

Disadvantages

- LEFT RECURSION really is a problem (cf. implementation)
- might check very unlikely predictions first

Disadvantages

- LEFT RECURSION really is a problem (cf. implementation)
- might check very unlikely predictions first
- no lookahead in primitive version

Disadvantages

- LEFT RECURSION really is a problem (cf. implementation)
- might check very unlikely predictions first
- no lookahead in primitive version
- Most Parsers are Non-deterministic

Implementations

- Push Down Automata

Implementations

- Push Down Automata
  - PDA, GNF and top-down are directly related

Implementations

- Push Down Automata
  - PDA, GNF and top-down are directly related
  - Their approach to the problem is very similar

Implementations

- Push Down Automata
  - PDA, GNF and top-down are directly related
  - Their approach to the problem is very similar
- Breadth first

Implementations

- Push Down Automata
  - PDA, GNF and top-down are directly related
  - Their approach to the problem is very similar
- Breadth first
  - Allow fast parsing (even on-line)

Implementations

- Push Down Automata
  - PDA, GNF and top-down are directly related
  - Their approach to the problem is very similar
- Breadth first
  - Allow fast parsing (even on-line)
  - Use lots of memory

Implementations

- Push Down Automata
  - PDA, GNF and top-down are directly related
  - Their approach to the problem is very similar
- Breadth first
  - Allow fast parsing (even on-line)
  - Use lots of memory
- Depth first

Implementations

- Push Down Automata
  - PDA, GNF and top-down are directly related
  - Their approach to the problem is very similar
- Breadth first
  - Allow fast parsing (even on-line)
  - Use lots of memory
- Depth first
  - Also called backtracking

Implementations

- Push Down Automata
  - PDA, GNF and top-down are directly related
  - Their approach to the problem is very similar
- Breadth first
  - Allow fast parsing (even on-line)
  - Use lots of memory
- Depth first
  - Also called backtracking
  - Are simple to write

Implementations

- Push Down Automata
  - PDA, GNF and top-down are directly related
  - Their approach to the problem is very similar
- Breadth first
  - Allow fast parsing (even on-line)
  - Use lots of memory
- Depth first
  - Also called backtracking
  - Are simple to write
  - Have certain problems with prefixes

Implementations

- Push Down Automata
  - PDA, GNF and top-down are directly related
  - Their approach to the problem is very similar
- Breadth first
  - Allow fast parsing (even on-line)
  - Use lots of memory
- Depth first
  - Also called backtracking
  - Are simple to write
  - Have certain problems with prefixes
- Recursive descent is a technique to implement a Depth first parser

PDA as an implementation model

- transitions of a pushdown automaton strongly resemble operations of a top-down parser:

PDA as an implementation model

- transitions of a pushdown automaton strongly resemble operations of a top-down parser:
- $\delta(q_0, \epsilon, A) = q_0, \epsilon, BC \longleftrightarrow$ PREDICT

PDA as an implementation model

- transitions of a pushdown automaton strongly resemble operations of a top-down parser:
- $\delta(q_0, \epsilon, A) = q_0, \epsilon, BC \longleftrightarrow$ PREDICT
- $\delta(q_0, a, a) = q_0, \epsilon, \epsilon \longleftrightarrow$ SCAN

PDA as an implementation model

- transitions of a pushdown automaton strongly resemble operations of a top-down parser:
- $\delta(q_0, \epsilon, A) = q_0, \epsilon, BC \longleftrightarrow$ PREDICT
- $\delta(q_0, a, a) = q_0, \epsilon, \epsilon \longleftrightarrow$ SCAN
- $\rightarrow$ in implementations, every tree hypothesis contains a stack and an input position

PDA as an implementation model

- transitions of a pushdown automaton strongly resemble operations of a top-down parser:
- $\delta(q_0, \epsilon, A) = q_0, \epsilon, BC \longleftrightarrow$ PREDICT
- $\delta(q_0, a, a) = q_0, \epsilon, \epsilon \longleftrightarrow$ SCAN
- $\rightarrow$ in implementations, every tree hypothesis contains a stack and an input position
- compare parsing schema item: $[\bullet\beta, j]$

Greibach Normal Form

- allows only productions of the following form:

Greibach Normal Form

- allows only productions of the following form:
- $A \longrightarrow aB_1 ... B_k$ with $k \geq 0$

Greibach Normal Form

- allows only productions of the following form:
- $A \longrightarrow aB_1 \ldots B_k$ with $k \geq 0$
- invented by American mathematician Sheila A. Greibach

Greibach Normal Form

- allows only productions of the following form:
- $A \longrightarrow aB_1 ... B_k$ with $k \geq 0$
- invented by American mathematician Sheila A. Greibach
- incidentally, she was also first to propose top-down-parsing

Greibach Normal Form

- allows only productions of the following form:
- $A \longrightarrow aB_1 ... B_k$ with $k \geq 0$
- invented by American mathematician Sheila A. Greibach
- incidentally, she was also first to propose top-down-parsing
- What's the relation? Why is GNF ideal for TD-parsing?

GNF and Top-Down-Parsing

- $A \longrightarrow aB_1 \ldots B_k$ with $k \geq 0$

GNF and Top-Down-Parsing

- $A \longrightarrow aB_1 ... B_k$ with $k \geq 0$
- a grammar in GNF form reduces the amount of prediction steps needed

GNF and Top-Down-Parsing

- $A \longrightarrow aB_1 ... B_k$ with $k \geq 0$
- a grammar in GNF form reduces the amount of prediction steps needed
- each prediction will result in a possible scanning step

GNF and Top-Down-Parsing

- $A \longrightarrow aB_1 ... B_k$ with $k \geq 0$
- a grammar in GNF form reduces the amount of prediction steps needed
- each prediction will result in a possible scanning step
- a wrong prediction can be discarded already with the next scanning step

GNF and Top-Down-Parsing

- $A \longrightarrow aB_1 \ldots B_k$ with $k \geq 0$
- a grammar in GNF form reduces the amount of prediction steps needed
- each prediction will result in a possible scanning step
- a wrong prediction can be discarded already with the next scanning step
- intelligent implementations would only make predictions that start with the next terminal in the input

GNF and Top-Down-Parsing

- $A \longrightarrow aB_1 ... B_k$ with $k \geq 0$
- a grammar in GNF form reduces the amount of prediction steps needed
- each prediction will result in a possible scanning step
- a wrong prediction can be discarded already with the next scanning step
- intelligent implementations would only make predictions that start with the next terminal in the input

$\rightarrow$ GNF is for TD what CNF is for BU

Breadth-first

- First described by Greibach (1964)

Breadth-first

- First described by Greibach (1964)
- Maintains a list of *possible* derivations ...

Breadth-first

- First described by Greibach (1964)
- Maintains a list of *possible* derivations ...
- ... which is kept in memory.

Breadth-first

- First described by Greibach (1964)
- Maintains a list of *possible* derivations . . .
- . . . which is kept in memory.
- Increases size of the list with every `prediction` operation

Breadth-first

- First described by Greibach (1964)
- Maintains a list of *possible* derivations . . .
- . . . which is kept in memory.
- Increases size of the list with every `prediction` operation
  - For every non-terminal *all* possible derivations are added.

Breadth-first

- First described by Greibach (1964)
- Maintains a list of *possible* derivations ...
- ... which is kept in memory.
- Increases size of the list with every `prediction` operation
  - For every non-terminal *all* possible derivations are added.
  - A LL parser `predicts` until every left-hand symbol is a terminal.

Breadth-first

- First described by Greibach (1964)
- Maintains a list of *possible* derivations . . .
- . . . which is kept in memory.
- Increases size of the list with every `prediction` operation
  - For every non-terminal *all* possible derivations are added.
  - A LL parser `predicts` until every left-hand symbol is a terminal.
- Decreases size of the list with every `matching` operation

Breadth-first

- First described by Greibach (1964)
- Maintains a list of *possible* derivations . . .
- . . . which is kept in memory.
- Increases size of the list with every `prediction` operation
  - For every non-terminal *all* possible derivations are added.
  - A LL parser `predicts` until every left-hand symbol is a terminal.
- Decreases size of the list with every `matching` operation
  - When a terminal does not match the input, the tree (prediction stack) is discarded.

Breadth-first

- First described by Greibach (1964)
- Maintains a list of *possible* derivations ...
- ... which is kept in memory.
- Increases size of the list with every `prediction` operation
  - For every non-terminal *all* possible derivations are added.
  - A LL parser `predicts` until every left-hand symbol is a terminal.
- Decreases size of the list with every `matching` operation
  - When a terminal does not match the input, the tree (prediction stack) is discarded.
- When do we stop parsing and accept?

Breadth-first

- First described by Greibach (1964)
- Maintains a list of *possible* derivations . . .
- . . . which is kept in memory.
- Increases size of the list with every `prediction` operation
  - For every non-terminal *all* possible derivations are added.
  - A LL parser `predicts` until every left-hand symbol is a terminal.
- Decreases size of the list with every `matching` operation
  - When a terminal does not match the input, the tree (prediction stack) is discarded.
- When do we stop parsing and accept?
- We introduce the end-of-input marker #

Breadth-first

- First described by Greibach (1964)
- Maintains a list of *possible* derivations ...
- ...which is kept in memory.
- Increases size of the list with every `prediction` operation
  - For every non-terminal *all* possible derivations are added.
  - A LL parser `predicts` until every left-hand symbol is a terminal.
- Decreases size of the list with every `matching` operation
  - When a terminal does not match the input, the tree (prediction stack) is discarded.
- When do we stop parsing and accept?
- We introduce the end-of-input marker #

Advantages and Shortcomings

Shortcomings

- Can be very efficient on time resources

Advantages and Shortcomings
Shortcomings

- Can be very efficient on time resources
- Can parse in real time

Advantages and Shortcomings

Shortcomings

- Can be very efficient on time resources
- Can parse in real time
- Always finds the best prediction (if written adequately)

Advantages and Shortcomings

Shortcomings

- Can be very efficient on time resources
- Can parse in real time
- Always finds the best prediction (if written adequately)

- Memory usage increases exponentially

Advantages and Shortcomings
Shortcomings

- Can be very efficient on time resources
- Can parse in real time
- Always finds the best prediction (if written adequately)

- Memory usage increases exponentially
  - This can be reduced by *Dynamic Programming techniques*
- Suffers from LEFT RECURSION

Advantages and Shortcomings

Shortcomings

- Can be very efficient on time resources
- Can parse in real time
- Always finds the best prediction (if written adequately)

- Memory usage increases exponentially
  - This can be reduced by *Dynamic Programming techniques*
- Suffers from LEFT RECURSION

Depth-first/Backtracking

- Predicts until it hits a terminal

Depth-first/Backtracking

- Predicts until it hits a terminal
- If the terminal . . .

Depth-first/Backtracking

- Predicts until it hits a terminal
- If the terminal . . .

matches: Success, go to the next higher branch and continue

Depth-first/Backtracking

- Predicts until it hits a terminal
- If the terminal ...

matches: Success, go to the next higher branch and continue
!matches: pop everything from the stack to the next higher branch

Depth-first/Backtracking

- Predicts until it hits a terminal
- If the terminal . . .

matches: Success, go to the next higher branch and continue
!matches: pop everything from the stack to the next higher branch

- Accepts when it hits the end-of-input symbol #

Depth-first/Backtracking

- Predicts until it hits a terminal
- If the terminal . . .

matches: Success, go to the next higher branch and continue
!matches: pop everything from the stack to the next higher branch

- Accepts when it hits the end-of-input symbol #

Advantages and Shortcomings

- Does not use that much memory

Advantages and Shortcomings

- Does not use that much memory
- Is easier to handle

Advantages and Shortcomings

- Does not use that much memory
- Is easier to handle
- Performance can be increased by using *statistical parsing methods*

Advantages and Shortcomings

- Does not use that much memory
- Is easier to handle
- Performance can be increased by using *statistical parsing methods*
- Generally disallows for on-line parsing

Advantages and Shortcomings

- Does not use that much memory
- Is easier to handle
- Performance can be increased by using *statistical parsing methods*
- Generally disallows for on-line parsing
- Is rather slow (can take up to exponential time)

Advantages and Shortcomings

- Does not use that much memory
- Is easier to handle
- Performance can be increased by using *statistical parsing methods*
- Generally disallows for on-line parsing
- Is rather slow (can take up to exponential time)
- The "accept first match" policy can lead to undergeneration

Advantages and Shortcomings

- Does not use that much memory
- Is easier to handle
- Performance can be increased by using *statistical parsing methods*
- Generally disallows for on-line parsing
- Is rather slow (can take up to exponential time)
- The “accept first match” policy can lead to undergeneration
- Also suffers from left recursion

Advantages and Shortcomings

- Does not use that much memory
- Is easier to handle
- Performance can be increased by using *statistical parsing methods*
- Generally disallows for on-line parsing
- Is rather slow (can take up to exponential time)
- The "accept first match" policy can lead to undergeneration
- Also suffers from left recursion

Left recursion

- An example grammar:
  S → DP VP
  DP → D NP
  NP → N | AP NP | NP PP
  VP → V DP | V PP | VP PP
  PP → P DP
  AP → A
  D → der | die | das | den | dem
  N → Fernglas | Frau | Mann | Mond | Wiese
  V → scheint | sieht
  A → kleine | kleinen | grosse | grossen
  P → auf | mit
- An example derivation for the sentence *"Der Mann sieht die Frau mit dem Fernglas"* . . .

Another derivation

that is quite problematic

- Just following the grammar . . .

Another derivation
that is quite problematic

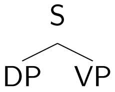
- Just following the grammar . . .

S

Another derivation

that is quite problematic

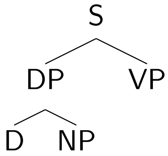
- Just following the grammar . . .



Another derivation

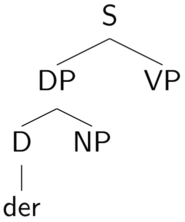
that is quite problematic

- Just following the grammar . . .



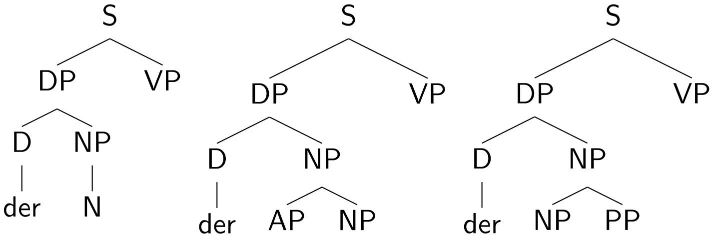
Another derivation
that is quite problematic

- Just following the grammar . . .



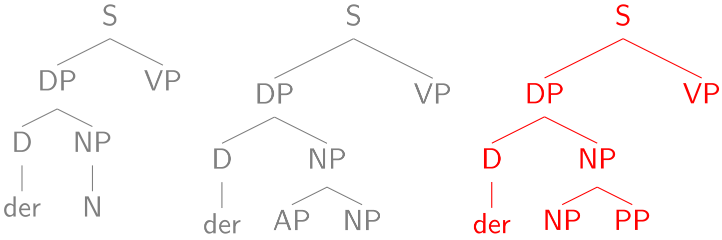
Another derivation
that is quite problematic

- Just following the grammar . . .



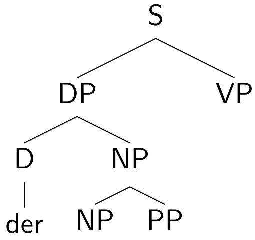
Another derivation
that is quite problematic

- Just following the grammar ...



Another derivation
that is quite problematic

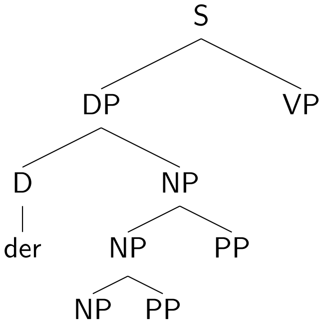
- Just following the grammar . . .



Another derivation

that is quite problematic

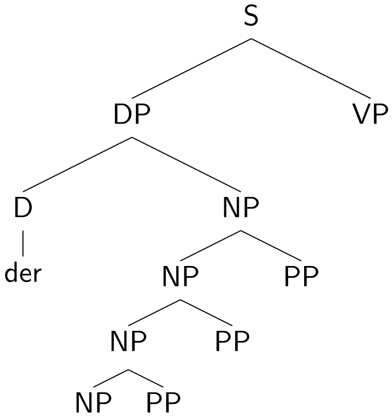
- Just following the grammar . . .



Another derivation

that is quite problematic

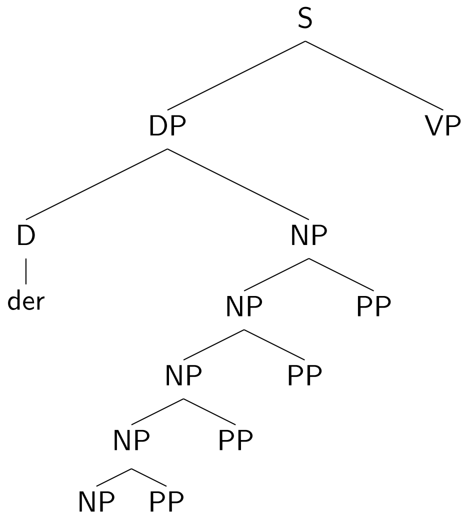
- Just following the grammar . . .



Another derivation

that is quite problematic

- Just following the grammar . . .



Left Recursion ctd.

- . . . we will get a LEFT RECURSION

Left Recursion ctd.

- ... we will get a LEFT RECURSION
- LEFT RECURSIONs can generate an infinite deal of garbage unless stopped from doing so

Left Recursion ctd.

- . . . we will get a LEFT RECURSION
- LEFT RECURSIONs can generate an infinite deal of garbage unless stopped from doing so
- There are **two types** of left-recursion:

Left Recursion ctd.

- . . . we will get a LEFT RECURSION
- LEFT RECURSIONs can generate an infinite deal of garbage unless stopped from doing so
- There are **two types** of left-recursion:
  - *Direct* left-recursion

Left Recursion ctd.

- ...we will get a LEFT RECURSION
- LEFT RECURSIONs can generate an infinite deal of garbage unless stopped from doing so
- There are **two types** of left-recursion:
  - *Direct* left-recursion

Example: NP $\rightarrow$ NP PP

Left Recursion ctd.

- ...we will get a LEFT RECURSION
- LEFT RECURSIONs can generate an infinite deal of garbage unless stopped from doing so
- There are **two types** of left-recursion:
  - *Direct* left-recursion

  Example: NP $\rightarrow$ NP PP
  - *Indirect* left-recursion

Left Recursion ctd.

- . . . we will get a LEFT RECURSION
- LEFT RECURSIONs can generate an infinite deal of garbage unless stopped from doing so
- There are **two types** of left-recursion:
  - *Direct* left-recursion

Example: NP → NP PP

  - *Indirect* left-recursion

Example: S → NP VP
VP → V' AP
V' → VP PP

Workarounds

- Keep track of the count of processed rules

Workarounds

- Keep track of the count of processed rules
  - $\rightarrow$ does not allow on-line parsing

Workarounds

- Keep track of the count of processed rules
  - $\rightarrow$ does not allow on-line parsing
- Rewrite the grammar
  - No ϵ- and unit-rules

Workarounds

- Keep track of the count of processed rules
  - $\rightarrow$ does not allow on-line parsing
- Rewrite the grammar
  - No ϵ- and unit-rules
  - Split the *direct* left-recursive rules up:
    - We start with
      NP $\rightarrow$ NP PP | N
    - And transform into:
      N′ $\rightarrow$ N
      N″ $\rightarrow$ PP
      N‴ $\rightarrow$ N″ N‴ | N″
      NP $\rightarrow$ N′ N‴ | N′

Our example revised

- Let us have a look at the previous grammar

Our example revised

- Let us have a look at the previous grammar

S → DP VP
DP → D NP
NP → N | AP NP | NP PP
VP → V DP | V PP | VP PP
PP → P DP
AP → A
D → der | die | das | den | dem
N → Fernglas | Frau | Mann | Mond | Wiese
V → scheint | sieht
A → kleine | kleinen | grosse | grossen
P → auf | mit

Our example revised

- Let us have a look at the previous grammar

S → DP VP
DP → D NP
NP → N | AP NP | NP PP
VP → V DP | V PP | VP PP
PP → P DP
AP → A
D → der | die | das | den | dem
N → Fernglas | Frau | Mann | Mond | Wiese
V → scheint | sieht
A → kleine | kleinen | grosse | grossen
P → auf | mit

Our example revised

- Let us have a look at the previous grammar

S → DP VP
DP → D NP
NP → N | AP NP | NP PP
VP → V DP | V PP | VP PP
PP → P DP
AP → A
D → der | die | das | den | dem
N → Fernglas | Frau | Mann | Mond | Wiese
V → scheint | sieht
A → kleine | kleinen | grosse | grossen
P → auf | mit

- This will now be taken care of . . .

Our example revised

- Let us have a look at the previous grammar
- Here are the revised rules:

Our example revised

- Let us have a look at the previous grammar
- Here are the revised rules:
  - VP → V DP | V PP | VP PP

    Vh → V DP | V PP

    Vt → PP

    Vts → Vt Vts | Vt

    VP → Vh Vts | Vh

Our example revised

- Let us have a look at the previous grammar
- Here are the revised rules:
  - VP → V DP | V PP | VP PP
    Vh → V DP | V PP
    Vt → PP
    Vts → Vt Vts | Vt
    VP → Vh Vts | Vh
  - NP → N | AP NP | NP PP
    Nh → AP NP | N
    Nt → PP
    Nts → Nt Nts | Nt
    NP → Nh Nts | Nh

Our example revised

- Let us have a look at the previous grammar
- Here are the revised rules:
  - VP → V DP | V PP | VP PP
    Vh → V DP | V PP
    Vt → PP
    Vts → Vt Vts | Vt
    VP → Vh Vts | Vh
  - NP → N | AP NP | NP PP
    Nh → AP NP | N
    Nt → PP
    Nts → Nt Nts | Nt
    NP → Nh Nts | Nh
- This is about three times faster than the first workaround.

Recursive Descent

- Grammars can be viewed as describing a *program*

Recursive Descent

- Grammars can be viewed as describing a *program*
- How to implement a grammar in our favorite programming language?
  One could ...

Recursive Descent

- Grammars can be viewed as describing a *program*
- How to implement a grammar in our favorite programming language?
  One could ...

Automata: ...try to emulate an *automaton* that describes the language.

Recursive Descent

- Grammars can be viewed as describing a *program*
- How to implement a grammar in our favorite programming language?
  One could ...

Automata: ...try to emulate an *automaton* that describes the language.

  - This is not very flexible.

Recursive Descent

- Grammars can be viewed as describing a *program*
- How to implement a grammar in our favorite programming language?
  One could ...

Automata: ...try to emulate an *automaton* that describes the language.

- This is not very flexible.
- May be inefficient when dealing with more complicated tasks.

Recursive Descent

- Grammars can be viewed as describing a *program*
- How to implement a grammar in our favorite programming language?
  One could . . .

Automata: . . . try to emulate an *automaton* that describes the language.

- This is not very flexible.
- May be inefficient when dealing with more complicated tasks.

Automation: . . . use a *parser-generator* (also: a *compiler-compiler*)

Recursive Descent

- Grammars can be viewed as describing a *program*
- How to implement a grammar in our favorite programming language?
  One could ...

Automata: ... try to emulate an *automaton* that describes the language.

- This is not very flexible.
- May be inefficient when dealing with more complicated tasks.

Automation: ... use a *parser-generator* (also: a *compiler-compiler*)

- May not cover our favorite programming language (or its latest version)

Recursive Descent

- Grammars can be viewed as describing a *program*
- How to implement a grammar in our favorite programming language?
  One could . . .

Automata: . . . try to emulate an *automaton* that describes the language.
- This is not very flexible.
- May be inefficient when dealing with more complicated tasks.

Automation: . . . use a *parser-generator* (also: a *compiler-compiler*)
- May not cover our favorite programming language (or its latest version)
- Could be less efficient

Recursive Descent

- Grammars can be viewed as describing a *program*
- How to implement a grammar in our favorite programming language?
  One could ...

Automata: ...try to emulate an *automaton* that describes the language.

- This is not very flexible.
- May be inefficient when dealing with more complicated tasks.

Automation: ...use a *parser-generator* (also: a *compiler-compiler*)

- May not cover our favorite programming language (or its latest version)
- Could be less efficient

Non-auto*: ...write a parser for every grammar at hand.

Recursive Descent

- Grammars can be viewed as describing a *program*
- How to implement a grammar in our favorite programming language?
  One could ...

Automata: ...try to emulate an *automaton* that describes the language.
  - This is not very flexible.
  - May be inefficient when dealing with more complicated tasks.

Automation: ...use a *parser-generator* (also: a *compiler-compiler*)
  - May not cover our favorite programming language (or its latest version)
  - Could be less efficient

Non-auto*: ...write a parser for every grammar at hand.

Recursive Descent: HOWTO

- Such parsers are implicitly deapth-first

1. Make up a function for each left-hand side

Recursive Descent: HOWTO

- Such parsers are implicitly deapth-first

1. Make up a function for each left-hand side
   - The function body represents the right hand side

Recursive Descent: HOWTO

- Such parsers are implicitly deapth-first

1. Make up a function for each left-hand side
   - The function body represents the right hand side
   - *All* functions have to return `true` for the parse to succeed

Recursive Descent: HOWTO

- Such parsers are implicitly deapth-first

1. Make up a function for each left-hand side
   - The function body represents the right hand side
   - *All* functions have to return `true` for the parse to succeed
2. Maintain the input as a global variable with a global pointer

Recursive Descent: HOWTO

- Such parsers are implicitly deapth-first

1. Make up a function for each left-hand side
   - The function body represents the right hand side
   - *All* functions have to return `true` for the parse to succeed
2. Maintain the input as a global variable with a global pointer
3. Have the methods call each other *recursively*

Recursive Descent: HOWTO

- Such parsers are implicitly deapth-first

1. Make up a function for each left-hand side
   - The function body represents the right hand side
   - *All* functions have to return `true` for the parse to succeed
2. Maintain the input as a global variable with a global pointer
3. Have the methods call each other *recursively*
4. The *base case* is when a rule hits a terminal

Recursive Descent: HOWTO

- Such parsers are implicitly deapth-first

1. Make up a function for each left-hand side
   - The function body represents the right hand side
   - *All* functions have to return `true` for the parse to succeed
2. Maintain the input as a global variable with a global pointer
3. Have the methods call each other *recursively*
4. The *base case* is when a rule hits a terminal
5. Every rule must contain it's own *pointer* to the position it points to in the input sentence

Recursive Descent: HOWTO

- Such parsers are implicitly deapth-first

1. Make up a function for each left-hand side
   - The function body represents the right hand side
   - *All* functions have to return `true` for the parse to succeed
2. Maintain the input as a global variable with a global pointer
3. Have the methods call each other *recursively*
4. The *base case* is when a rule hits a terminal
5. Every rule must contain it's own *pointer* to the position it points to in the input sentence
6. Maintain a stack of *matched predictions* to tell the derivation

Recursive Descent: HOWTO

- Such parsers are implicitly deapth-first

1. Make up a function for each left-hand side
   - The function body represents the right hand side
   - *All* functions have to return `true` for the parse to succeed
2. Maintain the input as a global variable with a global pointer
3. Have the methods call each other *recursively*
4. The *base case* is when a rule hits a terminal
5. Every rule must contain it's own *pointer* to the position it points to in the input sentence
6. Maintain a stack of *matched predictions* to tell the derivation

Certain Caveats

- Beware of left-recursion

Certain Caveats

- Beware of left-recursion
- Danger of undergeneration

Certain Caveats

- Beware of left-recursion
- Danger of undergeneration
  - Works only for *prefix-free grammars*

Certain Caveats

- Beware of left-recursion
- Danger of undergeneration
  - Works only for *prefix-free grammars*
    if
    A $\rightarrow^*$ x and A $\rightarrow^*$ xy
    this implies $y = \epsilon$
  - So one has to find a workaround for that either

Certain Caveats

- Beware of left-recursion
- Danger of undergeneration
  - Works only for *prefix-free grammars*
    if
    $A \rightarrow^* x$ and $A \rightarrow^*$ xy
    this implies $y = \epsilon$
  - So one has to find a workaround for that either
    - Be a little depth first

Summary

- LL top-down parsers can be used for on-line parsing

Summary

- LL top-down parsers can be used for on-line parsing
- They are intuitive and generally quite fast

Summary

- LL top-down parsers can be used for on-line parsing
- They are intuitive and generally quite fast
  - But the Breadth first parser is a memory hog

Summary

- LL top-down parsers can be used for on-line parsing
- They are intuitive and generally quite fast
  - But the Breadth first parser is a memory hog
  - while the Depth first parser is too slow

Summary

- LL top-down parsers can be used for on-line parsing
- They are intuitive and generally quite fast
  - But the Breadth first parser is a memory hog
  - while the Depth first parser is too slow
- Greibach Normal Form allows for comfortable parsing

Summary

- LL top-down parsers can be used for on-line parsing
- They are intuitive and generally quite fast
  - But the Breadth first parser is a memory hog
  - while the Depth first parser is too slow
- Greibach Normal Form allows for comfortable parsing
- GNF is for top-down what CNF is for bottom-up

Summary

- LL top-down parsers can be used for on-line parsing
- They are intuitive and generally quite fast
  - But the Breadth first parser is a memory hog
  - while the Depth first parser is too slow
- Greibach Normal Form allows for comfortable parsing
- GNF is for top-down what CNF is for bottom-up
- Recursive descent allows for easy implementation of a backtracking parser

Summary

- LL top-down parsers can be used for on-line parsing
- They are intuitive and generally quite fast
  - But the Breadth first parser is a memory hog
  - while the Depth first parser is too slow
- Greibach Normal Form allows for comfortable parsing
- GNF is for top-down what CNF is for bottom-up
- Recursive descent allows for easy implementation of a backtracking parser

Thanks a lot.