Introduction
Intuitive example
Features and Operations
Parsing Schema

Motivation
Why Top-Down?
Why Not?

Implementation
PDA + GNF
Breadth-first
Depth-first
Left recursion
Recursive Descent

Summary

Top Down Parsing

Johannes Dellert Aleksandar Dimitrov
Seminar für Sprachwissenschaft, Universität Tübingen
December 2006
Outline

Introduction

Intuitive example
Features and Operations
Parsing Schema

Motivation

Why Top-Down?
Why Not?

Implementation

PDA + GNF
Breadth-first
Depth-first
Left recursion
Recursive Descent

Summary
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
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
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':
Now let us parse the sentence 'der Mond scheint auf die Wiese'

First 'tree idea':
Intuitive example

'Tree idea' is expanded via leftmost derivations:

```
S
   \---\---
   NP   VP
```
'Tree idea' is expanded via leftmost derivations:

```
S
  └── NP
      └── D
    └── N
  └── VP
```
'Tree idea' is expanded via leftmost derivations:

```
S
  NP  VP
    D  N
     der
```

- NP
- VP
- D
- N
- der
Intuitive example

Tree begins to match input and is expanded:

```
S
|-- NP
|  |-- D
|  |  |  N
|  |  |  |
|  |  |  der
|-- VP
```
Top Down Parsing

Johannes Dellert,
Aleksandar Dimitrov

Introduction
Intuitive example
Features and Operations
Parsing Schema

Motivation
Why Top-Down?
Why Not?

Implementation
PDA + GNF
Breadth-first
Depth-first
Left recursion
Recursive Descent

Summary

Intuitive example

Tree begins to match input and is expanded:

```
S
  NP    VP
  |      |
D  N    |
  |      |
der  Mond
```
Intuitive example

► Tree begins to match input and is expanded:

```
S
  ▼
NP  VP
  ▼  ▼
D   N
  ▼ ▼
der  Mond
```
Intuitive example

Nondeterminism: Two different possible trees.
Intuitive example

Nondeterminism: Expanding both trees.
Intuitive example

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

S

NP       VP

D         N       VI        PP

|     |     |     |
der  Mond scheint
Intuitive example

Expanding the predicted tree

S
  └── NP
      └── D der
      └── N Mond

  └── VP
      └── VI scheint
            └── PP P NP
Intuitive example

Expanding the predicted tree

```
S
  NP
    D
    der
  VP
    VI
    scheint
    PP
    auf
    NP
```
Intuitive example

Expanding the predicted tree

S

NP
  D
  der

  N
  Mond

VP
  VI
  scheint

  PP
  P
  auf

NP
Intuitive example

Expanding the predicted tree

S
  └── NP
    └── D
      └── der
    └── N
      └── Mond
  └── VP
    └── VI
      └── scheint
    └── PP
      └── P
        └── auf
            └── D
                └── N
Intuitive example

Expanding the predicted tree

S
  └─ NP
     └─ D
         └─ der
  └─ VP
     └─ VI
         └─ scheint
     └─ PP
         └─ P
             └─ auf
                 └─ NP
                     └─ D
                         └─ die
Intuitive example

Expanding the predicted tree

S

NP          VP
  D          VI
  |          PP
  N  scheint
  |  auf
  der  Mond  P
  die  NP
  die  NP
Intuitive example

Expanding the predicted tree

S
  NP
    D
      der
  VP
    VI
      scheint
    PP
      P
        auf
      NP
        D
          die
        N
          Wiese
Top Down Parsing

Johannes Dellert, Aleksandar Dimitrov

Introduction

Intuitive example

Features and Operations

Parsing Schema

Motivation

Why Top-Down?

Why Not?

Implementation

PDA + GNF

Breadth-first

Depth-first

Left recursion

Recursive Descent

Summary

Intuitive example

▶ Tree completed

S

NP

D

der

N

Mond

VP

VI

scheint

PP

P

auf

NP

D

die

N

Wiese
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 → der Mond V PP for a rule VP → 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 → 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
  - e.g. der N VP → N VP, 'der' matched in the input string
 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 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: [$\beta, j$]
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: $\beta$ parts of the tree to be 'filled', $j$ current position in input string
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: \(\beta\) 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: \(\beta\) 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
How do we formalize the scanning step?
How do we formalize the scanning step?

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

\[ [\bullet \beta, j + 1] \] (1)

How do we formalize the prediction step?

\[ B \rightarrow \gamma \] (2)
How do we formalize the scanning step?

\[
\begin{align*}
[\bullet w_{j+1}\beta, j] \\
[\bullet \beta, j + 1]
\end{align*}
\]  

How do we formalize the prediction step?

\[
\begin{align*}
[\bullet B\beta, j] \\
[\bullet \gamma\beta, j]
\end{align*}
\]
\[B \rightarrow \gamma\]
Now a complete derivation - with the Schema

1  
[•$S$, 0]  INITIALIZING
Now a complete derivation - with the Schema

1. $\cdot S, 0 \cdot$ INITIALIZE
2. $\cdot NP \ VP, 0 \cdot$ PREDICT from 1
Top Down Parsing

Johannes Dellert, Aleksandar Dimitrov

Introduction

Intuitive example
Features and Operations
Parsing Schema

Motivation

Why Top-Down?
Why Not?

Implementation

PDA + GNF
Breadth-first
Depth-first
Left recursion
Recursive Descent

Summary

Now a complete derivation - with the Schema

1  [●S, 0]  INITIALIZE
2  [●NP VP, 0]  PREDICT from 1
3  [●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. \([\cdot S, 0]\) INITIALIZE
2. \([\cdot NP \ VP, 0]\) PREDICT from 1
3. \([\cdot D \ N \ VP, 0]\) PREDICT from 2
4. \([\cdot der \ N \ VP, 0]\) PREDICT from 3
5. \([\cdot die \ N \ VP, 0]\) 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. $[\bullet \text{S}, 0]$ INITIALIZE
2. $[\bullet \text{NP VP}, 0]$ PREDICT from 1
3. $[\bullet \text{D N VP}, 0]$ PREDICT from 2
4. $[\bullet \text{der N VP}, 0]$ PREDICT from 3
5. $[\bullet \text{die N VP}, 0]$ PREDICT from 3
6. $[\bullet \text{N VP}, 1]$ SCAN from 4
7. $[\bullet \text{Mond VP}, 1]$ PREDICT from 6
Now a complete derivation - with the Schema

1. [•S, 0] INITIALIZE
2. [•NP VP, 0] PREDICT from 1
3. [•D N VP, 0] PREDICT from 2
4. [•der N VP, 0] PREDICT from 3
5. [•die N VP, 0] PREDICT from 3
6. [•N VP, 1] SCAN from 4
7. [•Mond VP, 1] PREDICT from 6
8. [•Wiese VP, 1] PREDICT from 6
Now a complete derivation - with the Schema

1. [●S, 0] INITIALIZE
2. [●NP VP, 0] PREDICT from 1
3. [●D N VP, 0] PREDICT from 2
4. [●der N VP, 0] PREDICT from 3
5. [●die N VP, 0] PREDICT from 3
6. [●N VP, 1] SCAN from 4
7. [●Mond VP, 1] PREDICT from 6
8. [●Wiese VP, 1] PREDICT from 6
9. [●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  [● VP, 2]  SCAN from 7
10  [● VT NP, 2]  PREDICT from 9
Now a complete derivation - with the Schema

9 [●VP, 2] SCAN from 7
10 [●VT NP, 2] PREDICT from 9
11 [●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

\[
\begin{align*}
9 & \quad \text{[VP, 2]} \quad \text{SCAN from 7} \\
10 & \quad \text{[VT NP, 2]} \quad \text{PREDICT from 9} \\
11 & \quad \text{[VI PP, 2]} \quad \text{PREDICT from 9} \\
12 & \quad \text{[bescheint NP, 2]} \quad \text{PREDICT from 10} \\
13 & \quad \text{[scheint PP, 2]} \quad \text{PREDICT from 11}
\end{align*}
\]
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  [●VP, 2] SCAN from 7
10 [●VT NP, 2] PREDICT from 9
11 [●VI PP, 2] PREDICT from 9
12 [●bescheint NP, 2] PREDICT from 10
13 [●scheint PP, 2] PREDICT from 11
14 [●PP, 3] SCAN from 13
15 [●P NP, 3] PREDICT from 14
Now a complete derivation - with the Schema

9 \[\bullet VP, 2\] \text{SCAN from 7}
10 \[\bullet VT NP, 2\] \text{PREDICT from 9}
11 \[\bullet VI PP, 2\] \text{PREDICT from 9}
12 \[\bullet bescheint NP, 2\] \text{PREDICT from 10}
13 \[\bullet scheint PP, 2\] \text{PREDICT from 11}
14 \[\bullet PP, 3\] \text{SCAN from 13}
15 \[\bullet P NP, 3\] \text{PREDICT from 14}
16 \[\bullet auf NP, 3\] \text{PREDICT from 15}
Now a complete derivation - with the Schema

9  [•VP, 2] SCAN from 7
10 [•VT NP, 2] PREDICT from 9
11 [•VI PP, 2] PREDICT from 9
12 [•bescheint NP, 2] PREDICT from 10
13 [•scheint PP, 2] PREDICT from 11
14 [•PP, 3] SCAN from 13
15 [•P NP, 3] PREDICT from 14
16 [•auf NP, 3] PREDICT from 15
17 [•NP, 4] SCAN from 16
Now a complete derivation - with the Schema

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>[●VP, 2] SCAN from 7</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>[●VT NP, 2] PREDICT from 9</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>[●VI PP, 2] PREDICT from 9</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>[●bescheint NP, 2] PREDICT from 10</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>[●scheint PP, 2] PREDICT from 11</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>[●PP, 3] SCAN from 13</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>[●P NP, 3] PREDICT from 14</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>[●auf NP, 3] PREDICT from 15</td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>[●NP, 4] SCAN from 16</td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>[●D N, 4] PREDICT from 17</td>
<td></td>
</tr>
</tbody>
</table>
Now a complete derivation - with the Schema

<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
<th>Stack Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>SCAN from 7</td>
<td>[•VP, 2]</td>
</tr>
<tr>
<td>10</td>
<td>PREDICT from 9</td>
<td>[•VT NP, 2]</td>
</tr>
<tr>
<td>11</td>
<td>PREDICT from 9</td>
<td>[•VI PP, 2]</td>
</tr>
<tr>
<td>12</td>
<td>PREDICT from 10</td>
<td>[•bescheint NP, 2]</td>
</tr>
<tr>
<td>13</td>
<td>PREDICT from 11</td>
<td>[•scheint PP, 2]</td>
</tr>
<tr>
<td>14</td>
<td>SCAN from 13</td>
<td>[•PP, 3]</td>
</tr>
<tr>
<td>15</td>
<td>PREDICT from 14</td>
<td>[•P NP, 3]</td>
</tr>
<tr>
<td>16</td>
<td>PREDICT from 15</td>
<td>[•auf NP, 3]</td>
</tr>
<tr>
<td>17</td>
<td>SCAN from 16</td>
<td>[•NP, 4]</td>
</tr>
<tr>
<td>18</td>
<td>PREDICT from 17</td>
<td>[•D N, 4]</td>
</tr>
<tr>
<td>19</td>
<td>PREDICT from 18</td>
<td>[•der N, 4]</td>
</tr>
</tbody>
</table>
Now a complete derivation - with the Schema

<table>
<thead>
<tr>
<th>Line</th>
<th>Parse</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>[●VP, 2]</td>
<td>SCAN from 7</td>
</tr>
<tr>
<td>10</td>
<td>[●VT NP, 2]</td>
<td>PREDICT from 9</td>
</tr>
<tr>
<td>11</td>
<td>[●VI PP, 2]</td>
<td>PREDICT from 9</td>
</tr>
<tr>
<td>12</td>
<td>[●bescheint NP, 2]</td>
<td>PREDICT from 10</td>
</tr>
<tr>
<td>13</td>
<td>[●scheint PP, 2]</td>
<td>PREDICT from 11</td>
</tr>
<tr>
<td>14</td>
<td>[●PP, 3]</td>
<td>SCAN from 13</td>
</tr>
<tr>
<td>15</td>
<td>[●P NP, 3]</td>
<td>PREDICT from 14</td>
</tr>
<tr>
<td>16</td>
<td>[●auf NP, 3]</td>
<td>PREDICT from 15</td>
</tr>
<tr>
<td>17</td>
<td>[●NP, 4]</td>
<td>SCAN from 16</td>
</tr>
<tr>
<td>18</td>
<td>[●D N, 4]</td>
<td>PREDICT from 17</td>
</tr>
<tr>
<td>19</td>
<td>[●der N, 4]</td>
<td>PREDICT from 18</td>
</tr>
<tr>
<td>20</td>
<td>[●die N, 4]</td>
<td>PREDICT from 18</td>
</tr>
</tbody>
</table>
Top Down Parsing
Johannes Dellert, Aleksandar Dimitrov

Introduction
Intuitive example
Features and Operations
Parsing Schema

Motivation
Why Top-Down?
Why Not?

Implementation
PDA + GNF
Breadth-first
Depth-first
Left recursion
Recursive Descent

Summary

Now a complete derivation - with the Schema

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>[VP, 2]</td>
<td>SCAN from 7</td>
</tr>
<tr>
<td>10</td>
<td>[VT NP, 2]</td>
<td>PREDICT from 9</td>
</tr>
<tr>
<td>11</td>
<td>[VI PP, 2]</td>
<td>PREDICT from 9</td>
</tr>
<tr>
<td>12</td>
<td>[beschein NP, 2]</td>
<td>PREDICT from 10</td>
</tr>
<tr>
<td>13</td>
<td>[scheint PP, 2]</td>
<td>PREDICT from 11</td>
</tr>
<tr>
<td>14</td>
<td>[PP, 3]</td>
<td>SCAN from 13</td>
</tr>
<tr>
<td>15</td>
<td>[P NP, 3]</td>
<td>PREDICT from 14</td>
</tr>
<tr>
<td>16</td>
<td>[auf NP, 3]</td>
<td>PREDICT from 15</td>
</tr>
<tr>
<td>17</td>
<td>[NP, 4]</td>
<td>SCAN from 16</td>
</tr>
<tr>
<td>18</td>
<td>[D N, 4]</td>
<td>PREDICT from 17</td>
</tr>
<tr>
<td>19</td>
<td>[der N, 4]</td>
<td>PREDICT from 18</td>
</tr>
<tr>
<td>20</td>
<td>[die N, 4]</td>
<td>PREDICT from 18</td>
</tr>
<tr>
<td>21</td>
<td>[N, 5]</td>
<td>SCAN from 20</td>
</tr>
</tbody>
</table>
Now a complete derivation - with the Schema

9  [•VP, 2]  SCAN from 7
10 [•VT NP, 2]  PREDICT from 9
11 [•VI PP, 2]  PREDICT from 9
12 [•bescheint NP, 2]  PREDICT from 10
13 [•scheint PP, 2]  PREDICT from 11
14 [•PP, 3]  SCAN from 13
15 [•P NP, 3]  PREDICT from 14
16 [•auf NP, 3]  PREDICT from 15
17 [•NP, 4]  SCAN from 16
18 [•D N, 4]  PREDICT from 17
19 [•der N, 4]  PREDICT from 18
20 [•die N, 4]  PREDICT from 18
21 [•N, 5]  SCAN from 20
22 [•Mond, 5]  PREDICT from 21
Now a complete derivation - with the Schema

9  [●VP, 2]  SCAN from 7
10 [●VT NP, 2]  PREDICT from 9
11 [●VI PP, 2]  PREDICT from 9
12 [●bescheint NP, 2]  PREDICT from 10
13 [●scheint PP, 2]  PREDICT from 11
14 [●PP, 3]  SCAN from 13
15 [●P NP, 3]  PREDICT from 14
16 [●auf NP, 3]  PREDICT from 15
17 [●NP, 4]  SCAN from 16
18 [●D N, 4]  PREDICT from 17
19 [●der N, 4]  PREDICT from 18
20 [●die N, 4]  PREDICT from 18
21 [●N, 5]  SCAN from 20
22 [●Mond, 5]  PREDICT from 21
23 [●Wiese, 5]  PREDICT from 21
Now a complete derivation - with the Schema

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>[●VP, 2]</td>
<td>SCAN from 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>[●VT NP, 2]</td>
<td>PREDICT from 9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>[●VI PP, 2]</td>
<td>PREDICT from 9</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>[●bescheint NP, 2]</td>
<td>PREDICT from 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>[●scheint PP, 2]</td>
<td>PREDICT from 11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>[●PP, 3]</td>
<td>SCAN from 13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>[●P NP, 3]</td>
<td>PREDICT from 14</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>[●auf NP, 3]</td>
<td>PREDICT from 15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>[●NP, 4]</td>
<td>SCAN from 16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>[●D N, 4]</td>
<td>PREDICT from 17</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>[●der N, 4]</td>
<td>PREDICT from 18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>[●die N, 4]</td>
<td>PREDICT from 18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>[●N, 5]</td>
<td>SCAN from 20</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>[●Mond, 5]</td>
<td>PREDICT from 21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>23</td>
<td>[●Wiese, 5]</td>
<td>PREDICT from 21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>[●, 6]</td>
<td>SCAN from 23 - GOAL</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:
  - $\delta(q_0, \epsilon, A) = q_0, \epsilon, BC \xrightarrow{\text{PREDICT}}$
transitions of a pushdown automaton strongly resemble operations of a top-down parser:

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

- $\delta(q_0, \epsilon, A) = q_0, \epsilon, BC \leftarrow \rightarrow$ PREDICT
- $\delta(q_0, a, a) = q_0, \epsilon, \epsilon \leftarrow \rightarrow$ SCAN

→ in implementations, every tree hypothesis contains a stack and an input position
transitions of a pushdown automaton strongly resemble operations of a top-down parser:

\[ \delta(q_0, \epsilon, A) = q_0, \epsilon, BC \longleftrightarrow \text{PREDICT} \]

\[ \delta(q_0, a, a) = q_0, \epsilon, \epsilon \longleftrightarrow \text{SCAN} \]

→ 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:
- $A \rightarrow aB_1...B_k$ with $k \geq 0$
Greibach Normal Form

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

Greibach Normal Form

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

- allows only productions of the following form:
- \( A \rightarrow aB_1...B_k \text{ with } k \geq 0 \)
- 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:
  \[ A \longrightarrow aB_1...B_k \text{ with } k \geq 0 \]
- 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

- $A \rightarrow aB_1 \ldots B_k$ with $k \geq 0$
GNF and Top-Down-Parsing

- $A \rightarrow aB_1...B_k$ with $k \geq 0$
- a grammar in GNF form reduces the amount of prediction steps needed
GNF and Top-Down-Parsing

- $A \longrightarrow aB_1 \ldots B_k$ 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
GNF and Top-Down-Parsing

- $A \rightarrow aB_1...B_k$ 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

- $A \rightarrow aB_1...B_k$ 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
GNF and Top-Down-Parsing

- \[ A \rightarrow aB_1 \ldots B_k \text{ with } k \geq 0 \]

- 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 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.
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 . . .

```
S
   /\   \\
  DP  VP
```
Another derivation
that is quite problematic

▶ Just following the grammar . . .

S

DP     VP

D    NP
Another derivation that is quite problematic

- Just following the grammar . . .

```
S
  /\       
DP  VP
  /    |
D    NP
     /|
    der
```
Another derivation that is quite problematic

Just following the grammar . . .

S
   ▶ DP
      der
      N
      ▶ NP
      ▶ VP
      ▶ S
      ▶ DP
      der
      ▶ NP
      ▶ AP
      ▶ NP
      ▶ VP
      ▶ S
      ▶ DP
      der
      ▶ NP
      ▶ NP
      ▶ PP
Another derivation that is quite problematic

▶ Just following the grammar . . .

S
   /   
  /    
DP    VP
   /    |
  /     D
 /      |
der   NP
 |
 N

S
   /   
  /    
DP    VP
   /    |
  /     D
 /      |
der   NP
 |
 AP    NP
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
 |
Another derivation
that is quite problematic

- Just following the grammar . . .

```
S
  |   |
  DP VP
   |   |
  D NP
   |   |
  der NP PP
```

Another derivation that is quite problematic

- Just following the grammar . . .

```
S
  |   |
  DP VP
   |   |
  D NP
   |   |
  der NP PP
```

Another derivation that is quite problematic

- Just following the grammar . . .

```
S
  |   |
  DP VP
   |   |
  D NP
   |   |
  der NP PP
```
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 ...
...we will get a **LEFT RECURRENCE**
...we will get a **LEFT RECURSION**

**LEFT RECURSIONS** can generate an infinite deal of garbage unless stopped from doing so
... 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:
... 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
...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
...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
...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
  - Example: \( S \rightarrow NP \ VP \\
  \quad VP \rightarrow V' \ AP \\
  \quad V' \rightarrow VP \ PP \)
Workarounds

- Keep track of the count of processed rules
Workarounds

- Keep track of the count of processed rules
  - → does not allow on-line parsing
Workarounds

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

- Keep track of the count of processed rules
  - → does not allow on-line parsing
- Rewrite the grammar
  - No \(\epsilon\)- and unit-rules
  - Split the \textit{direct} left-recursive rules up:
    - We start with
      \[
      \text{NP} \rightarrow \text{NP PP} | \text{N}
      \]
    - And transform into:
      \[
      \text{N'} \rightarrow \text{N} \\
      \text{N''} \rightarrow \text{PP} \\
      \text{N'''} \rightarrow \text{N'' N'''} | \text{N'} \\
      \text{NP} \rightarrow \text{N' N''' | N'}
      \]
Our example revised

- Let us have a look at the previous grammar
Our example revised

Let us have a look at the previous grammar

\[
\begin{align*}
S & \rightarrow DP \ VP \\
DP & \rightarrow D \ NP \\
NP & \rightarrow N \mid AP \ NP \mid NP \ PP \\
VP & \rightarrow V \ DP \mid V \ PP \mid VP \ PP \\
PP & \rightarrow P \ DP \\
AP & \rightarrow A \\
D & \rightarrow \text{der} \mid \text{die} \mid \text{das} \mid \text{den} \mid \text{dem} \\
N & \rightarrow \text{Fernglas} \mid \text{Frau} \mid \text{Mann} \mid \text{Mond} \mid \text{Wiese} \\
V & \rightarrow \text{scheint} \mid \text{sieht} \\
A & \rightarrow \text{kleine} \mid \text{klein} \mid \text{grosse} \mid \text{grossen} \\
P & \rightarrow \text{auf} \mid \text{mit}
\end{align*}
\]
Our example revised

Let us have a look at the previous grammar:

\[
\begin{align*}
S & \rightarrow \text{DP \ VP} \\
\text{DP} & \rightarrow \text{D \ NP} \\
\text{NP} & \rightarrow \text{N} \mid \text{AP \ NP} \mid \text{NP \ PP} \\
\text{VP} & \rightarrow \text{V \ DP} \mid \text{V \ PP} \mid \text{VP \ PP} \\
\text{PP} & \rightarrow \text{P \ DP} \\
\text{AP} & \rightarrow \text{A} \\
\text{D} & \rightarrow \text{der} \mid \text{die} \mid \text{das} \mid \text{den} \mid \text{dem} \\
\text{N} & \rightarrow \text{Fernglas} \mid \text{Frau} \mid \text{Mann} \mid \text{Mond} \mid \text{Wiese} \\
\text{V} & \rightarrow \text{scheint} \mid \text{sieht} \\
\text{A} & \rightarrow \text{kleine} \mid \text{kleinen} \mid \text{grosse} \mid \text{grossen} \\
\text{P} & \rightarrow \text{auf} \mid \text{mit}
\end{align*}
\]
Our example revised

- Let us have a look at the previous grammar

\[
S \rightarrow \text{DP VP} \\
\text{DP} \rightarrow \text{D NP} \\
\text{NP} \rightarrow \text{N} \mid \text{AP NP} \mid \text{NP PP} \\
\text{VP} \rightarrow \text{V DP} \mid \text{V PP} \mid \text{VP PP} \\
\text{PP} \rightarrow \text{P DP} \\
\text{AP} \rightarrow \text{A} \\
\text{D} \rightarrow \text{der} \mid \text{die} \mid \text{das} \mid \text{den} \mid \text{dem} \\
\text{N} \rightarrow \text{Fernglas} \mid \text{Frau} \mid \text{Mann} \mid \text{Mond} \mid \text{Wiese} \\
\text{V} \rightarrow \text{scheint} \mid \text{sieht} \\
\text{A} \rightarrow \text{kleine} \mid \text{klein} \mid \text{grosse} \mid \text{grossen} \\
\text{P} \rightarrow \text{auf} \mid \text{mit}
\]

- 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:

  - \( VP \rightarrow V \ DP \mid V \ PP \mid VP \ PP \)
  - \( Vh \rightarrow V \ DP \mid V \ PP \)
  - \( Vt \rightarrow PP \)
  - \( Vts \rightarrow Vt \ Vts \mid Vt \)
  - \( VP \rightarrow Vh \ Vts \mid Vh \)

This is about three times faster than the first workaround.
Our example revised

- Let us have a look at the previous grammar
- Here are the revised rules:
  - \( VP \rightarrow V \, DP \mid V \, PP \mid VP \, PP \)
  - \( Vh \rightarrow V \, DP \mid V \, PP \)
  - \( Vt \rightarrow PP \)
  - \( Vts \rightarrow Vt \, Vts \mid Vt \)
  - \( VP \rightarrow Vh \, Vts \mid Vh \)
  - \( NP \rightarrow N \mid AP \, NP \mid NP \, PP \)
  - \( Nh \rightarrow AP \, NP \mid N \)
  - \( Nt \rightarrow PP \)
  - \( Nts \rightarrow Nt \, Nts \mid Nt \)
  - \( NP \rightarrow Nh \, Nts \mid Nh \)
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.
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.
Grammars can be viewed as describing a \textit{program}

How to implement a grammar in our favorite programming language?
One could . . .

\textbf{Automata:} . . . try to emulate an \textit{automaton} that describes the language.

\textbullet{} This is not very flexible.
\textbullet{} May be inefficient when dealing with more complicated tasks.
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)
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 depth-first

1. Make up a function for each left-hand side
Such parsers are implicitly depth-first

1. Make up a function for each left-hand side
   ▶ The function body represents the right hand side
Such parsers are implicitly depth-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 depth-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 depth-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*
Such parsers are implicitly depth-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 depth-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 its own *pointer* to the position it points to in the input sentence
Such parsers are implicitly depth-first

1. Make up a function for each left-hand side
   - The function body represents the right hand side
   - All functions have to return \texttt{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 \textit{recursively}

4. The \textit{base case} is when a rule hits a \texttt{terminal}

5. Every rule must contain it’s own \texttt{pointer} to the position it points to in the input sentence

6. Maintain a stack of \textit{matched predictions} to tell the derivation
Recursive Descent: HOWTO

- Such parsers are implicitly depth-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 →* x and A →* xy
  - this implies y = ε
  - 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.