DD1361
Programmeringsparadigm

Formella Språk & Syntaxanalys
Föreläsning 3

Per Austrin

2015-11-13

Snabb repetition

Huvudkoncept hittils:

Formellt språk

- en mängd strängar

Reguljära språk

- den klass av formella språk som kan beskrivas med reguljära uttryck eller ändliga automater (DFA:er)
- viktigt: reguljära uttryck och automater lika kraftfulla, kan beskriva samma saker

Grammatiker

- verktyg för att beskriva språk som med enkla rekursiva definitioner

Stackautomater

- automat med minne i form av en stack

Idag

Stackautomater (forts.)

Lexikal analys

Härledningar och syntaxträd

Omskrivning av grammatiker

Rekursiv medåkning och LL(1)-grammatiker

Idag

Stackautomater (forts.)

Lexikal analys

Härledningar och syntaxträd

Omskrivning av grammatiker

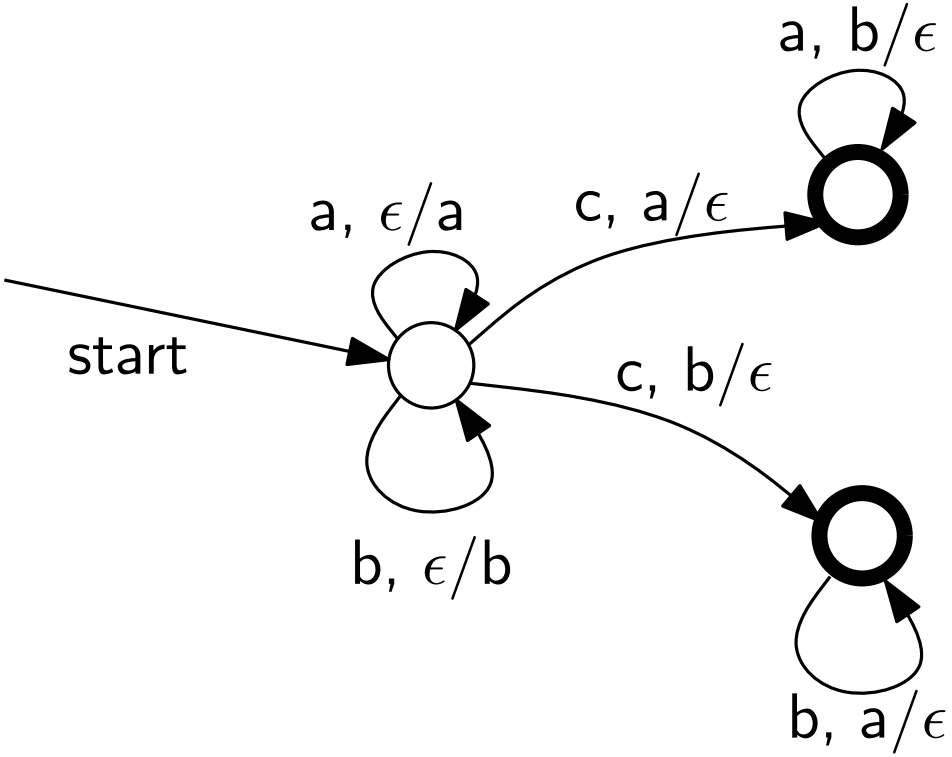
Rekursiv medåkning och LL(1)-grammatiker

Stackautomater (repetition från förra fredagen)

Stackautomat (PDA, Push-Down Automaton):
som DFA, men har ett obegränsat minne i form av en stack

Stackautomater (repetition från förra fredagen)

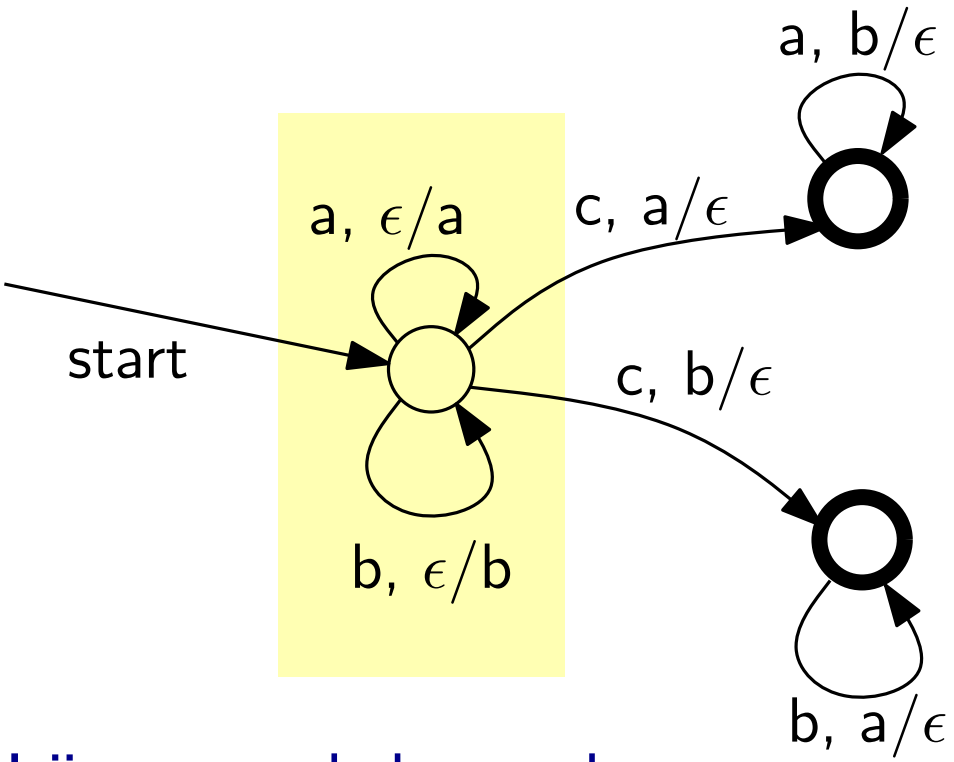
Stackautomat (PDA, Push-Down Automaton):
som DFA, men har ett obegränsat minne i form av en stack



x, y/z: läs x, poppa y
från stacken, pusha z

Stackautomater (repetition från förra fredagen)

Stackautomat (PDA, Push-Down Automaton):
som DFA, men har ett obegränsat minne i form av en stack

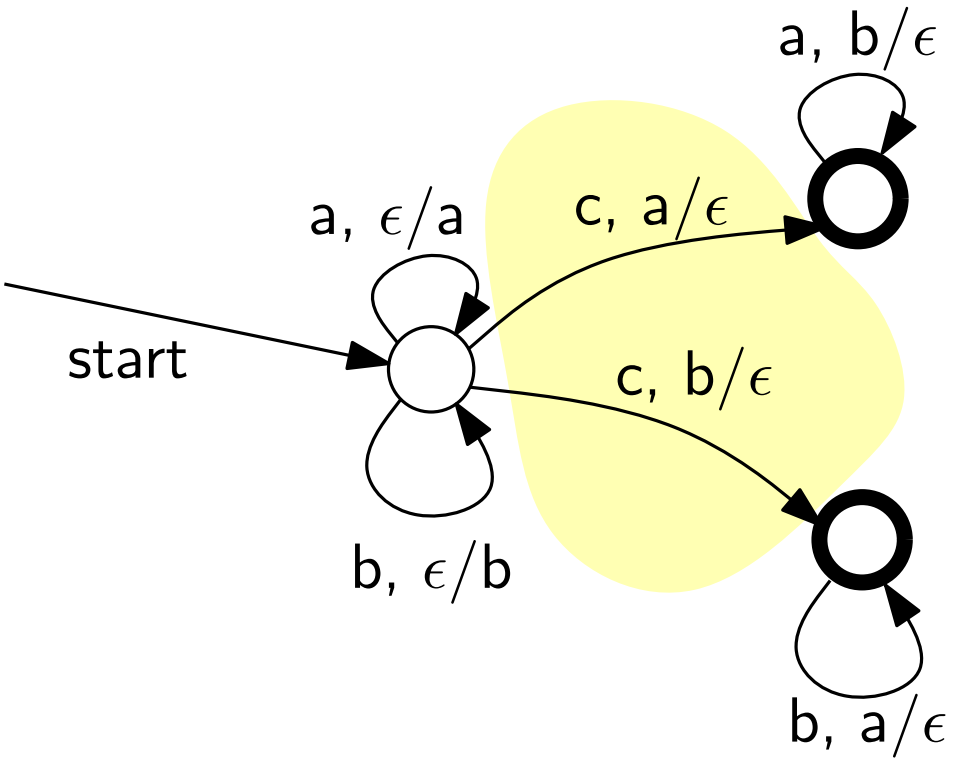


x, y/z: läs x, poppa y från stacken, pusha z

Läs a:n och b:n och lägg på stacken

Stackautomater (repetition från förra fredagen)

Stackautomat (PDA, Push-Down Automaton):
som DFA, men har ett obegränsat minne i form av en stack

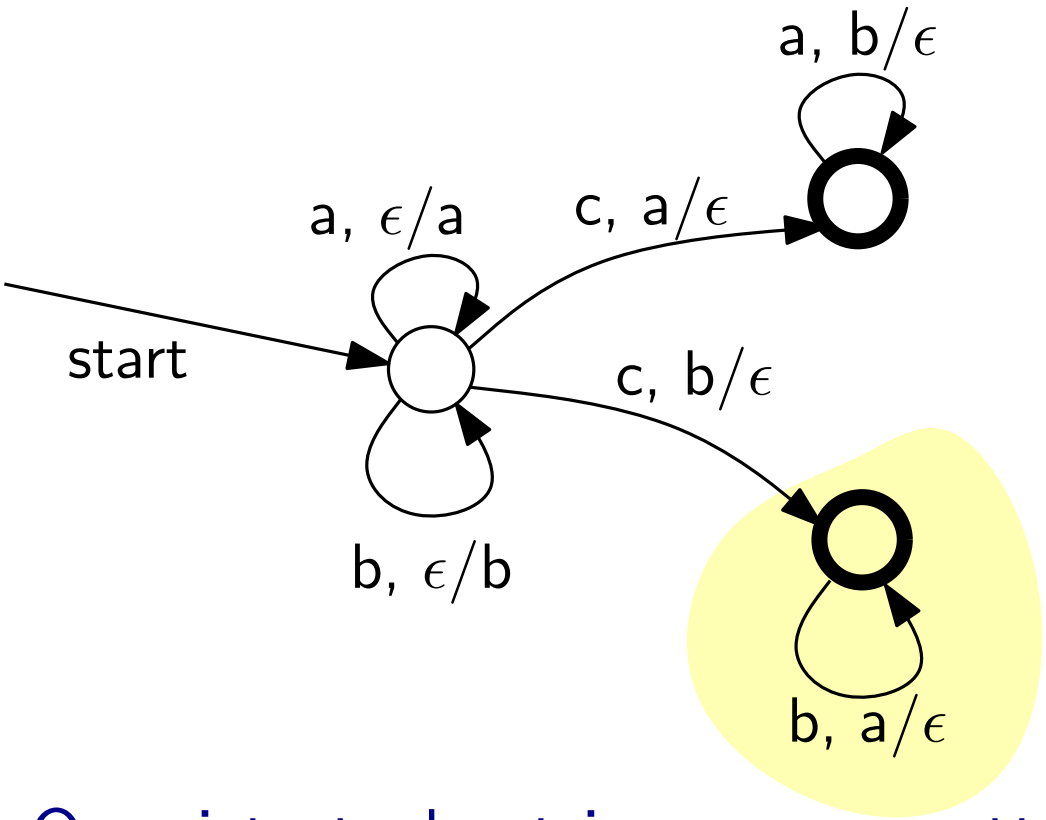


x, y/z: läs x, poppa y från stacken, pusha z

När det kommer ett c, hoppa till något av de accepterande tillstånden beroende på om det översta tecknet på stacken är a eller b

Stackautomater (repetition från förra fredagen)

Stackautomat (PDA, Push-Down Automaton):
som DFA, men har ett obegränsat minne i form av en stack

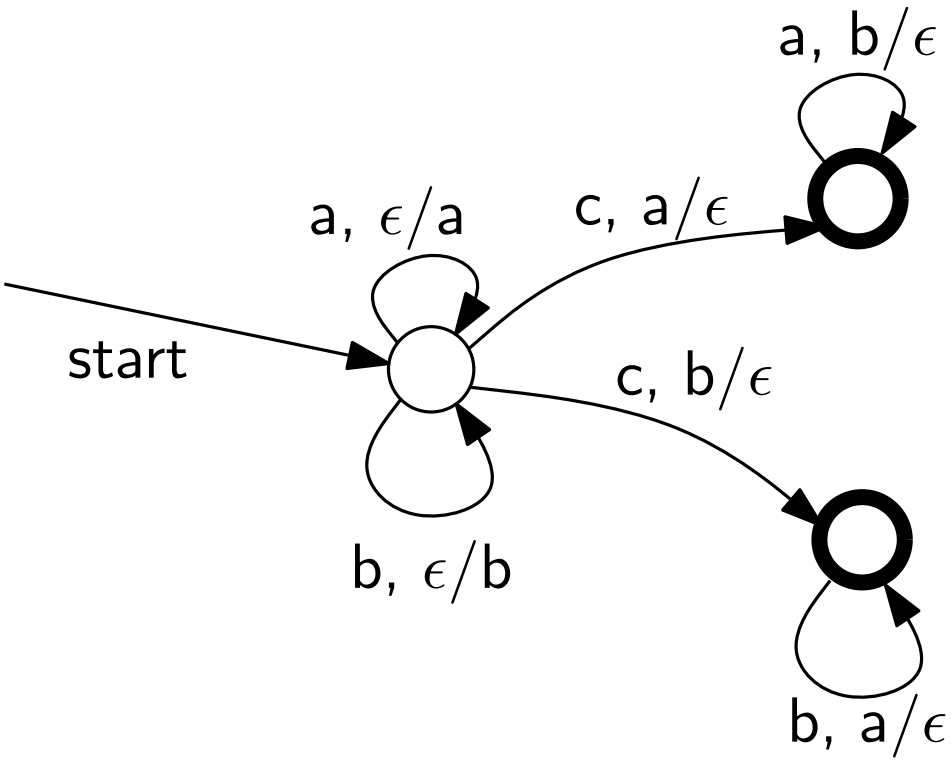


x, y/z: läs x, poppa y från stacken, pusha z

Om sista tecknet innan c var ett b: fortsätt läsa b:n från indata och kolla att det ligger minst lika många a:n på stacken

Stackautomater (repetition från förra fredagen)

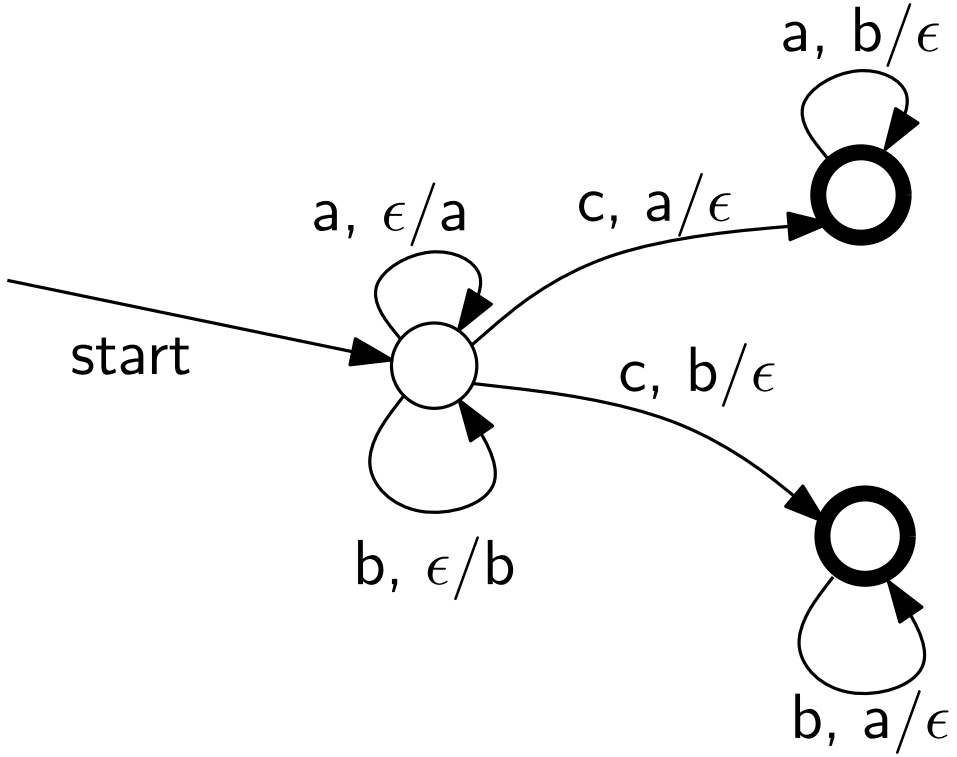
Stackautomat (PDA, Push-Down Automaton):
som DFA, men har ett obegränsat minne i form av en stack



x, y/z: läs x, poppa y från stacken, pusha z

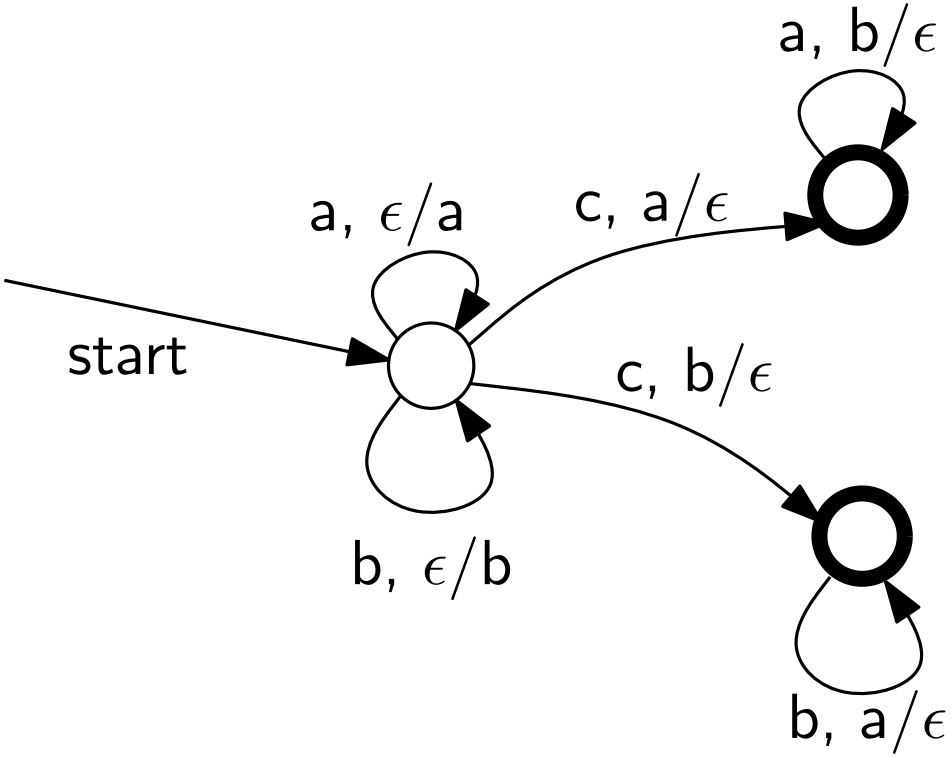
Automaten accepterar om den är i ett accepterande tillstånd **OCH** stacken är tom när indata är slut

Exempelkörning, stackautomat (repetition)



x, y/z: läs x, poppa y från stacken, pusha z

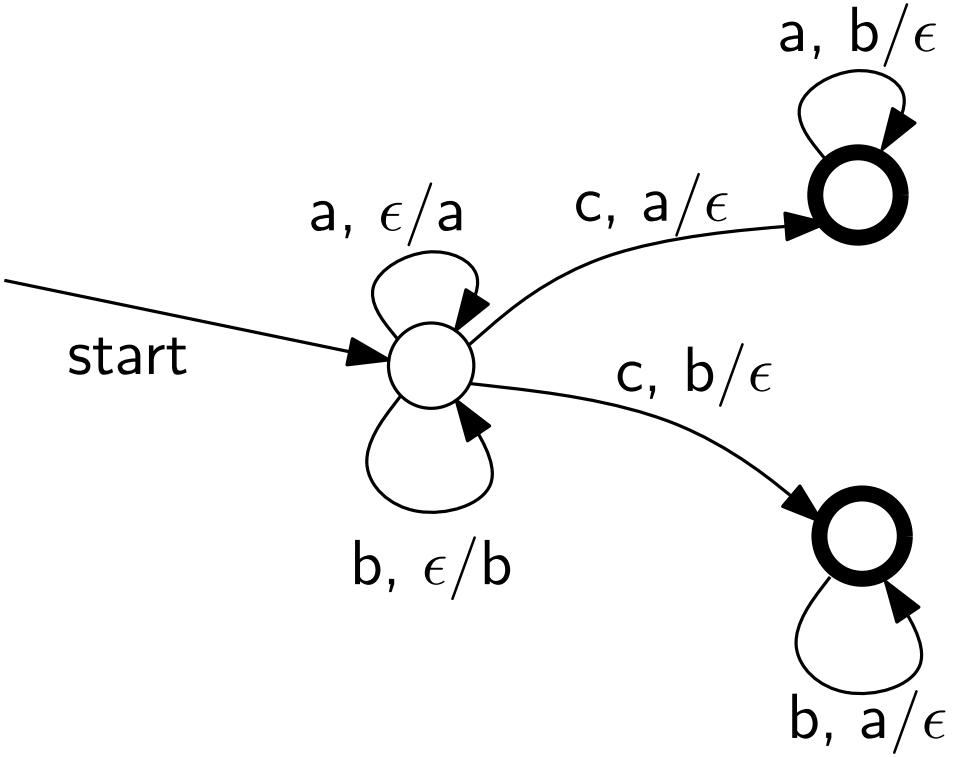
Exempelkörning, stackautomat (repetition)



x, y/z: läs x, poppa y från stacken, pusha z

Indata:
baabcbbb

Exempelkörning, stackautomat (repetition)

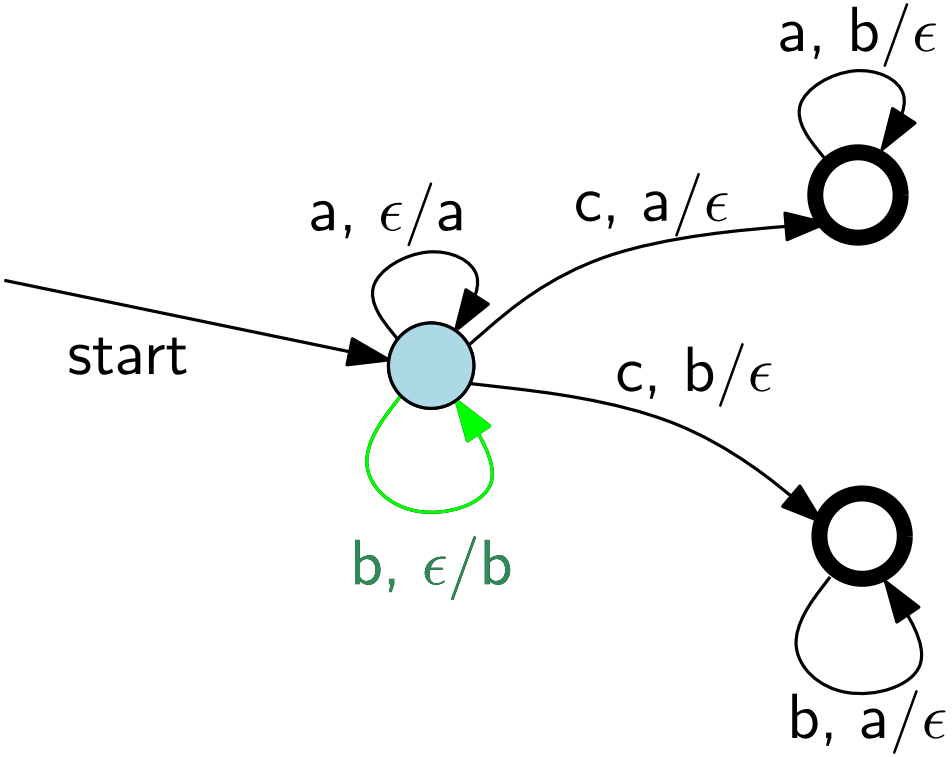


x, y/z: läs x, poppa y från stacken, pusha z

Indata:
baabcbbb

Stacken:

Exempelkörning, stackautomat (repetition)



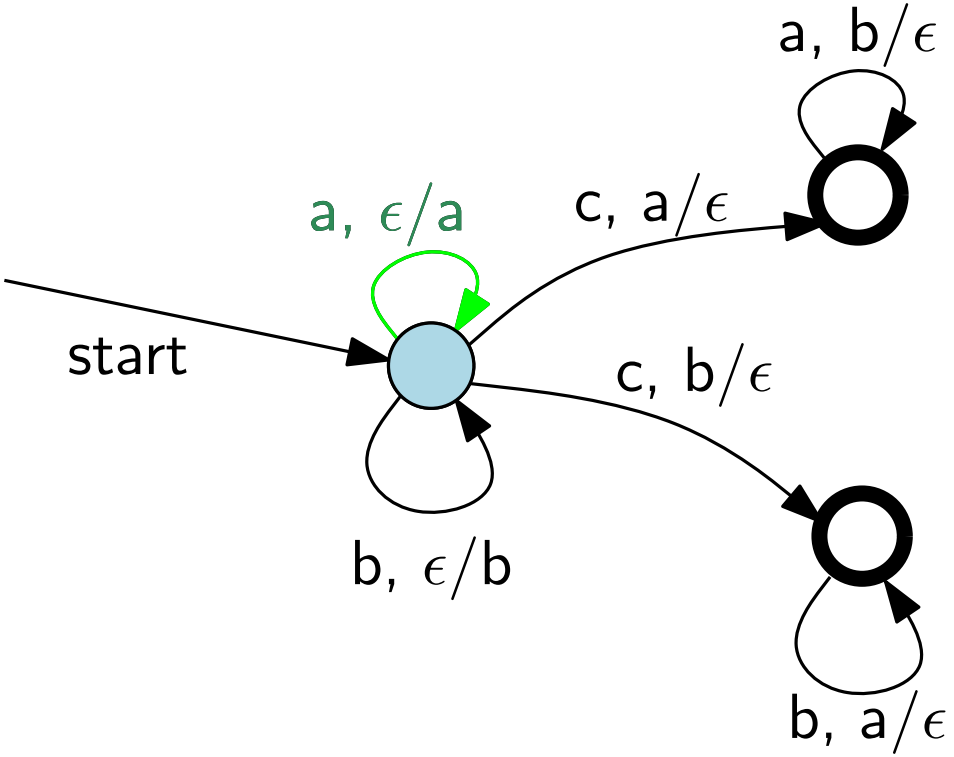
x, y/z: läs x, poppa y från stacken, pusha z

Indata:
baabcbbb

Stacken:

b pushas

Exempelkörning, stackautomat (repetition)



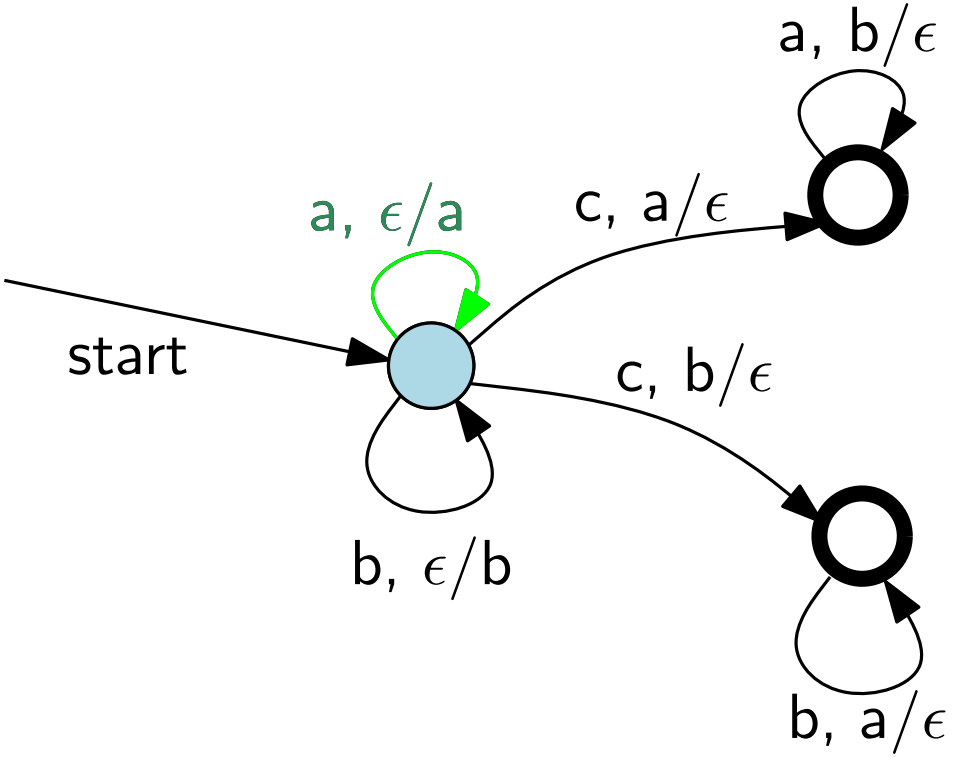
x, y/z: läs x, poppa y från stacken, pusha z

Indata:
baabcbbb

Stacken:

a pushas
b

Exempelkörning, stackautomat (repetition)



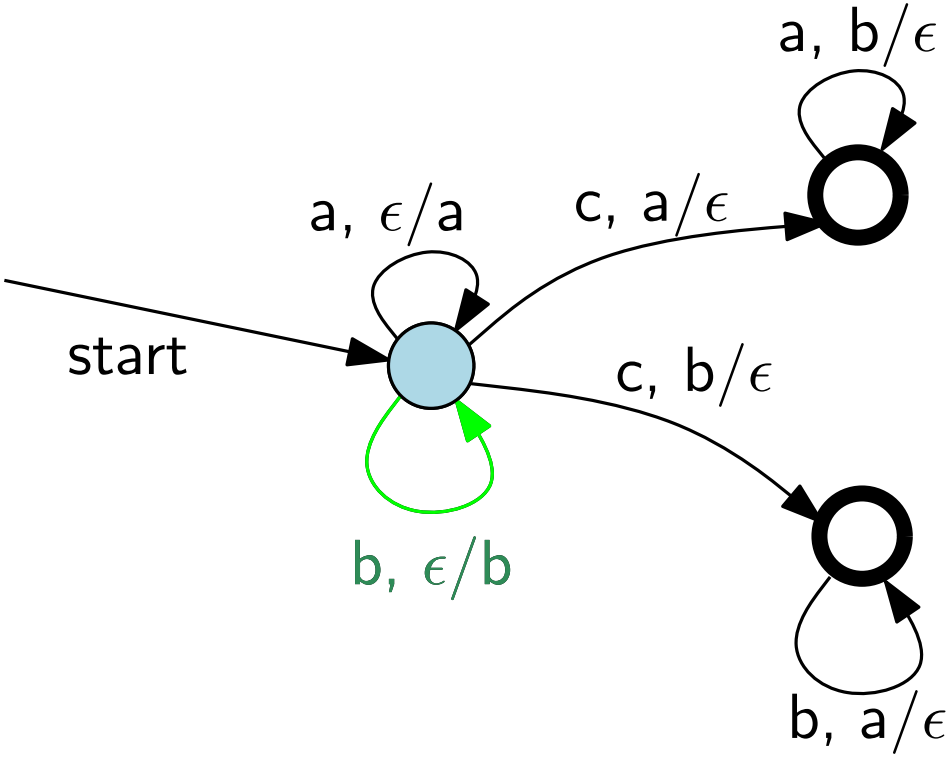
x, y/z: läs x, poppa y från stacken, pusha z

Indata:
baabcbbb

Stacken:

a pushas
a
b

Exempelkörning, stackautomat (repetition)

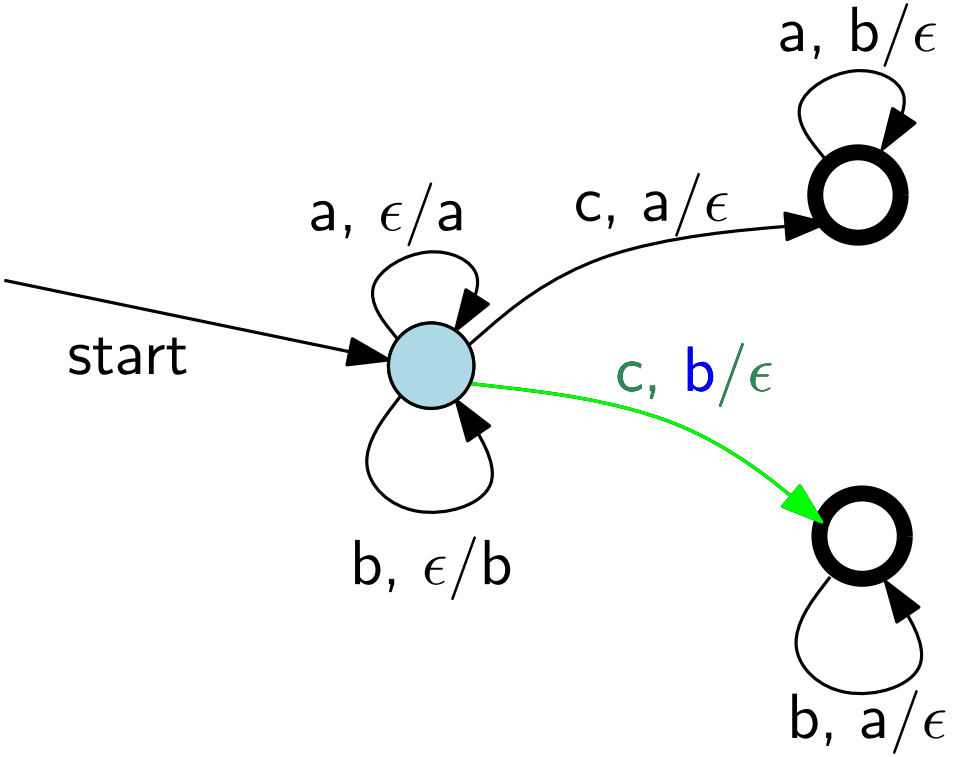


x, y/z: läs x, poppa y från stacken, pusha z

Indata:
baabcbbb

Stacken:
b pushas
a
a
b

Exempelkörning, stackautomat (repetition)

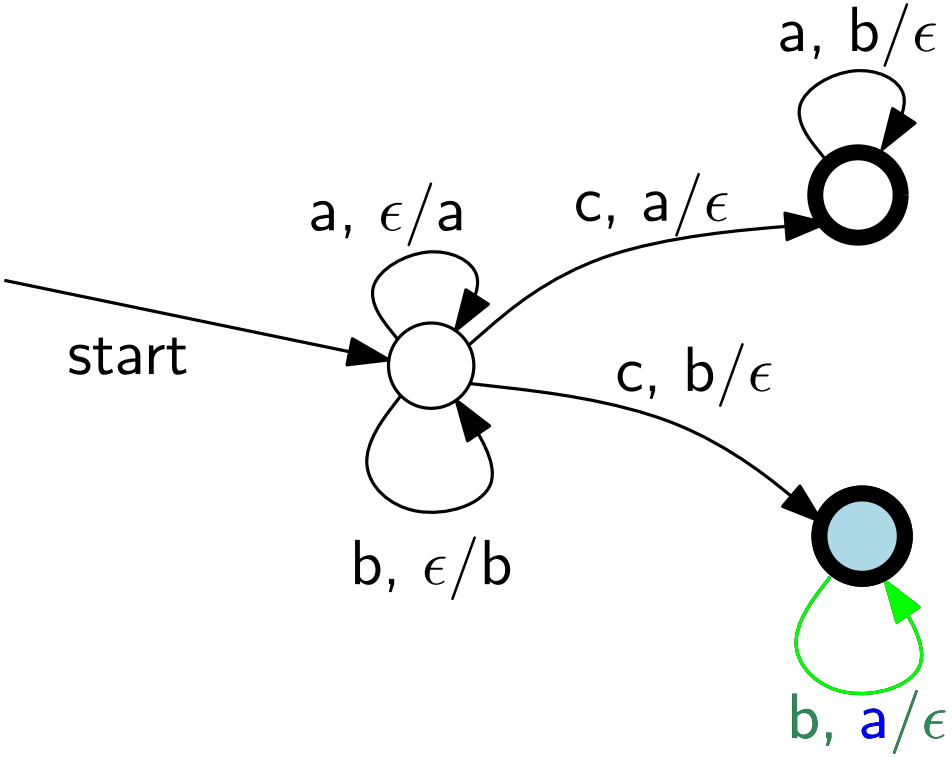


x, y/z: läs x, poppa y från stacken, pusha z

Indata:
baabcbbb

Stacken:
b poppas
a
a
b

Exempelkörning, stackautomat (repetition)



x, y/z: läs x, poppa y från stacken, pusha z

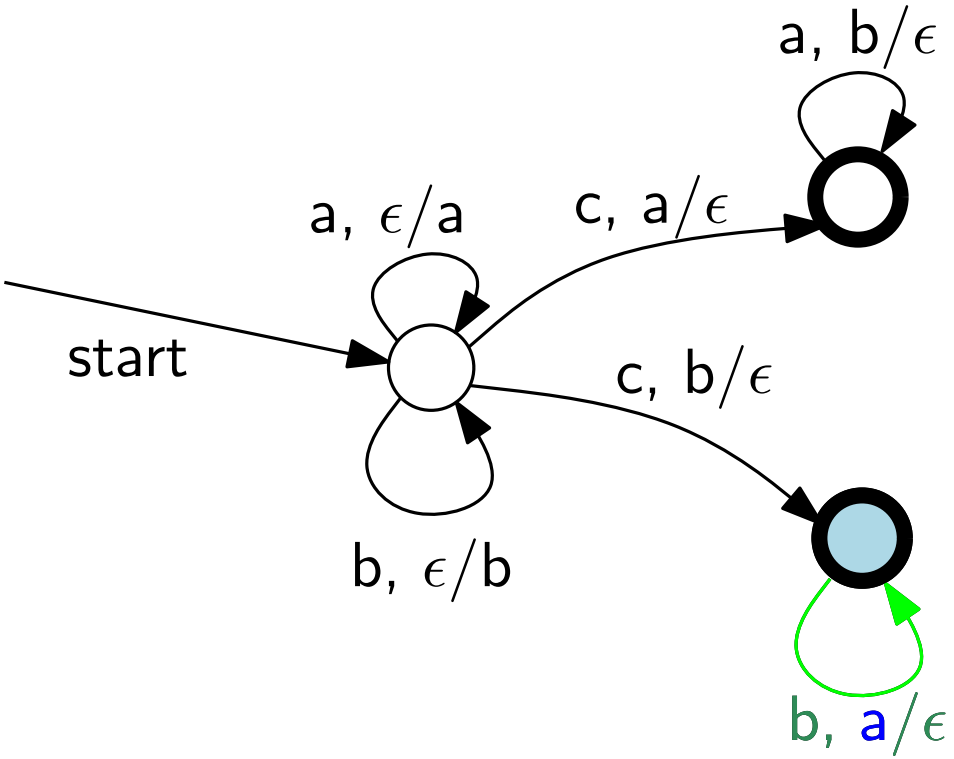
Indata:
baabcbbb

Stacken:

a poppas
a
b

Exempelkörning, stackautomat (repetition)

x, y/z: läs x, poppa y från stacken, pusha z



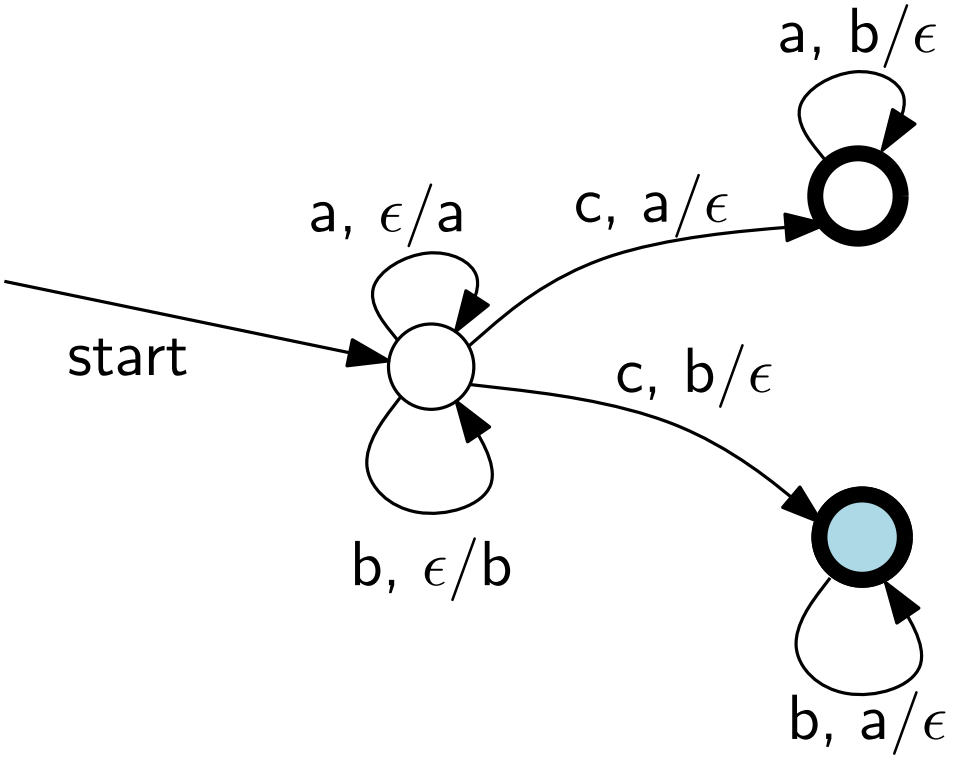
Indata:
baabcbbb

Stacken:

a poppas
b

Exempelkörning, stackautomat (repetition)

x, y/z: läs x, poppa y från stacken, pusha z



Indata:
baabcbbb

Övergång saknas, för tecknet b måste vi ha ett a högst upp på stacken
$\longrightarrow$ automaten accepterar inte

Stacken:

b

Stackautomat för balanserade parentesuttryck

En grammatik för balanserade parentesuttryck:

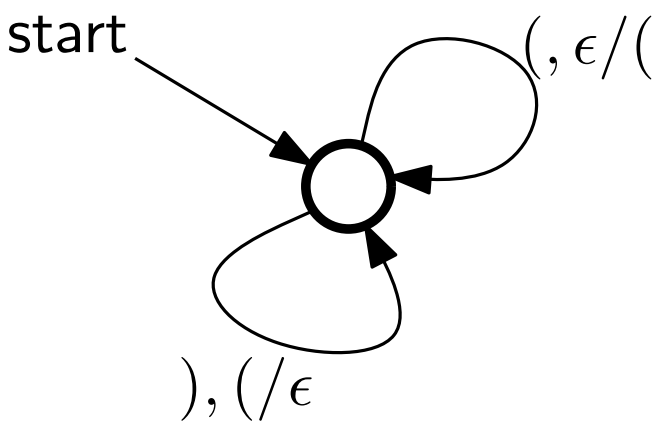
$$\text{Expr} \rightarrow \epsilon | (\text{Expr})\text{Expr}$$

Stackautomat för balanserade parentesuttryck

En grammatik för balanserade parentesuttryck:

$$\text{Expr} \rightarrow \epsilon | (\text{Expr})\text{Expr}$$

Vi kan också lätt konstruera en PDA för språket, vi behöver faktiskt bara ett enda tillstånd! (Plus det implicita fail-tillståndet.)



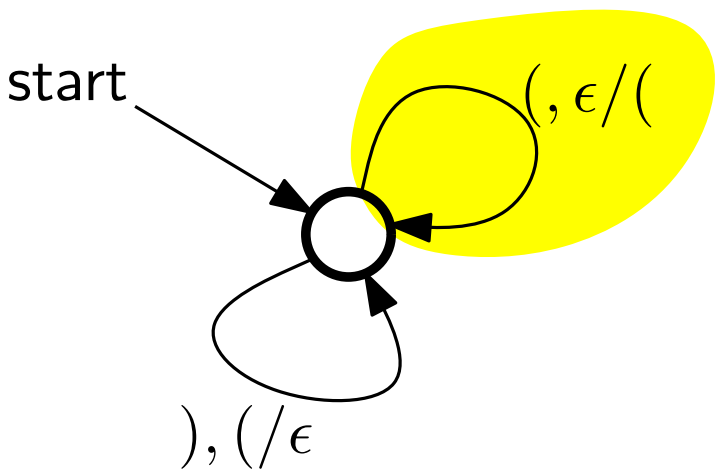
Stackautomat för balanserade parentesuttryck

En grammatik för balanserade parentesuttryck:

$$\text{Expr} \rightarrow \epsilon | (\text{Expr})\text{Expr}$$

Vi kan också lätt konstruera en PDA för språket, vi behöver faktiskt bara ett enda tillstånd!

(Plus det implicita fail-tillståndet.)



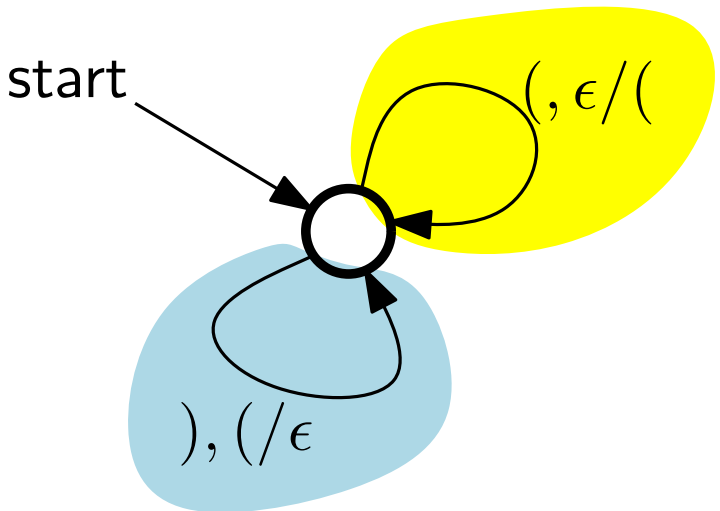
När vi ser en vänsterparentes, pusha på vänsterparentes på stacken

Stackautomat för balanserade parentesuttryck

En grammatik för balanserade parentesuttryck:

$$\text{Expr} \rightarrow \epsilon | (\text{Expr})\text{Expr}$$

Vi kan också lätt konstruera en PDA för språket, vi behöver faktiskt bara ett enda tillstånd! (Plus det implicita fail-tillståndet.)



När vi ser en vänsterparentes, pusha på vänsterparentes på stacken

När vi ser högerparentes, poppa en vänsterparentes från stacken

Från grammatiker till stackautomater

Parentesuttryck var enkelt att bygga stackautomat för.

Från grammatiker till stackautomater

Parentesuttryck var enkelt att bygga stackautomat för.

Kan man alltid konvertera en grammatik till en stackautomat?

Från grammatiker till stackautomater

Parentesuttryck var enkelt att bygga stackautomat för.

Kan man alltid konvertera en grammatik till en stackautomat?

Nej, det kan man inte!

Från grammatiker till stackautomater

Parentesuttryck var enkelt att bygga stackautomat för.

Kan man alltid konvertera en grammatik till en stackautomat?

Nej, det kan man inte!

Exempel: Palindrom över $\{a, b\}$
Palin $\rightarrow \epsilon \,|\, a \,|\, b \,|\, a$ Palin $a \,|\, b$ Palin b

Det existerar inte någon PDA som känner igen detta språk.

Från grammatiker till stackautomater

Parentesuttryck var enkelt att bygga stackautomat för.

Kan man alltid konvertera en grammatik till en stackautomat?

Nej, det kan man inte!

Exempel: Palindrom över $\{a, b\}$
Palin $\rightarrow \epsilon \mid a \mid b \mid a$ Palin $a \mid b$ Palin b

Det existerar inte någon PDA som känner igen detta språk.

Intuition: PDA:n borde, när den kommit exakt halvvägs in i strängen, börja matcha av tecken mot de som den redan sett, men det finns inget sätt för den att veta när den är halvvägs.

Från grammatiker till stackautomater

Parentesuttryck var enkelt att bygga stackautomat för.

Kan man alltid konvertera en grammatik till en stackautomat?

Nej, det kan man inte!

Exempel: Palindrom över $\{a, b\}$
Palin $\rightarrow \epsilon \mid a \mid b \mid a$ Palin $a \mid b$ Palin b

Det existerar inte någon PDA som känner igen detta språk.

Intuition: PDA:n borde, när den kommit exakt halvvägs in i strängen, börja matcha av tecken mot de som den redan sett, men det finns inget sätt för den att veta när den är halvvägs.

Man kan bevisa detta med ett pumping-lemma för PDA:er, liknande beviset vi såg för DFA:er (ingår ej i kursen).

Grammatiker vs. stackautomater

Om stackautomater inte är tillräckligt kraftfulla för att kunna användas till alla grammatiker, vad är poängen?

Grammatiker vs. stackautomater

Om stackautomater inte är tillräckligt kraftfulla för att kunna användas till alla grammatiker, vad är poängen?

Poängen är att de kan användas för “de flesta” grammatiker, t.ex. (nästan?) alltid de man får när man konstruerar programspråk.

Idag

Stackautomater (forts.)

Lexikal analys

Härledningar och syntaxträd

Omskrivning av grammatiker

Rekursiv medåkning och LL(1)-grammatiker

Grammatik för aritmetiska uttryck

Låt oss skriva en grammatik för aritmetiska uttryck med heltal, operatorerna $+, -, *, /$, och parenteser.

Grammatik för aritmetiska uttryck

Låt oss skriva en grammatik för aritmetiska uttryck med heltal, operatorerna $+, -, *, /$, och parenteser.

Möjlig grammatik:

$$
\begin{aligned}
\text{Expr} \rightarrow & \text{Number} \mid \\
& \text{Expr} + \text{Expr} \mid \\
& \text{Expr} - \text{Expr} \mid \\
& \text{Expr} * \text{Expr} \mid \\
& \text{Expr} / \text{Expr} \mid \\
& (\text{Expr}) \\
\text{Number} \rightarrow & \text{Digit} \mid \text{Digit Number} \\
\text{Digit} \rightarrow & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Icke-slutsymboler:
Expr (representerar uttryck)
Number (representerar heltal)
Digit (representerar siffror)

Onödigt komplicerat?

Det väsentliga i den här grammatiken är de rekursiva reglerna för hur man formar ett uttryck.

$$
\begin{aligned}
\text{Expr} \rightarrow\ & \text{Number} \mid \\
& \text{Expr} + \text{Expr} \mid \\
& \text{Expr} - \text{Expr} \mid \\
& \text{Expr} * \text{Expr} \mid \\
& \text{Expr} / \text{Expr} \mid \\
& (\text{Expr}) \\
\text{Number} \rightarrow\ & \text{Digit} \mid \text{Digit Number} \\
\text{Digit} \rightarrow\ & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Onödigt komplicerat?

Det väsentliga i den här grammatiken är de rekursiva reglerna för hur man formar ett uttryck.

Reglerna för vad som är ett tal känns lite som ett bihang som vi var tvungna att ha med för att göra grammatiken fullständig.

$$
\begin{aligned}
\text{Expr} \rightarrow\ & \text{Number} \mid \\
& \text{Expr} + \text{Expr} \mid \\
& \text{Expr} - \text{Expr} \mid \\
& \text{Expr} * \text{Expr} \mid \\
& \text{Expr} / \text{Expr} \mid \\
& (\text{Expr}) \\
\text{Number} \rightarrow\ & \text{Digit} \mid \text{Digit Number} \\
\text{Digit} \rightarrow\ & 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Onödigt komplicerat?

Det väsentliga i den här grammatiken är de rekursiva reglerna för hur man formar ett uttryck.

Reglerna för vad som är ett tal känns lite som ett bihang som vi var tvungna att ha med för att göra grammatiken fullständig.

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Number} \mid \\
&\quad\ \text{Expr} + \text{Expr} \mid \\
&\quad\ \text{Expr} - \text{Expr} \mid \\
&\quad\ \text{Expr} * \text{Expr} \mid \\
&\quad\ \text{Expr} / \text{Expr} \mid \\
&\quad\ (\text{Expr}) \\
\text{Number} &\rightarrow \text{Digit} \mid \text{Digit Number} \\
\text{Digit} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Att känna igen tal är ju väldigt enkelt (kan t.ex. göras med det reguljära uttrycket `[0-9]+`), är lite overkill att använda grammatik för det.

Pre-processa heltal

Idé: *pre-processa* indata-strängen för att ta hand om de delar som utgör ”enkla” beståndsdelar av grammatiken

Pre-processa heltal

Idé: *pre-processa* indata-strängen för att ta hand om de delar som utgör ”enkla” beståndsdelar av grammatiken

Exempel: låt oss titta på strängen “`378*232*(582-01)`”

Pre-processa heltal

Idé: *pre-processa* indata-strängen för att ta hand om de delar som utgör ”enkla” beståndsdelar av grammatiken

Exempel: låt oss titta på strängen “`378*232*(582-01)`”

Detta är *teckensekvensen*
‘3’, ‘7’, ‘8’, ‘*’, ‘2’, ‘3’, ‘2’, ‘*’, ‘(’, ‘5’, ‘8’, ‘2’, ‘-’, ‘0’, ‘1’, ‘)’

Pre-processa heltal

Idé: *pre-processa* indata-strängen för att ta hand om de delar som utgör ”enkla” beståndsdelar av grammatiken

Exempel: låt oss titta på strängen “`378*232*(582-01)`”

Detta är *teckensekvensen*
‘3’, ‘7’, ‘8’, ‘*’, ‘2’, ‘3’, ‘2’, ‘*’, ‘(’, ‘5’, ‘8’, ‘2’, ‘-’, ‘0’, ‘1’, ‘)’

Vi pre-processar detta till den nya sekvensen
Number, ‘*’, Number, ‘*’, ‘(’, Number, ‘-’, Number, ‘)’

Pre-processa heltal

Idé: *pre-processa* indata-strängen för att ta hand om de delar som utgör ”enkla” beståndsdelar av grammatiken

Exempel: låt oss titta på strängen “`378*232*(582-01)`”

Detta är *teckensekvensen*
‘3’, ‘7’, ‘8’, ‘*’, ‘2’, ‘3’, ‘2’, ‘*’, ‘(’, ‘5’, ‘8’, ‘2’, ‘-’, ‘0’, ‘1’, ‘)’

Vi pre-processar detta till den nya sekvensen
Number, ‘*’, Number, ‘*’, ‘(’, Number, ‘-’, Number, ‘)’

Elementen i den nya sekvensen kallas för *tokens*.

Pre-processa heltal

Idé: *pre-processa* indata-strängen för att ta hand om de delar som utgör ”enkla” beståndsdelar av grammatiken

Exempel: låt oss titta på strängen “`378*232*(582-01)`”

Detta är *teckensekvensen*
‘3’, ‘7’, ‘8’, ‘*’, ‘2’, ‘3’, ‘2’, ‘*’, ‘(’, ‘5’, ‘8’, ‘2’, ‘-’, ‘0’, ‘1’, ‘)’

Vi pre-processar detta till den nya sekvensen
Number, ‘*’, Number, ‘*’, ‘(’, Number, ‘-’, Number, ‘)’

Elementen i den nya sekvensen kallas för *tokens*.

Några tokens är helt enkelt enstaka tecken från indata-strängen, men andra består av delsträngar från indata-strängen som getts en typ (i det här fallet “Number”)

Pre-processa heltal

Idé: *pre-processa* indata-strängen för att ta hand om de delar som utgör ”enkla” beståndsdelar av grammatiken

Exempel: låt oss titta på strängen “`378*232*(582-01)`”

Detta är *teckensekvensen*
‘3’, ‘7’, ‘8’, ‘*’, ‘2’, ‘3’, ‘2’, ‘*’, ‘(’, ‘5’, ‘8’, ‘2’, ‘-’, ‘0’, ‘1’, ‘)’

Vi pre-processar detta till den nya sekvensen
Number, ‘*’, Number, ‘*’, ‘(’, Number, ‘-’, Number, ‘)’

Elementen i den nya sekvensen kallas för *tokens*.

Några tokens är helt enkelt enstaka tecken från indata-strängen, men andra består av delsträngar från indata-strängen som getts en typ (i det här fallet “Number”)

Vi betraktar denna *tokensekvens* som ett indata att parsa

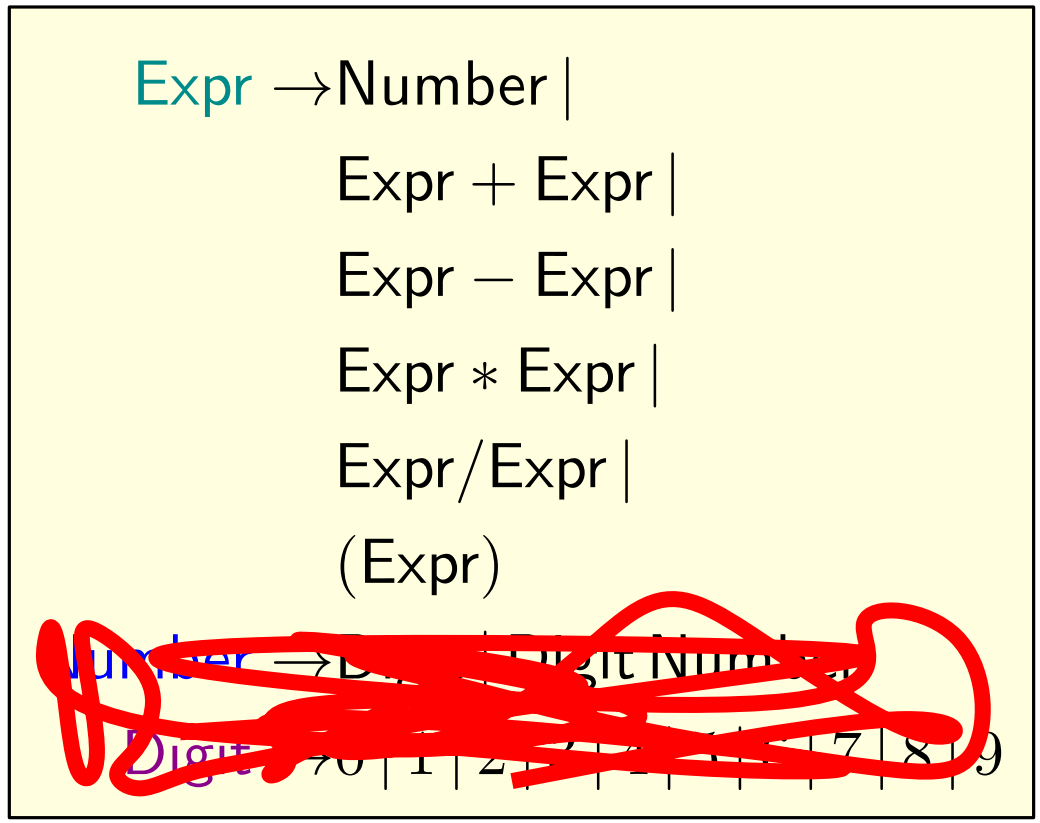
Åter till grammatiken

Vad händer med vår grammatik?

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Number} \mid \\
&\quad \text{Expr} + \text{Expr} \mid \\
&\quad \text{Expr} - \text{Expr} \mid \\
&\quad \text{Expr} * \text{Expr} \mid \\
&\quad \text{Expr} / \text{Expr} \mid \\
&\quad (\text{Expr}) \\
\text{Number} &\rightarrow \text{Digit} \mid \text{Digit Number} \\
\text{Digit} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

Åter till grammatiken

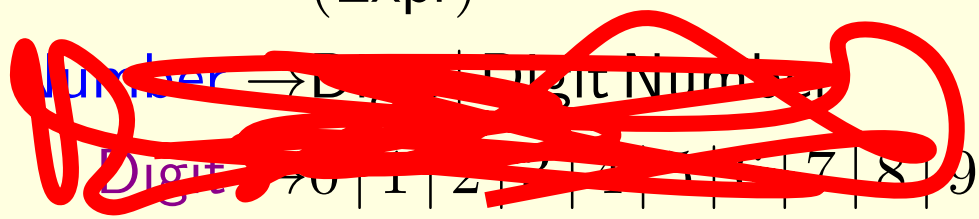
Vad händer med vår grammatik?



Åter till grammatiken

Vad händer med vår grammatik?

$$
\begin{aligned}
\text{Expr} \rightarrow & \text{Number} \mid \\
& \text{Expr} + \text{Expr} \mid \\
& \text{Expr} - \text{Expr} \mid \\
& \text{Expr} * \text{Expr} \mid \\
& \text{Expr} / \text{Expr} \mid \\
& (\text{Expr})
\end{aligned}
$$



Man kan säga att vi har "uppgraderat" (eller nedgraderat?) "Number" till en slut-symbol istället för en icke-slutsymbol

Lexikal analys

Lexikal analys är processen att transformera en indatasträng (sekvens av tecken) till en sekvens av tokens.

Syften:

1. Abstrahera bort smådetaljer ur grammatiken.
2. Städa bort irrelevanta delar av indata
   - Kommentarer i programmeringsspråk
   - Whitespace
     Vi vill antagligen att “12+5” och “12 + 5” ska behandlas likadant, men att “1 2+5” ska behandlas annorlunda

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
       -- beräkna differensen
       x = alpha-beta
```

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Vad är lämpliga val av tokens i Haskell?

Förslag:
Where/Let/Derives/etc alla keywords i språket
Equal/Plus/Minus/Times/etc operatorer
Int/Double/etc tal av olika typer
Name Variabel/funktionsnamn
(och många fler)

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Detta är en kommentar så vi ignorerar den

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name, Where

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name, Where

Detta är en kommentar så vi ignorerar den

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name, Where, Name

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name, Where, Name, Equal

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name, Where, Name, Equal, Name

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name, Where, Name, Equal, Name, Minus

Lexikal analys, exempel

Betrakta följande Haskell-funktion

```
-- Det här är min funktion
minFunktion alfa beta =
    5*x
    where
        -- beräkna differensen
        x = alpha-beta
```

Möjlig tokenisering/lexikal analys:

Name, Name, Name, Equal, Int, Times, Name, Where, Name, Equal, Name, Minus, Name

Idag

Stackautomater (forts.)

Lexikal analys

Härledningar och syntaxträd

Omskrivning av grammatiker

Rekursiv medåkning och LL(1)-grammatiker

Språk – inte bara syntax

Hittills har vi *bara* pratat om *syntax*: givet en sträng, ligger den i språket, ja eller nej?

Språk – inte bara syntax

Hittills har vi *bara* pratat om *syntax*: givet en sträng, ligger den i språket, ja eller nej?

Exempel

- Är `4*(5+3)` ett syntaktiskt korrekt aritmetiskt uttryck?
- Är min java-fil ett giltigt java-program?

Språk – inte bara syntax

Hittills har vi *bara* pratat om *syntax*: givet en sträng, ligger den i språket, ja eller nej?

Exempel

- Är `4*(5+3)` ett syntaktiskt korrekt aritmetiskt uttryck?
- Är min java-fil ett giltigt java-program?

Men ofta har ju strängarna i språken en mening – en *semantik* – som man vill kunna analysera

Språk – inte bara syntax

Hittills har vi *bara* pratat om *syntax*: givet en sträng, ligger den i språket, ja eller nej?

Exempel

- Är `4*(5+3)` ett syntaktiskt korrekt aritmetiskt uttryck?
- Är min java-fil ett giltigt java-program?

Men ofta har ju strängarna i språken en mening – en *semantik* – som man vill kunna analysera

Exempel

- Vad är värdet av `4*(5+3)`?
- Vad händer när man kör mitt java-program?

Härledning

En *härledning* av en indatasträng är en sekvens av grammatikregler som producerar strängen

$$
\begin{aligned}
\text{Expr} \rightarrow\ & \text{Num} \mid \\
& \text{Expr} + \text{Expr} \mid \\
& \text{Expr} - \text{Expr} \mid \\
& \text{Expr} * \text{Expr} \mid \\
& \text{Expr} / \text{Expr} \mid \\
& (\text{Expr})
\end{aligned}
$$

Härledning

En *härledning* av en indatasträng är en sekvens av grammatikregler som producerar strängen

Exempel: Låt oss härleda "4*(5+3)", dvs token-sekvensen

Num, '*', '(', Num, '+', Num, ')'

$$
\begin{aligned}
\text{Expr} \rightarrow & \text{Num} \mid \\
& \text{Expr} + \text{Expr} \mid \\
& \text{Expr} - \text{Expr} \mid \\
& \text{Expr} * \text{Expr} \mid \\
& \text{Expr} / \text{Expr} \mid \\
& (\text{Expr})
\end{aligned}
$$

Härledning

En *härledning* av en indatasträng är en sekvens av grammatikregler som producerar strängen

Exempel: Låt oss härleda "4*(5+3)", dvs token-sekvensen

Num, '*', '(', Num, '+', Num, ')'

$$
\begin{aligned}
\text{Expr} \rightarrow & \text{Num} \mid \\
& \text{Expr} + \text{Expr} \mid \\
& \text{Expr} - \text{Expr} \mid \\
& \text{Expr} * \text{Expr} \mid \\
& \text{Expr} / \text{Expr} \mid \\
& (\text{Expr})
\end{aligned}
$$

Härledning:

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Expr} * \text{Expr} \\
&\rightarrow \text{Num} * \text{Expr} \\
&\rightarrow \text{Num} * (\text{Expr}) \\
&\rightarrow \text{Num} * (\text{Expr} + \text{Expr}) \\
&\rightarrow \text{Num} * (\text{Num} + \text{Expr}) \\
&\rightarrow \text{Num} * (\text{Num} + \text{Num})
\end{aligned}
$$

Härledning

En *härledning* av en indatasträng är en sekvens av grammatikregler som producerar strängen

Exempel: Låt oss härleda "4*(5+3)", dvs token-sekvensen

Num, '*', '(', Num, '+', Num, ')'

$$
\begin{aligned}
\text{Expr} \rightarrow & \text{Num} \mid \\
& \text{Expr} + \text{Expr} \mid \\
& \text{Expr} - \text{Expr} \mid \\
& \text{Expr} * \text{Expr} \mid \\
& \text{Expr} / \text{Expr} \mid \\
& (\text{Expr})
\end{aligned}
$$

Härledning:

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Expr} * \text{Expr} \\
&\rightarrow \text{Num} * \text{Expr} \\
&\rightarrow \text{Num} * (\text{Expr}) \\
&\rightarrow \text{Num} * (\text{Expr} + \text{Expr}) \\
&\rightarrow \text{Num} * (\text{Num} + \text{Expr}) \\
&\rightarrow \text{Num} * (\text{Num} + \text{Num})
\end{aligned}
$$

Varje steg använder en av produktionsreglerna, och vi landar till slut i indata-sekvensen

Syntaxträd

Det bästa sättet att representera härledningar på är med *syntaxträd*

Syntaxträd

Det bästa sättet att representera härledningar på är med *syntaxträd*

Härledning

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Expr} * \text{Expr} \\
&\rightarrow \text{Num} * \text{Expr} \\
&\rightarrow \text{Num} * (\text{Expr}) \\
&\rightarrow \text{Num} * (\text{Expr} + \text{Expr}) \\
&\rightarrow \text{Num} * (\text{Num} + \text{Expr}) \\
&\rightarrow \text{Num} * (\text{Num} + \text{Num})
\end{aligned}
$$

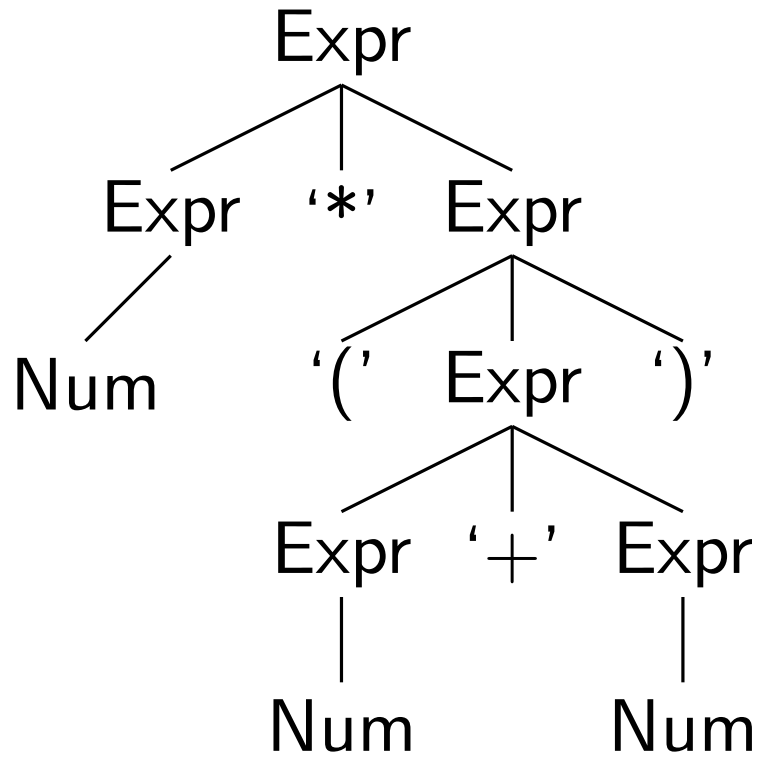
Syntaxträd

Det bästa sättet att representera härledningar på är med *syntaxträd*

Härledning

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Expr} * \text{Expr} \\
&\rightarrow \text{Num} * \text{Expr} \\
&\rightarrow \text{Num} * (\text{Expr}) \\
&\rightarrow \text{Num} * (\text{Expr}+\text{Expr}) \\
&\rightarrow \text{Num} * (\text{Num}+\text{Expr}) \\
&\rightarrow \text{Num} * (\text{Num}+\text{Num})
\end{aligned}
$$

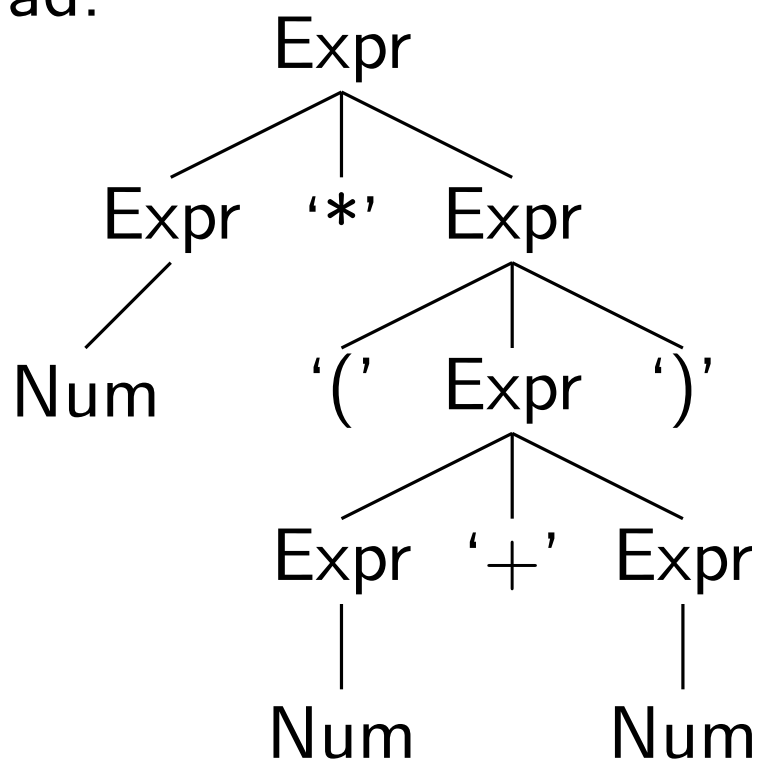
Visualiserat som syntaxträd



Semantik från härledningar/syntaxträd

Uttryck: `4*(5+3)`

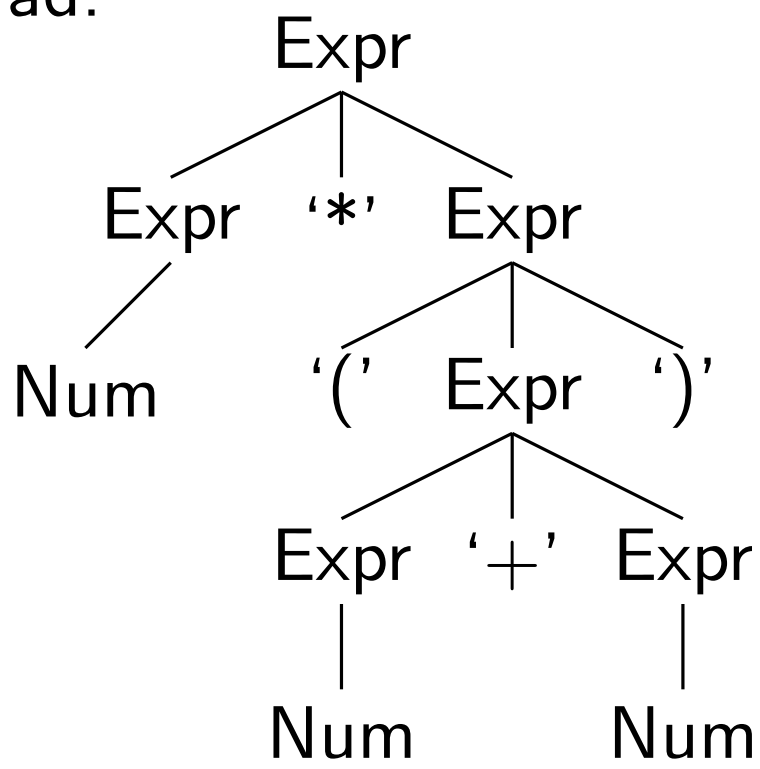
Syntaxträd:



Semantik från härledningar/syntaxträd

Uttryck: `4*(5+3)`

Syntaxträd:

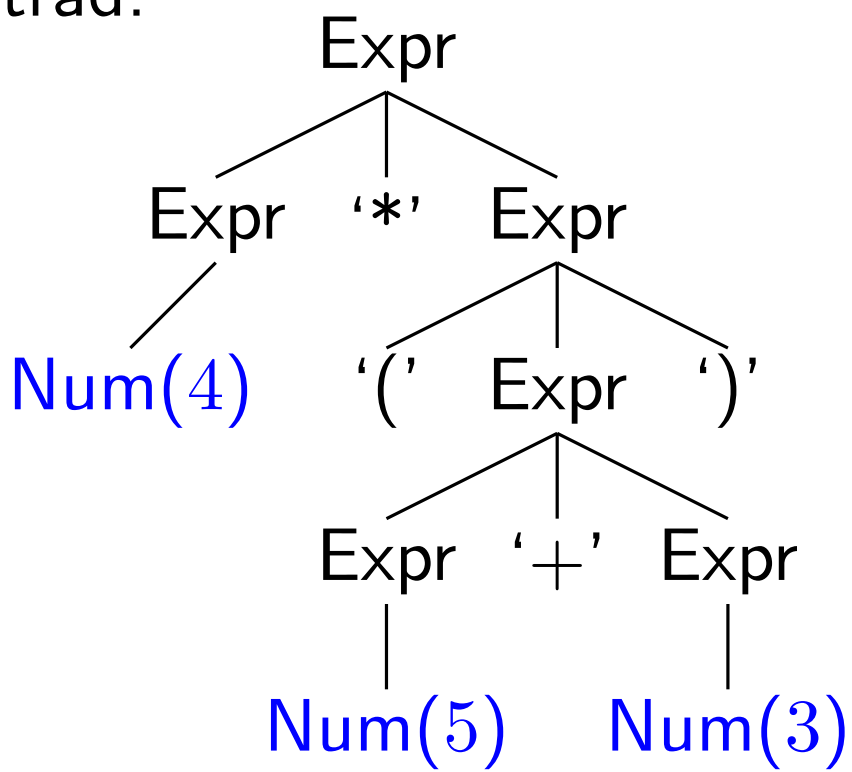


För att kunna beräkna
värdet av uttrycket behöver
vi ju veta exakt vilka tal som
de olika Num-elementen var.

Semantik från härledningar/syntaxträd

Uttryck: `4*(5+3)`

Syntaxträd:



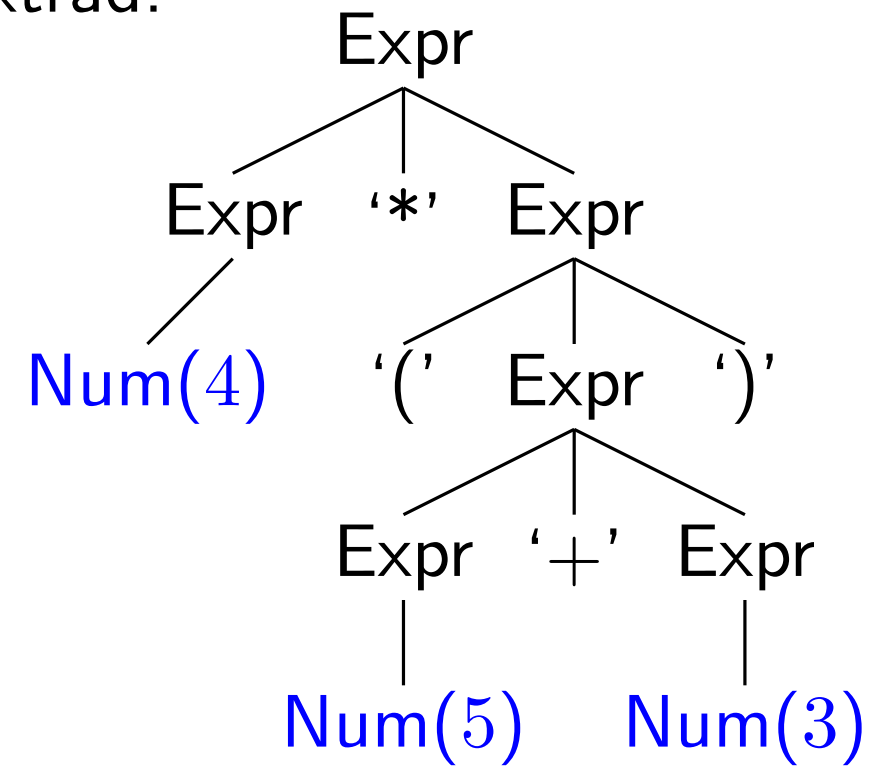
För att kunna beräkna
värdet av uttrycket behöver
vi ju veta exakt vilka tal som
de olika Num-elementen var.

Varje Num-token “taggas”
med semantisk data som talar
om vilket tal det representerar

Semantik från härledningar/syntaxträd

Uttryck: `4*(5+3)`

Syntaxträd:



För att kunna beräkna
värdet av uttrycket behöver
vi ju veta exakt vilka tal som
de olika Num-elementen var.

Varje Num-token “taggas”
med semantisk data som talar
om vilket tal det representerar

Man kan nu beräkna värdet med en enkel sökning i trädet.

```
if leafnode: value = token.value()
if plusnode: value = left.value() + right.value()
if mulnode: value = left.value() * right.value()
etc...
```

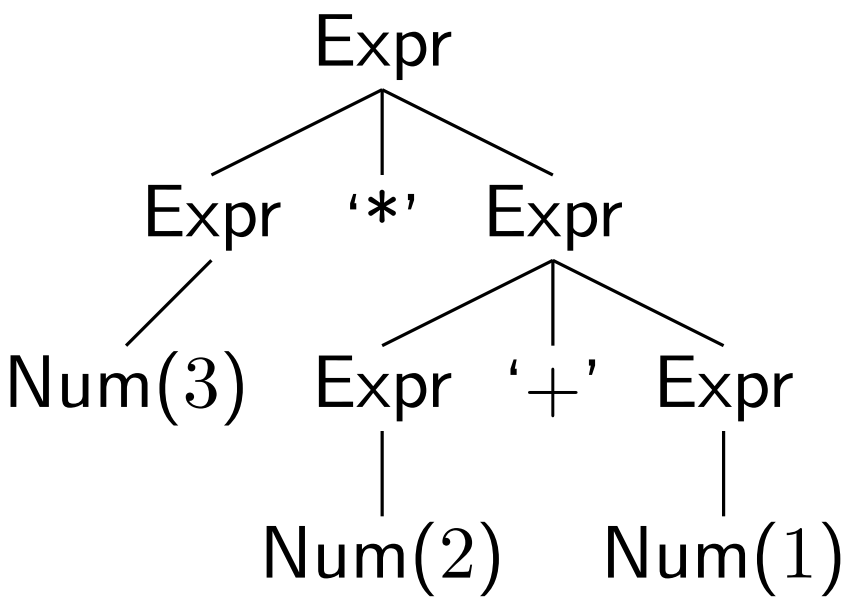
Syntaxträd, exempel 2

Uttryck: `3*2+1`

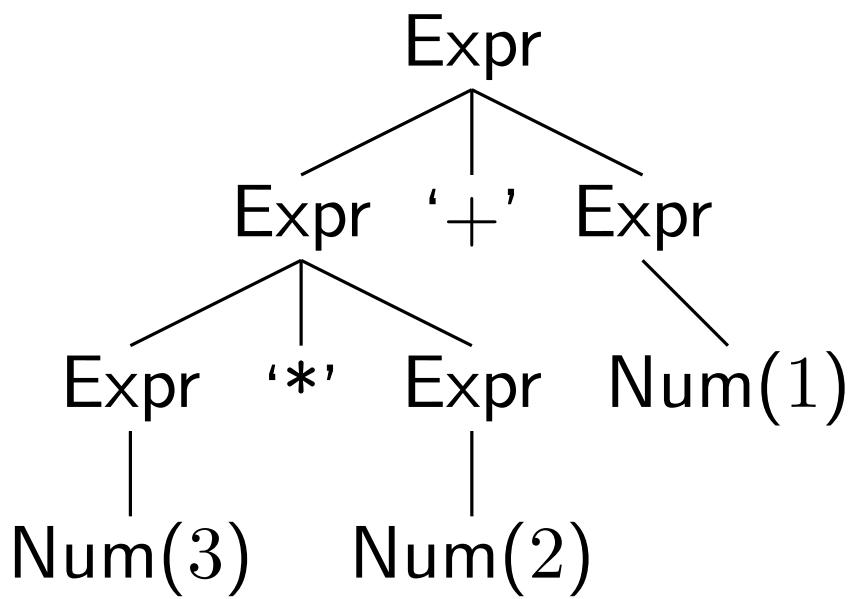
Syntaxträd, exempel 2

Uttryck: 3*2+1

Anna hävdar att det har följande syntaxträd:



Bengt hävdar att det har följande syntaxträd:

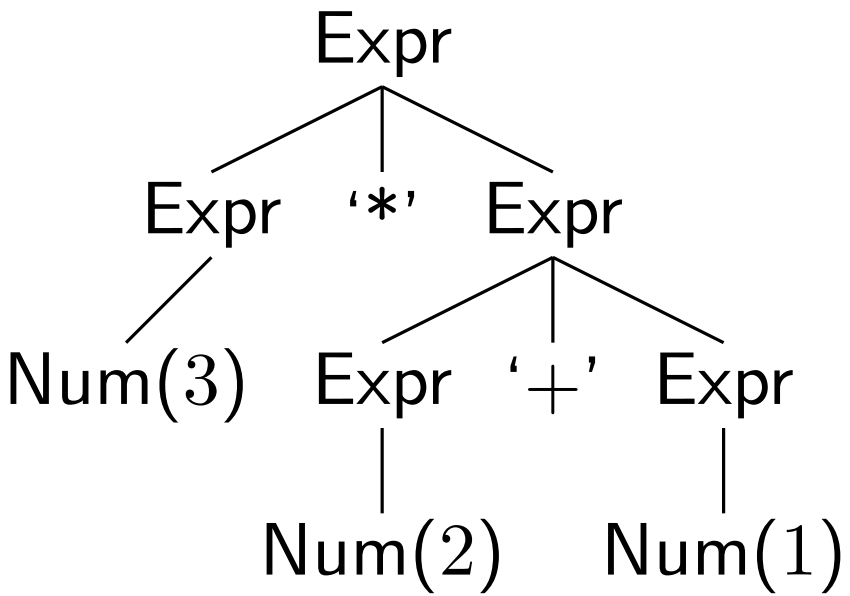


Vem har rätt?

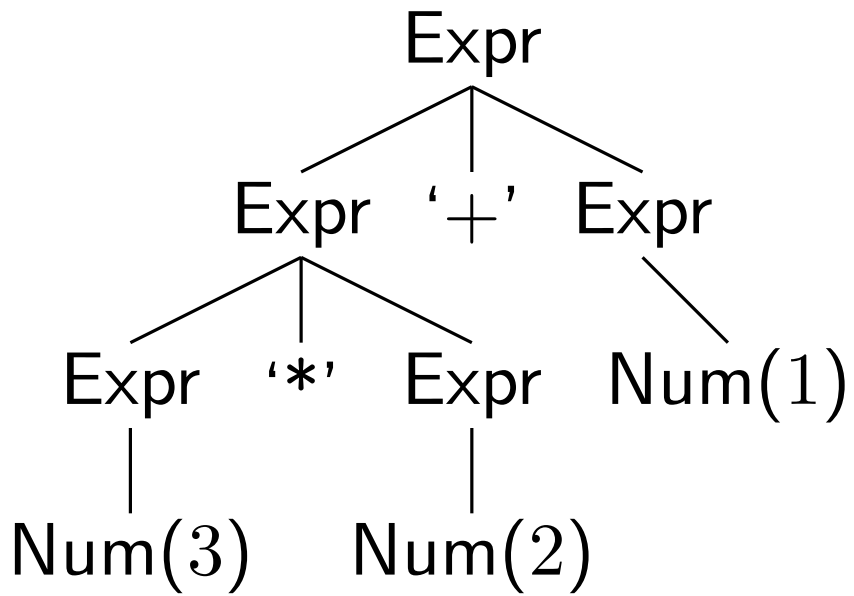
Syntaxträd, exempel 2

Uttryck: 3*2+1

Anna hävdar att det har följande syntaxträd:



Bengt hävdar att det har följande syntaxträd:



Vem har rätt?

Båda är korrekta syntaxträd för uttrycket enligt grammatiken!

Tvetydighet

Båda är korrekta syntaxträd för uttrycket enligt grammatiken

Tvetydighet

Båda är korrekta syntaxträd för uttrycket enligt grammatiken

Grammatiken är *tvetydig* – det finns flera olika syntaxträd för samma sträng

Tvetydighet

Båda är korrekta syntaxträd för uttrycket enligt grammatiken

Grammatiken är *tvetydig* – det finns flera olika syntaxträd för samma sträng

Om vi utvärderar uttrycket med de olika träden får vi olika svar: $3 \cdot (2 + 1) = 9$ vs. $(3 \cdot 2) + 1 = 7$

Tvetydighet

Båda är korrekta syntaxträd för uttrycket enligt grammatiken

Grammatiken är *tvetydig* – det finns flera olika syntaxträd för samma sträng

Om vi utvärderar uttrycket med de olika träden får vi olika svar: $3 \cdot (2 + 1) = 9$ vs. $(3 \cdot 2) + 1 = 7$

Att en grammatik är tvetydig spelar ingen roll för syntaxen (ändrar inte vad som är giltiga strängar), men gör att det finns flera olika semantiska tolkningar av strängarna i språket.

Tvetydighet

Båda är korrekta syntaxträd för uttrycket enligt grammatiken

Grammatiken är *tvetydig* – det finns flera olika syntaxträd för samma sträng

Om vi utvärderar uttrycket med de olika träden får vi olika svar: $3 \cdot (2 + 1) = 9$ vs. $(3 \cdot 2) + 1 = 7$

Att en grammatik är tvetydig spelar ingen roll för syntaxen (ändrar inte vad som är giltiga strängar), men gör att det finns flera olika semantiska tolkningar av strängarna i språket.

Sådan dubbeltydighet kan vara charmig i naturliga språk, men antagligen inget man vill ha i sitt programmeringsspråk...

Idag

Stackautomater (forts.)

Lexikal analys

Härledningar och syntaxträd

Omskrivning av grammatiker

Rekursiv medåkning och LL(1)-grammatiker

Varför blev det tvetydigt?

Exempel:

```
3*2+1
5-2*3/2+3*(5-3)
```

Varför blev det tvetydigt?

Exempel:

```
3*2+1
5-2*3/2+3*(5-3)
```

*Vi vill att * och / ska ha högre prioritet än + och -*

Varför blev det tvetydigt?

Exempel:

```
3*2+1
5-2*3/2+3*(5-3)
```

*Vi vill att * och / ska ha högre prioritet än + och -*

Men detta finns inte med i grammatiken

Varför blev det tvetydigt?

Exempel:

```
3*2+1
5-2*3/2+3*(5-3)
```

*Vi vill att * och / ska ha högre prioritet än + och -*

Men detta finns inte med i grammatiken

Ett mer detaljerat resonemang om hur aritmetiska uttryck ser ut:

- ett uttryck består av en eller flera termer (separerade av $+$ eller $-$)
- varje term består av en eller flera faktorer (separerade av $*$ eller $/$)
- varje faktor är antingen ett tal, eller ett uttryck omgivet av parenteser

Grammatik för aritmetiska uttryck, version 3

Ny, bättre, grammatik för aritmetiska uttryck:

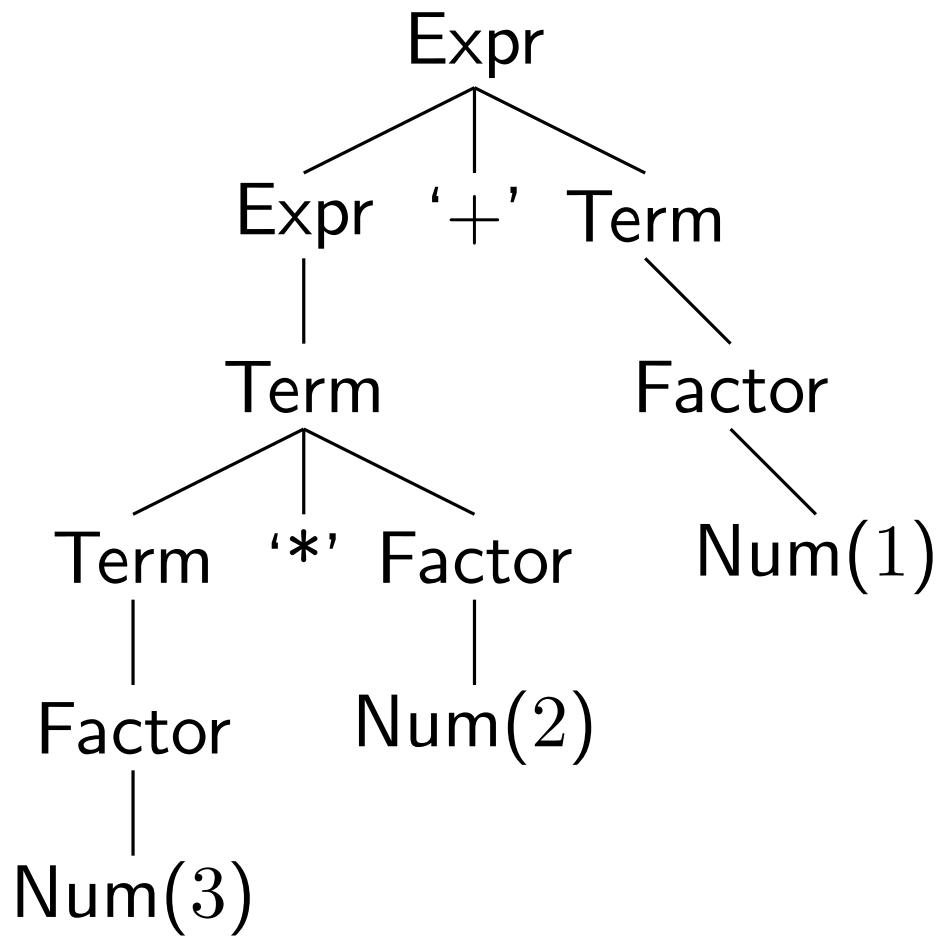
$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Term} \mid \\
&\quad\ \text{Expr} + \text{Term} \mid \\
&\quad\ \text{Expr} - \text{Term} \\
\text{Term} &\rightarrow \text{Factor} \mid \\
&\quad\ \text{Term} * \text{Factor} \mid \\
&\quad\ \text{Term} / \text{Factor} \\
\text{Factor} &\rightarrow \text{Num} \mid (\text{Expr})
\end{aligned}
$$

Grammatik för aritmetiska uttryck, version 3

Ny, bättre, grammatik för aritmetiska uttryck:

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Term} \mid \\
&\quad \text{Expr} + \text{Term} \mid \\
&\quad \text{Expr} - \text{Term} \\
\text{Term} &\rightarrow \text{Factor} \mid \\
&\quad \text{Term} * \text{Factor} \mid \\
&\quad \text{Term} / \text{Factor} \\
\text{Factor} &\rightarrow \text{Num} \mid (\text{Expr})
\end{aligned}
$$

Syntaxträd för 3*2+1:



Eliminera tvetydighet

Tyvärr är det i allmänhet väldigt svårt att se på en grammatik om den är tvetydig eller inte

Eliminera tvetydighet

Tyvärr är det i allmänhet väldigt svårt att se på en grammatik om den är tvetydig eller inte

Har man tur (eller otur...) och hittar två olika syntax-träd för samma sträng så vet man att grammatiken är tvetydig

Eliminera tvetydighet

Tyvärr är det i allmänhet väldigt svårt att se på en grammatik om den är tvetydig eller inte

Har man tur (eller otur...) och hittar två olika syntax-träd för samma sträng så vet man att grammatiken är tvetydig

Men det finns inget enkelt sätt att övertyga sig om att en grammatik är otvetydig

(Att avgöra om en grammatik är tvetydig är ett så kallat *oavgörbart* problem – det existerar ingen algoritm för att göra detta.)

Idag

Lexikal analys

Härledningar och syntaxträd

Omskrivning av grammatiker

Rekursiv medåkning och LL(1)-grammatiker

Grammatik för binära träd

BinTree $\rightarrow$ Leaf LParen Number RParen |
Branch LParen BinTree Comma BinTree RParen

Grammatik för binära träd

BinTree →Leaf LParen Number RParen |
Branch LParen BinTree Comma BinTree RParen

Slutsymboler:
Leaf: strängen "`leaf`"
Branch: strängen "`branch`"
Number: `[0-9]+`
LParen, RParen, Comma: parenteser och kommatecken

Grammatik för binära träd

BinTree →Leaf LParen Number RParen |
Branch LParen BinTree Comma BinTree RParen

Slutsymboler:
Leaf: strängen "`leaf`"
Branch: strängen "`branch`"
Number: `[0-9]+`
LParen, RParen, Comma: parenteser och kommatecken

Exempel:
`branch(branch(leaf(17),leaf(42)),leaf(5))`

Grammatik för binära träd

BinTree →Leaf LParen Number RParen |
Branch LParen BinTree Comma BinTree RParen

Slutsymboler:
Leaf: strängen "`leaf`"
Branch: strängen "`branch`"
Number: `[0-9]+`
LParen, RParen, Comma: parenteser och kommatecken

Exempel:
`branch(branch(leaf(17),leaf(42)),leaf(5))`

Efter lexikal analys:
Branch, LParen, Branch, LParen, Leaf, LParen, Number, RParen, Comma, Leaf, LParen, Number, RParen, RParen, Comma, Leaf, LParen, Number, RParen, RParen

Rekursiv medåkning

Rekursiv medåkning är en “meta-algoritm” för att konstruera en parser för en grammatik.

Rekursiv medåkning

Rekursiv medåkning är en "meta-algoritm" för att konstruera en parser för en grammatik.

Huvudidé:

- en funktion per icke-slutsymbol, som är ansvarig för att parsa den icke-slutsymbolen.
- tittar på nästa token i indata och väljer produktionsregel baserat på det
- sedan rekursiva anrop för att parsa de olika delarna av högerledet

Rekursiv medåkning för binära träd

$$\mathsf{BinTree} \rightarrow \mathsf{Leaf}(\mathsf{Number}) \mid \mathsf{Branch}(\mathsf{BinTree}, \mathsf{BinTree})$$

Rekursiv medåkning för binära träd

$$\text{BinTree} \rightarrow \text{Leaf}(\text{Number}) \mid \text{Branch}(\text{BinTree}, \text{BinTree})$$

Bara en icke-slutsymbol, vi ska ha en funktion `BinTree` som är ansvarig för att parsa BinTree

Rekursiv medåkning för binära träd

$$\text{BinTree} \rightarrow \text{Leaf}(\text{Number}) \mid \text{Branch}(\text{BinTree}, \text{BinTree})$$

Bara en icke-slutsymbol, vi ska ha en funktion `BinTree` som är ansvarig för att parsa BinTree

```
ParseTree BinTree() {
   Token t = NextToken();
   if (t.type == Leaf) {
      if (NextToken().type != LParen) throw SyntaxError();
      Token Num = NextToken();
      if (Num.type != Number) throw SyntaxError();
      if (NextToken().type != RParen) throw SyntaxError();
      return new LeafNode(Num.data);
   } else if (t.type == Branch) {
      if (NextToken().type != LParen) throw SyntaxError();
      ParseTree left = BinTree();
      if (NextToken().type != Comma) throw SyntaxError();
      ParseTree right = BinTree();
      if (NextToken().type != RParen) throw SyntaxError();
      return new BranchNode(left, right);
   }
}
```

Rekursiv medåkning för binära träd

Bin ree)

Bara en ee som är ansvarig

Fullständig Java-implementation av lexikal analys och rekursiv medåknings-parser för binära träd finns på kurshemsidan under ”kursmaterial” för dagens föreläsning

```
ParseTree B
   Token t = NextToken();
   if (t.type == Leaf) {
      if (NextToken().type != LParen) throw SyntaxError();
      Token Num = NextToken();
      if (Num.type != Number) throw SyntaxError();
      if (NextToken().type != RParen) throw SyntaxError();
      return new LeafNode(Num.data);
   } else if (t.type == Branch) {
      if (NextToken().type != LParen) throw SyntaxError();
      ParseTree left = BinTree();
      if (NextToken().type != Comma) throw SyntaxError();
      ParseTree right = BinTree();
      if (NextToken().type != RParen) throw SyntaxError();
      return new BranchNode(left, right);
   }
}
```

Rekursiv medåkning för aritmetiska uttryck?

För att skriva en parser för detta med rekursiv medåkning ska vi ha tre funktioner `Expr`, `Term`, `Factor`

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Term} \mid \\
&\quad\ \text{Expr} + \text{Term} \mid \\
&\quad\ \text{Expr} - \text{Term} \\
\text{Term} &\rightarrow \text{Factor} \mid \\
&\quad\ \text{Term} * \text{Factor} \mid \\
&\quad\ \text{Term} / \text{Factor} \\
\text{Factor} &\rightarrow \text{Num} \mid (\text{Expr})
\end{aligned}
$$

Rekursiv medåkning för aritmetiska uttryck?

För att skriva en parser för detta med rekursiv medåkning ska vi ha tre funktioner `Expr`, `Term`, `Factor`

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Term} \mid \\
&\quad \text{Expr} + \text{Term} \mid \\
&\quad \text{Expr} - \text{Term} \\
\text{Term} &\rightarrow \text{Factor} \mid \\
&\quad \text{Term} * \text{Factor} \mid \\
&\quad \text{Term} / \text{Factor} \\
\text{Factor} &\rightarrow \text{Num} \mid (\text{Expr})
\end{aligned}
$$

```
ParseTree Factor() {
   Token t = NextToken();
   if (t.type == Num) { // produktionsregel 1
      return NumberNode(t.value);
   } else { // produktionsregel 2
      if (t.type != LParen) throw SyntaxError();
      ParseTree subexpression = Expr();
      if (NextToken().type != RParen) throw SyntaxError();
   }
}
```

Rekursiv medåkning för aritmetiska uttryck?

För att skriva en parser för detta med rekursiv medåkning ska vi ha tre funktioner `Expr`, `Term`, `Factor`

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Term} \mid \\
&\quad\ \text{Expr} + \text{Term} \mid \\
&\quad\ \text{Expr} - \text{Term} \\
\text{Term} &\rightarrow \text{Factor} \mid \\
&\quad\ \text{Term} * \text{Factor} \mid \\
&\quad\ \text{Term} / \text{Factor} \\
\text{Factor} &\rightarrow \text{Num} \mid (\text{Expr})
\end{aligned}
$$

```
ParseTree Factor() {
   Token t = NextToken();
   if (t.type == Num) { // produktionsregel 1
      return NumberNode(t.value);
   } else { // produktionsregel 2
      if (t.type != LParen) throw SyntaxError();
      ParseTree subexpression = Expr();
      if (NextToken().type != RParen) throw SyntaxError();
   }
}

ParseTree Term() {
   Token t = NextToken();
   TODO: hur veta vilken produktionsregel som ska användas???
}
```

LL(1)-grammatiker

En grammatik som man kan parsa med rekursiv medåkning kallas för en LL(1)-grammatik

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Term} \mid \\
&\quad\ \text{Expr} + \text{Term} \mid \\
&\quad\ \text{Expr} - \text{Term} \\
\text{Term} &\rightarrow \text{Factor} \mid \\
&\quad\ \text{Term} * \text{Factor} \mid \\
&\quad\ \text{Term} / \text{Factor} \\
\text{Factor} &\rightarrow \text{Num} \mid (\text{Expr})
\end{aligned}
$$

LL(1)-grammatiker

En grammatik som man kan parsa med rekursiv medåkning kallas för en LL(1)-grammatik

Den här grammatiken för aritmetiska uttryck är inte LL(1) och kan inte parsas med rekursiv medåkning

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Term} \mid \\
&\quad\ \text{Expr} + \text{Term} \mid \\
&\quad\ \text{Expr} - \text{Term} \\
\text{Term} &\rightarrow \text{Factor} \mid \\
&\quad\ \text{Term} * \text{Factor} \mid \\
&\quad\ \text{Term} / \text{Factor} \\
\text{Factor} &\rightarrow \text{Num} \mid (\text{Expr})
\end{aligned}
$$

LL(1)-grammatiker

En grammatik som man kan parsa med rekursiv medåkning kallas för en LL(1)-grammatik

Den här grammatiken för aritmetiska uttryck är inte LL(1) och kan inte parsas med rekursiv medåkning

$$
\begin{aligned}
\text{Expr} &\rightarrow \text{Term} \mid \\
&\quad\ \text{Expr} + \text{Term} \mid \\
&\quad\ \text{Expr} - \text{Term} \\
\text{Term} &\rightarrow \text{Factor} \mid \\
&\quad\ \text{Term} * \text{Factor} \mid \\
&\quad\ \text{Term} / \text{Factor} \\
\text{Factor} &\rightarrow \text{Num} \mid (\text{Expr})
\end{aligned}
$$

(Ettan står för att vi tittar ett token framåt och bestämmer oss för vilken produktionsregel som ska användas, mer generellt finns LL(k) där vi tittar på de kommande k tokens när vi ska välja produktionsregel)

Nästa föreläsning

Olika typer av grammatiker

- Kontextfria
- LL-grammatiker
- LR-grammatiker

Kanske lite om parser-generatorer

Sammanfattning av kursavsnittet och anvisningar för kontrollskrivningen