

Kap. 5, Del 2:  
SLR(1), LR(1)- og LALR(1)-grammatikker  
INF5110 – 22/2-2011



---

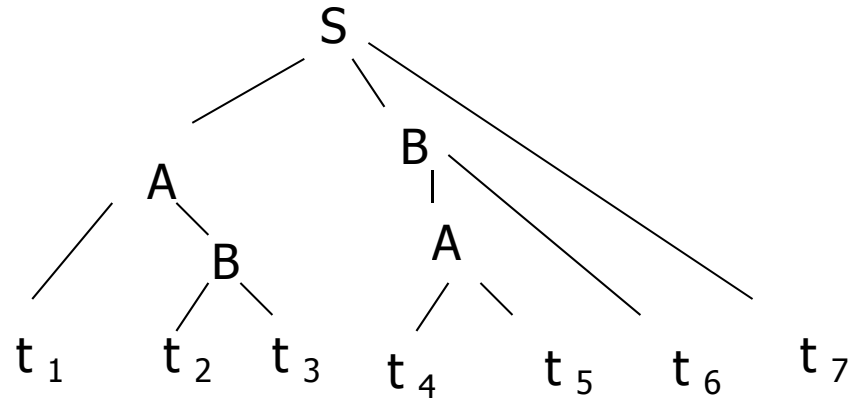
Stein Krogdahl,  
Ifi, UiO

**Oppgaver til kap 4:**

På slutten av dagens foiler ligger noen oppgaver med svarforslag. Disse vil bli forholdsvis raskt gjennomgått onsdag 23/2. **For fullt utbytt: Se på dem på forhånd!**



# “Bottom up” parsing (nedenfra-og-opp)



## LR-parsing og grammatikker:

- LR(0) Det teoretisk sterkeste, om man ikke vil se på noe lookahead-symbol
- SLR(1) “Simple LR”, en enkel bruk av LR(0)-DFA’en, ut fra Follow-mengder
- LALR(1) “LookAhead-LR”, veldig likt SLR, men med mer presise lookahed-mengder
- LR(1) Det teoretisk sterkeste, om man vil se på ett lookahead-symbol

## -Automatisert:

- CUP, YACC, Bison ( bruker LALR(1) )

# SLR(1) - grammatikker, SLR(1) - algoritmer

- Svært få grammatikker er LR(0)
- Ved å se på Follow-mengdene kan vi få en mye sterkere algoritme
- Tar også nå utgangspunkt i LR(0)-DFA'en
- Tabellene er nesten like, men nå må "reduser-linjene" spesifiseres for hvert mulig "neste input-symbol".

.....  
A → α.  
...  
B → β.

.....  
A → α.  
...  
B<sub>1</sub> → β<sub>1</sub>. b<sub>1</sub> γ<sub>1</sub>  
...  
B<sub>2</sub> → β<sub>2</sub>. b<sub>2</sub> γ<sub>2</sub>

LR(0): Har her en (uløselig) red./red.-konflikt

SLR(1): Dersom:  $\text{Follow}(A) \cap \text{Follow}(B) = \emptyset$  så kan konflikten løses ved å se på neste input:

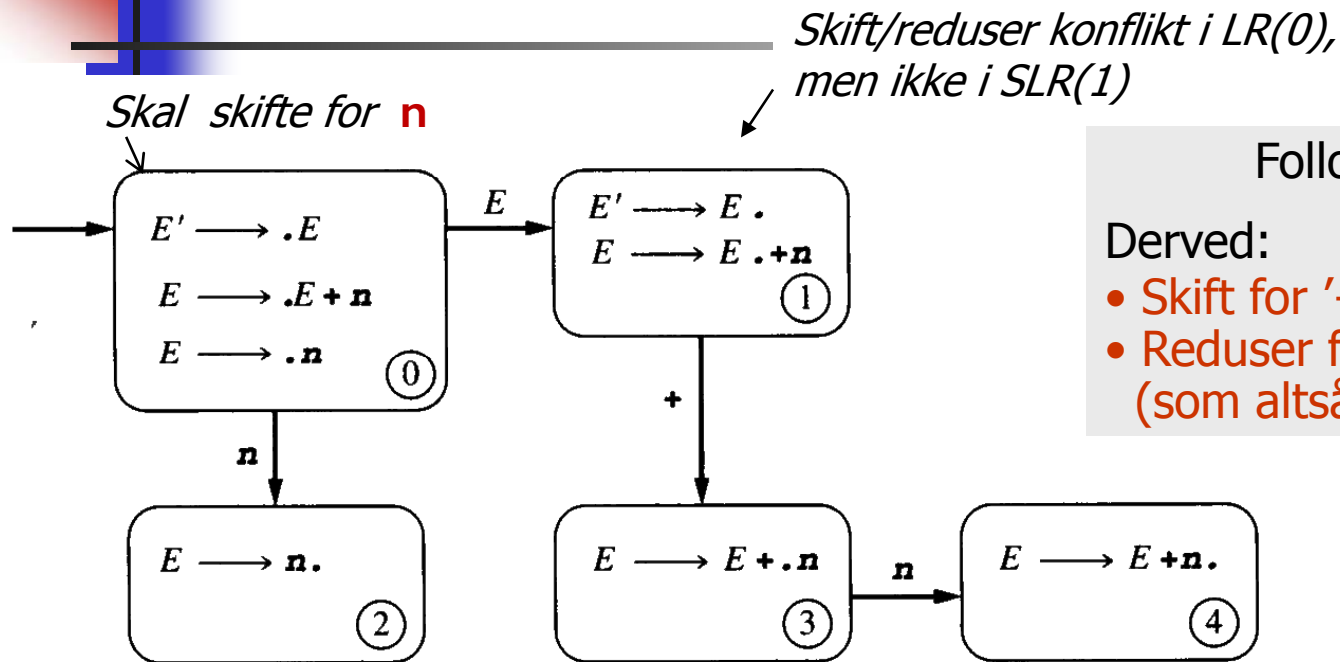
Om  $\text{token} \in \text{Follow}(A)$  reduser med  $A \rightarrow \alpha$   
Om  $\text{token} \in \text{Follow}(B)$  reduser med  $B \rightarrow \beta$   
(ellers feil)

LR(0): Har her en (uløselig) skift/red.-konflikt

SLR(1): Dersom:  $\text{Follow}(A) \cap \{b_1, b_2, \dots\} = \emptyset$  så kan konflikten løses ved å se på neste input:

Om  $\text{token} \in \text{Follow}(A)$ : reduser med  $A \rightarrow \alpha$   
For  $\text{token} = b_1, b_2, \dots$ : skift (input avgjør ny tilstand)  
(ellers feil)

# Er denne grammatikken SLR(1)?



Follow(E') = { \$ }

Derived:

- Skift for '+'
- Reduser for '\$', med  $E' \rightarrow E$  (som altså er accept)

## SLR(1)-kravet slik det er formulert i boka:

For alle DFA-tilstander  $s$  skal gjelde:

1. For any item  $A \rightarrow \alpha.X\beta$  in  $s$  with  $X$  a terminal, there is no complete item  $B \rightarrow \gamma.$  in  $s$  with  $X$  in Follow( $B$ ).
2. For any two complete items  $A \rightarrow \alpha.$  and  $B \rightarrow \beta.$  in  $s$ , Follow( $A$ )  $\cap$  Follow( $B$ ) is empty.

Ville ellers ha skift /  
reduser -konflikt ved  
input  $X$

Ville ellers ha reduser /  
reduser -konflikt ved input  $i$   
denne mengden

"Complete item" = Har punktumet til slutt



En tilsvarende formulering av SLR(1) kravet  
 Dersom følgende gir entydig algoritme, er grammatikken SLR(1)

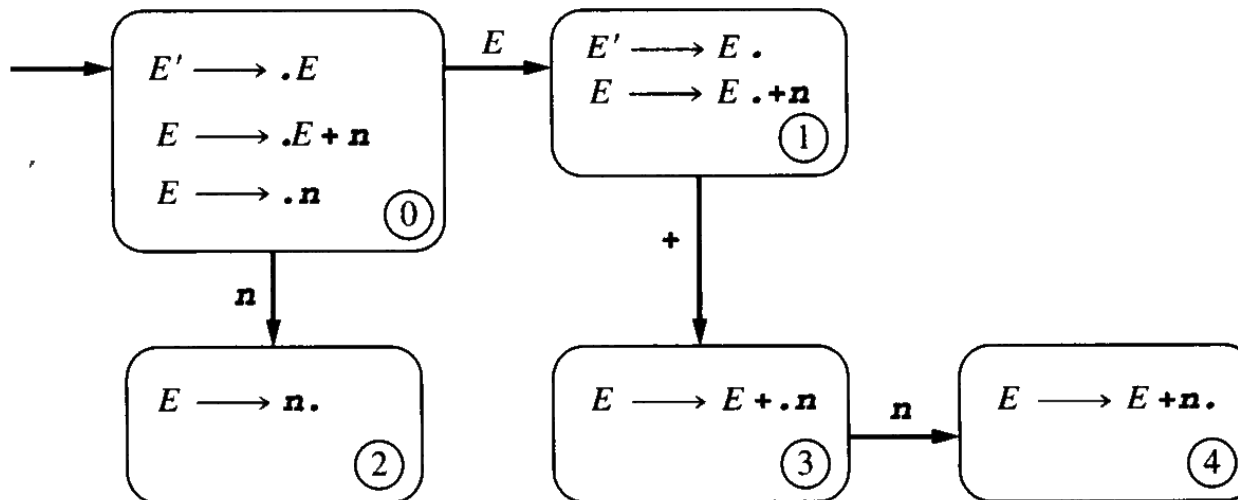
---

*The SLR(1) parsing algorithm.* Let  $s$  be the current state (at the top of the parsing stack). Then actions are defined as follows:

1. If state  $s$  contains any item of the form  $A \rightarrow \alpha.X\beta$ , where  $X$  is a terminal, and  $X$  is the next token in the input string, then the action is to shift the current input token onto the stack, and the new state to be pushed on the stack is the state ~~containing the item  $A \rightarrow \alpha.X\beta$~~   $t$ , where  $s \xrightarrow{X} t$
2. If state  $s$  contains the complete item  $A \rightarrow \gamma.$ , and the next token in the input string is in  $\text{Follow}(A)$ , then the action is to reduce by the rule  $A \rightarrow \gamma$ . A reduction by the rule  $S' \rightarrow S$ , where  $S$  is the start state, is equivalent to acceptance; this will happen only if the next input token is  $\$$ .<sup>4</sup> In all other cases, the new state is computed as follows. Remove the string  $\gamma$  and all of its corresponding states from the parsing stack. Correspondingly, back up in the DFA to the state from which the construction of  $\gamma$  began. By construction, this state ~~must contain an item of the form  $B \rightarrow \alpha.A\beta$~~   $u$ , where  $u \xrightarrow{A} t$
3. If the next input token is such that neither of the above two cases applies, an error is declared.

Dette er nytt i forhold til LR(0).

# Tabell-oppsett for SLR(1)-grammatikk



SLR(1): Både skift og reduser kan være på sammen linje. ("Accept" er egentlig reduksjon med  $A' \rightarrow A$ )

| State | Input |                            |                            | Goto |
|-------|-------|----------------------------|----------------------------|------|
|       | n     | +                          | \$                         |      |
| 0     | s2    |                            |                            | E    |
| 1     |       | s3                         | accept                     | 1    |
| 2     |       | r( $E \rightarrow n$ )     | r( $E \rightarrow n$ )     |      |
| 3     | s4    |                            |                            |      |
| 4     |       | r( $E \rightarrow E + n$ ) | r( $E \rightarrow E + n$ ) |      |

SLR(1)-kravet på en annen måte: Denne tabellen må være entydig!

# Parsering for SLR(1)-grammatikk

| State | Input |                            |                            | Goto |
|-------|-------|----------------------------|----------------------------|------|
|       | n     | +                          | \$                         |      |
| 0     | s2    |                            |                            | E    |
| 1     |       | s3                         | accept                     | 1    |
| 2     |       | r( $E \rightarrow n$ )     | r( $E \rightarrow n$ )     |      |
| 3     | s4    |                            |                            |      |
| 4     |       | r( $E \rightarrow E + n$ ) | r( $E \rightarrow E + n$ ) |      |

Kan også se på gale setninger som:

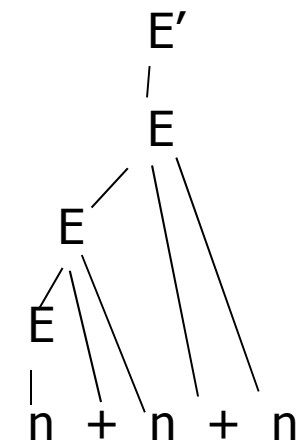
+ n \$

n n \$

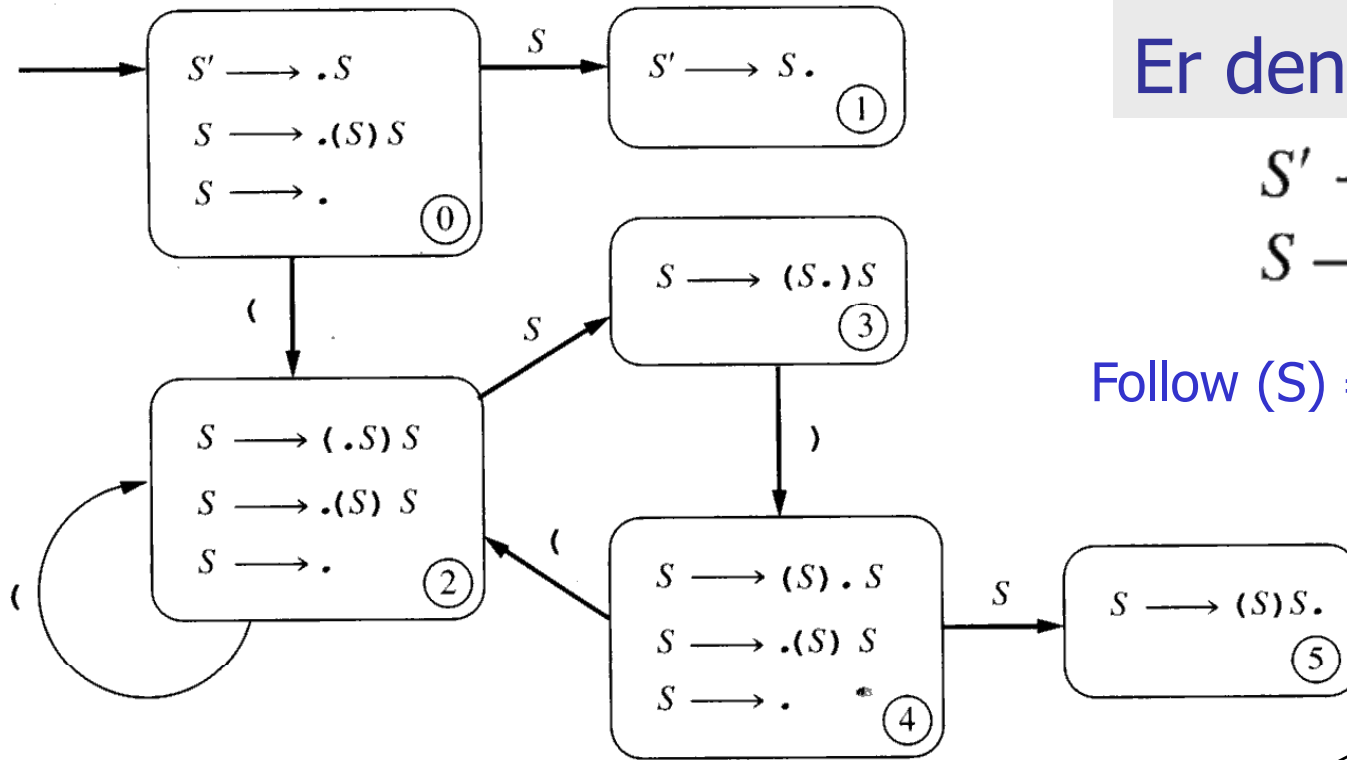
n + \$

Parsering av setningen: n + n + n

|   | Parsing stack    | Input        | Action                       |
|---|------------------|--------------|------------------------------|
| 1 | \$ 0             | n + n + n \$ | shift 2                      |
| 2 | \$ 0 n 2         | + n + n \$   | reduce $E \rightarrow n$     |
| 3 | \$ 0 E 1         | + n + n \$   | shift 3                      |
| 4 | \$ 0 E 1 + 3     | n + n \$     | shift 4                      |
| 5 | \$ 0 E 1 + 3 n 4 | + n \$       | reduce $E \rightarrow E + n$ |
| 6 | \$ 0 E 1         | + n \$       | shift 3                      |
| 7 | \$ 0 E 1 + 3     | n \$         | shift 4                      |
| 8 | \$ 0 E 1 + 3 n 4 | \$           | reduce $E \rightarrow E + n$ |
| 9 | \$ 0 E 1         | \$           | accept                       |



Merk at man ikke behøver å teste om det er mer input, siden man får accept bare foran \$ 7



Er denne SLR(1) ?

$$S' \rightarrow S$$

$$S \rightarrow ( S ) S \mid \epsilon$$

Follow (S) = { ), \$ }

Til senere:  
 Dette får vi i SLR(1), men ikke i LALR(1). Begge oppdager feilen, men LALR gjør det noe tidligere.

| State | Input |                |                | Goto |
|-------|-------|----------------|----------------|------|
|       | (     | )              | \$             | S    |
| 0     | s2    | r(S → ε)       | r(S → ε)       | 1    |
| 1     |       |                | accept         |      |
| 2     | s2    | r(S → ε)       | r(S → ε)       | 3    |
| 3     |       | s4             |                |      |
| 4     | s2    | r(S → ε)       | r(S → ε)       | 5    |
| 5     |       | r(S → ( S ) S) | r(S → ( S ) S) |      |





SLR( $k$ ) – mulig å lage teori for dette, men lite praktisk å få ut av det

---

### 5.3.4 SLR( $k$ ) Grammars

As with other parsing algorithms, the SLR(1) parsing algorithm can be extended to SLR( $k$ ) parsing where parsing actions are based on  $k \geq 1$  symbols of lookahead. Using the sets  $\text{First}_k$  and  $\text{Follow}_k$  as defined in the previous chapter, an SLR( $k$ ) parser uses the following two rules:

1. If state  $s$  contains an item of the form  $A \rightarrow \alpha.X\beta$  ( $X$  a token), and  $Xw \in \text{First}_k(X\beta)$  are the next  $k$  tokens in the input string, then the action is to shift the current input token onto the stack, and the new state to be pushed on the stack is the state containing the item  $A \rightarrow \alpha X.\beta$ .
2. If state  $s$  contains the complete item  $A \rightarrow \alpha.$ , and  $w \in \text{Follow}_k(A)$  are the next  $k$  tokens in the input string, then the action is to reduce by the rule  $A \rightarrow \alpha$ .

SLR( $k$ ) parsing is more powerful than SLR(1) parsing when  $k > 1$ , but at a substantial cost in complexity, since the parsing table grows exponentially in size with  $k$ .



## Flertydige grammatikker er aldri SLR(1) eller LR(1) (og de er heller ikke LL(1)!)

---

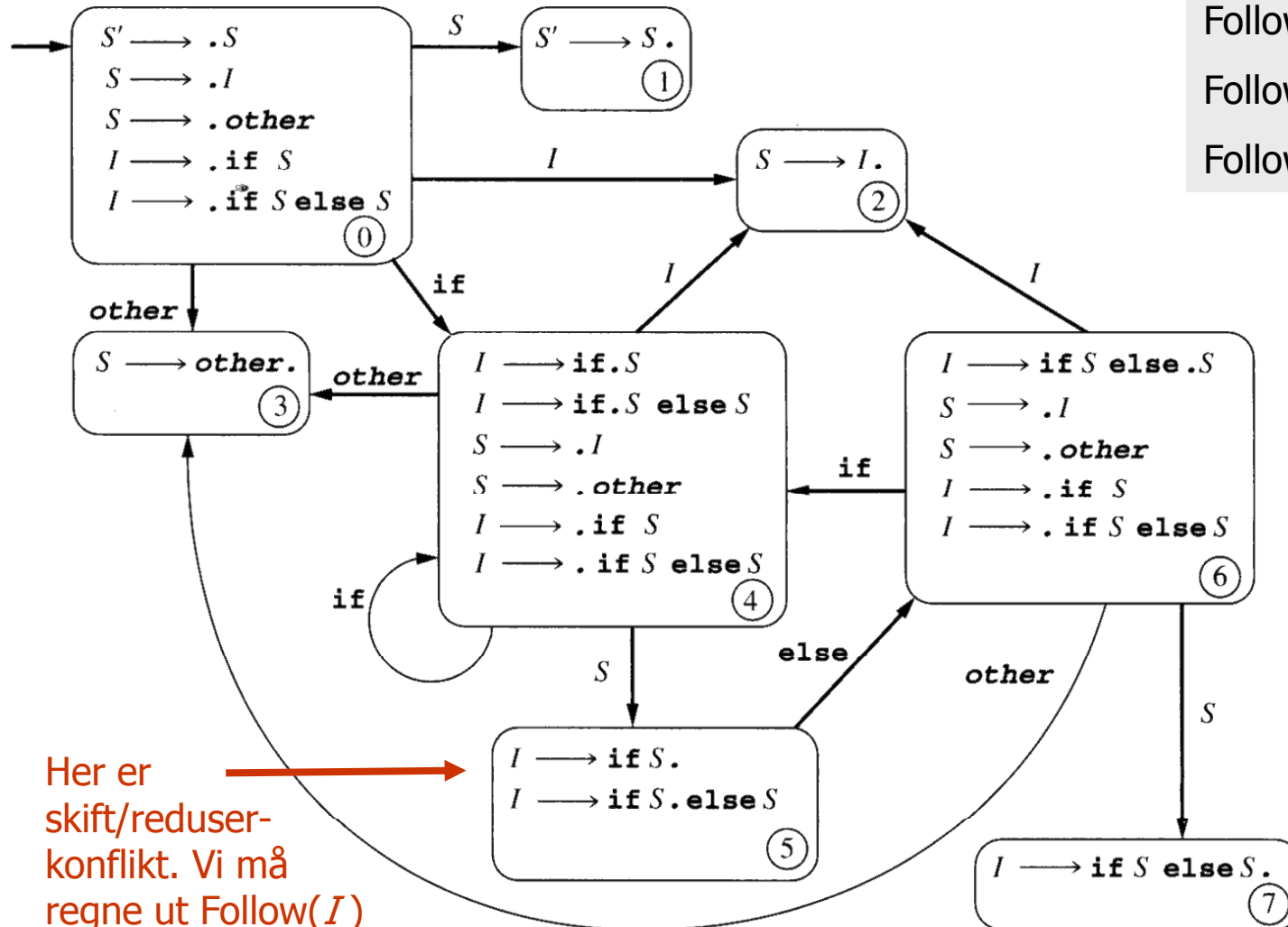
- Flertydige grammatikker er heller ikke SLR(k) eller LR(k) for noen k
- LR(0)-DFA'er for flertydige grammatikker vil derfor alltid ha "uløselige" konflikter, samme hvor mange tegn man får lov å se framover
  - Og det gjelder også LR(1)-DFA'er, LR(2)-DFA'er, osv
- **Men:** Konfliktene kan oftest likevel løses med intuisjon og fornuft, f.eks. slik:
  - For uttrykks-grammetikker: Angi presedens, assosiativitet e.l.
  - For "dangling else"-problemet: Om det er tvil, les videre uten å redusere
- Vanlig strategi i Yacc, CUP etc.:
  - Skift/Reduser-konflikter: **Bruker skift, om intet annet er angitt**

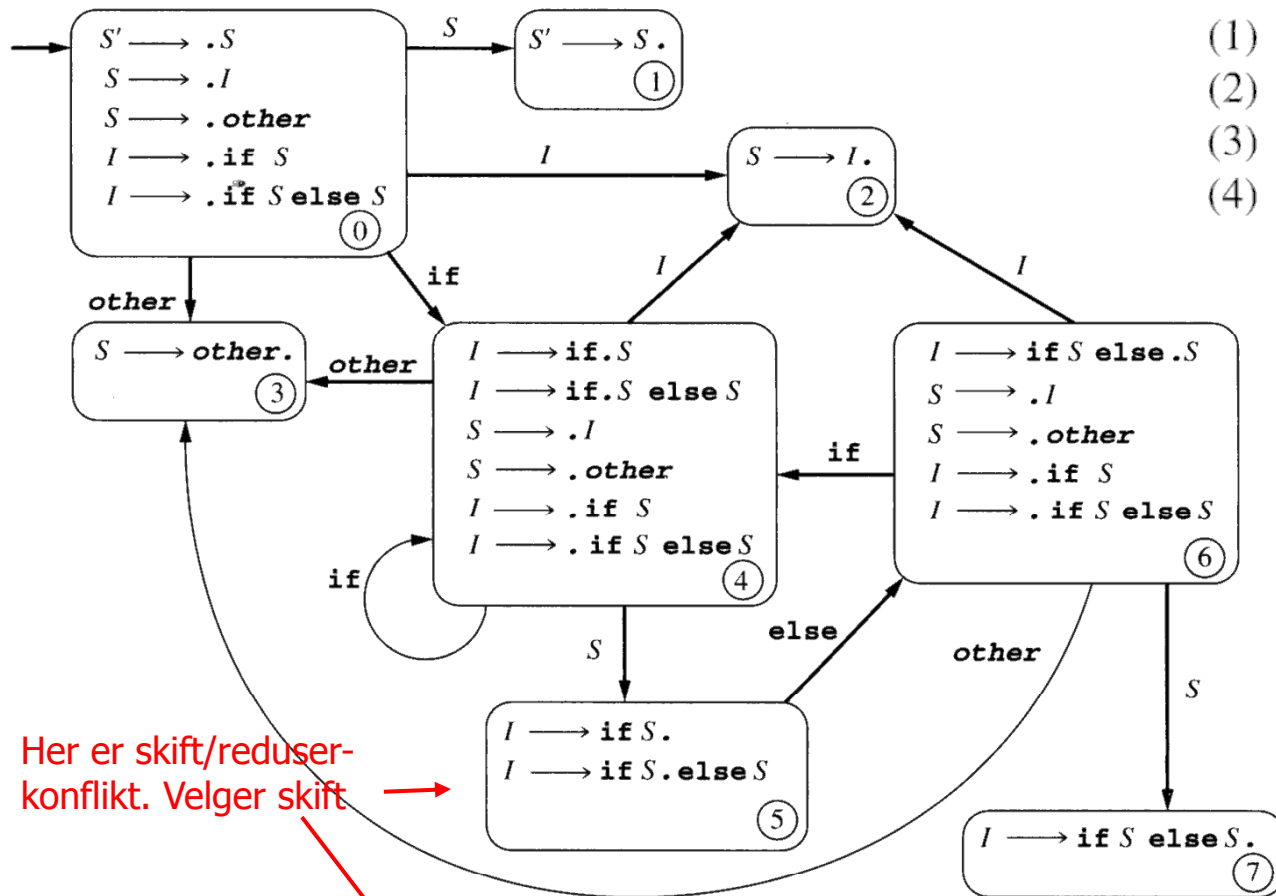
$statement \rightarrow if\text{-stmt} \mid \mathbf{other}$   
 $if\text{-stmt} \rightarrow \mathbf{if} ( exp ) statement$   
 $\quad \quad \quad \mid \mathbf{if} ( exp ) statement \mathbf{else} statement$   
 $exp \rightarrow 0 \mid 1$

$S \rightarrow I \mid \mathbf{other}$   
 $I \rightarrow \mathbf{if} S \mid \mathbf{if} S \mathbf{else} S$

Follow(S) = { \$, else }  
 Follow(I) = { \$, else }  
 Follow(S') = { \$ }

NB: Her *måtte* det bli  
 minst én konflikt,  
 siden grammatikken  
 er flertydig.





- (1)  $S \rightarrow I$
- (2)  $S \rightarrow \text{other}$
- (3)  $I \rightarrow \text{if } S$
- (4)  $I \rightarrow \text{if } S \text{ else } S$

Follow(S) = { \$, else }

Follow(I) = { \$, else }

Follow(S') = { \$ }

I tabellen betyr:

s3 – skift fra input til stakk.  
Legg så tilstand 3 på toppen av stakken

r3 – reduser ved regel 3 i grammatikken.  
Tilstanden på toppen av (den reduserte) stakken gir da ny tilstand ved "Goto"

Her er skift/reduser-konflikt. Velger skift

| State | Input |      |       |        | Goto |   |
|-------|-------|------|-------|--------|------|---|
|       | if    | else | other | \$     | S    | I |
| 0     | s4    |      | s3    |        | 1    | 2 |
| 1     |       |      |       | accept |      |   |
| 2     |       | r1   |       | r1     |      |   |
| 3     |       | r2   |       | r2     |      |   |
| 4     | s4    |      | s3    |        | 5    | 2 |
| 5     |       | s6   |       | r3     |      |   |
| 6     | s4    |      | s3    |        | 7    | 2 |
| 7     |       | r4   |       | r4     |      |   |

## SLR(1) tabell med "hånd-løst" konflikt (tilstand 5)

| State | Input |      |       |        | Goto |   |
|-------|-------|------|-------|--------|------|---|
|       | if    | else | other | \$     | S    | I |
| 0     | s4    |      | s3    |        | 1    | 2 |
| 1     |       |      |       | accept |      |   |
| 2     |       | r1   |       | r1     |      |   |
| 3     |       | r2   |       | r2     |      |   |
| 4     | s4    |      | s3    |        | 5    | 2 |
| 5     |       | s6   |       | r3     |      |   |
| 6     | s4    |      | s3    |        | 7    | 2 |
| 7     |       | r4   |       | r4     |      |   |

### Parsering:

\$ 0                    if if other else other \$

\$ 0 if 4                if other else other \$

\$ 0 if 4 if 4            other else other \$

\$ 0 if 4 if 4 other 3        else other \$

\$ 0 if 4 if 4 S 5            else other \$

\$ 0 if 4 if 4 S 5 **else 6**        other \$

(1)  $S \rightarrow I$

(2)  $S \rightarrow \mathbf{other}$

(3)  $I \rightarrow \mathbf{if} S$

(4)  $I \rightarrow \mathbf{if} S \mathbf{else} S$

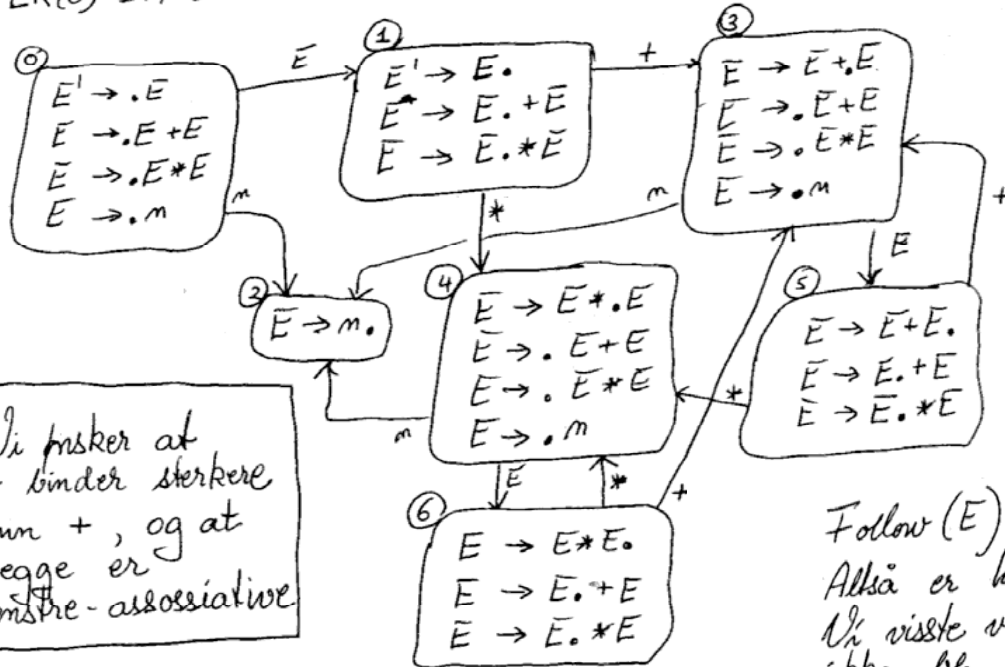
# Mer bruk av flertydige grammatikker ved LR-parsering (ikke i boka, men pensum)

Eksempel: Enkel uttrykk-grammatikk

$E' \rightarrow E$   
 $E \rightarrow E + E \mid E * E \mid m$

Vi vil at denne grammatikken er flertydig

LR(0)-DFA'en:



Vi ønsker at  $*$  binder sterkere enn  $+$ , og at begge er venstre-assosiative.

**Fordel ved flertydige grammatikker:**  
 De er som regel enklere å sette opp, se f.eks. til venstre her, og tidligere grammatikker for if-setningen

**Konflikter må oppstå, men:**  
 man kan løse mange konflikter ved å angi presedens, assosiativitet, m.m. Dette kan f.eks. gjøres i CUP og Yacc

Tilstand 5: **Stakk= .....E+E** Input=  
 $\$$ : reduser, fordi skift ikke lovlig for  $\$$   
 $+$ : reduser, fordi  $+$  er venstreassosiativ  
 $*$ : skift, fordi  $*$  har presedens over  $+$

Tilstand 6: **Stakk= .....E\*E** Input=  
 $\$$ : reduser, fordi skift ikke lovlig for  $\$$   
 $+$ : reduser, fordi  $*$  har presedens over  $+$   
 $*$ : reduser, fordi  $*$  er venstreassosiativ

Hva om også  $**$ ? (høyreass.). Blir oppgave!

Follow( $E$ ) =  $\{+, *, \$\}$   
 Alltså er hverken 5 eller 6 SLR-tilstander.  
 Vi visste vi måtte få konflikter slik at grammatikken ikke ble SLR-språk ingen flertydige grammatikker er SLR.  
 Hva skal vi så gjøre i tilstand 5 og 6?

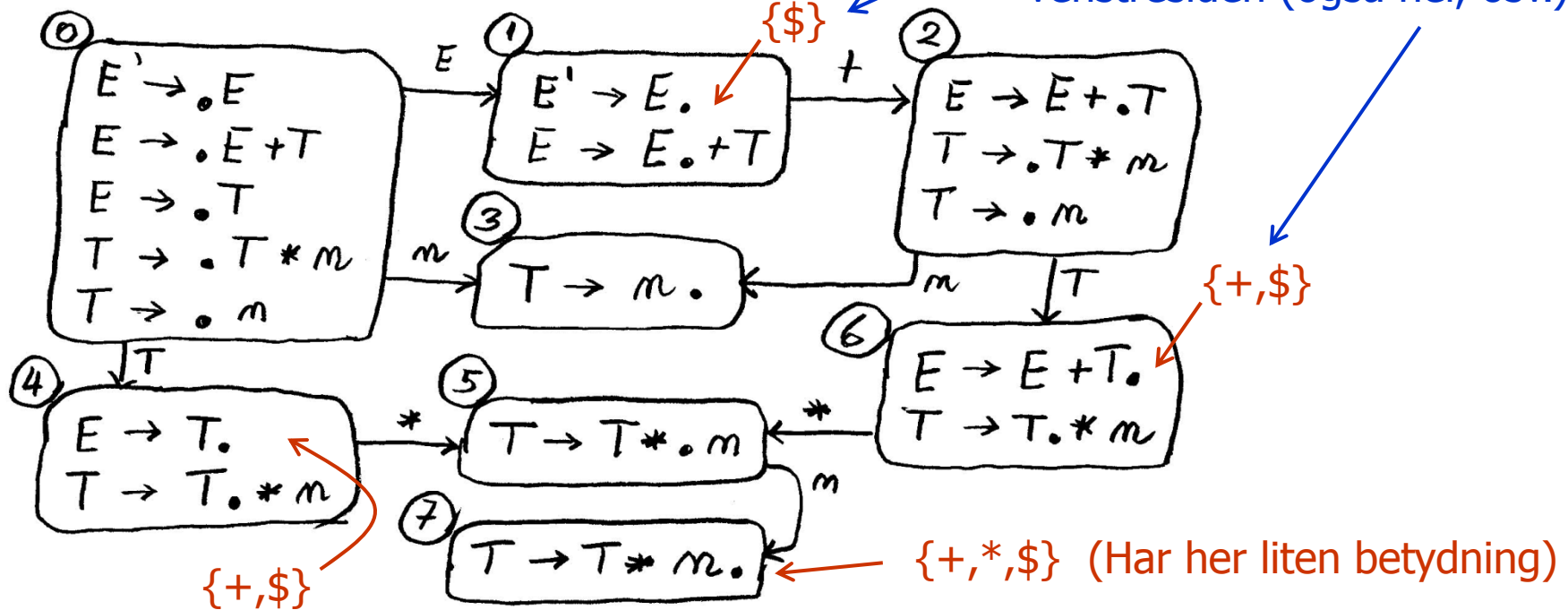
|   | $m$ | $+$      | $*$      | $\$$     | $E$ |
|---|-----|----------|----------|----------|-----|
| 0 | s2  |          |          | accept   |     |
| 1 |     | s3       | s4       |          |     |
| 2 |     | r(E→m)   | r(E→m)   | r(E→n)   | 5   |
| 3 | s2  |          |          |          | 6   |
| 4 | s2  |          |          |          |     |
| 5 |     | r(E→E+E) | s4       | r(E→E+E) |     |
| 6 |     | r(E→E*E) | r(E→E*E) | r(E→E*E) |     |

# Til sammenlikning: Den tilsvarende entydige grammatikken, med innbakt prioritet og venstre-assosiativitet

$E' \rightarrow E$   
 $E \rightarrow E + T \mid T$   
 $T \rightarrow T * m \mid m$

Follow  
 $E': \{ \$ \}$  (gjelder alltid)  
 $E: \{ + \$ \}$   
 $T: \{ * + \$ \}$

Etterfølger-mengden til venstresiden (også her, osv.)



- LR(0)-DFA'en her har 8, og ikke 7 tilstander (som den flertydige)
- Grammatikken er, som vi ser, en SLR(1)-grammatikk

# Om å bygge tre ved LR-parsering.

Vi bruker den parallelle variabelstakken som CUP tilbyr!

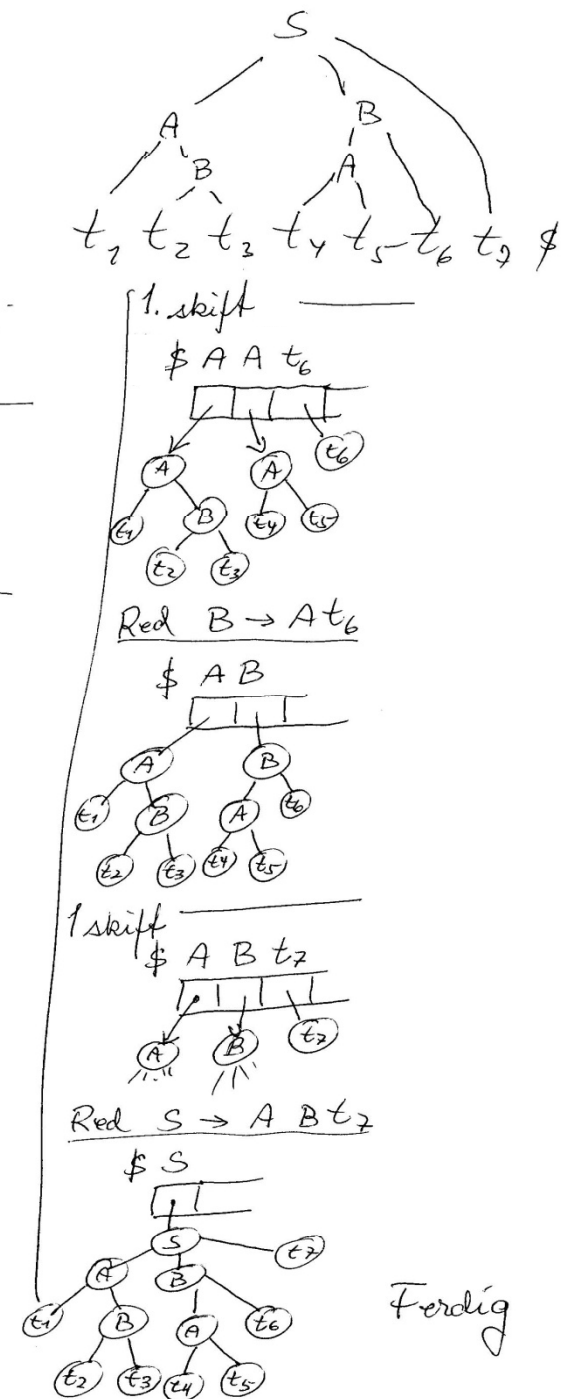
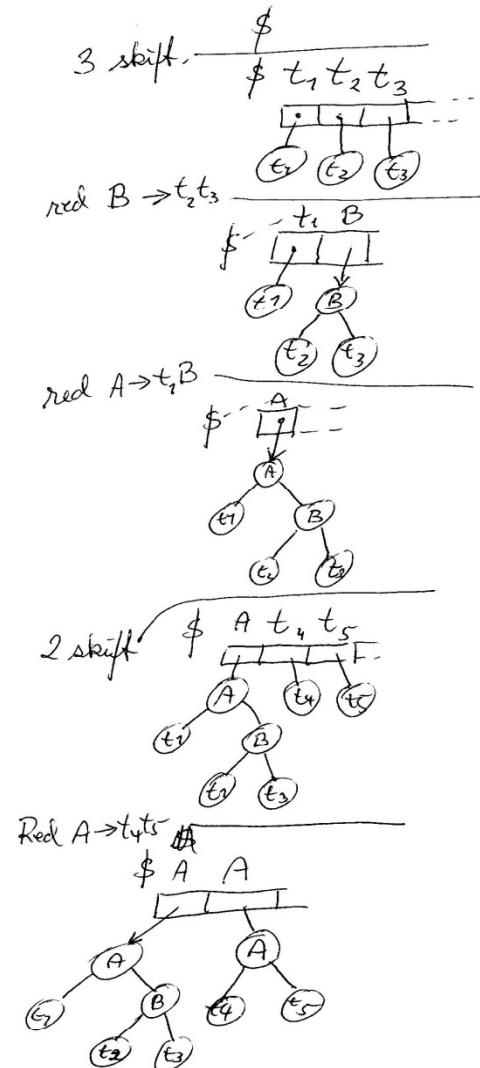
Ved skift:

Man skifter, og lager en ny node for den innkommende terminalen

Ved reduksjon:

Man lager en ny node for venstresiden i produksjonen og henger under denne det som lå tilsvarende høyresiden på stakken

**Konklusjon:** Etter hvert som man reduserer det tenkte treet dukker det opp igjen som et fysisk tre under stakken



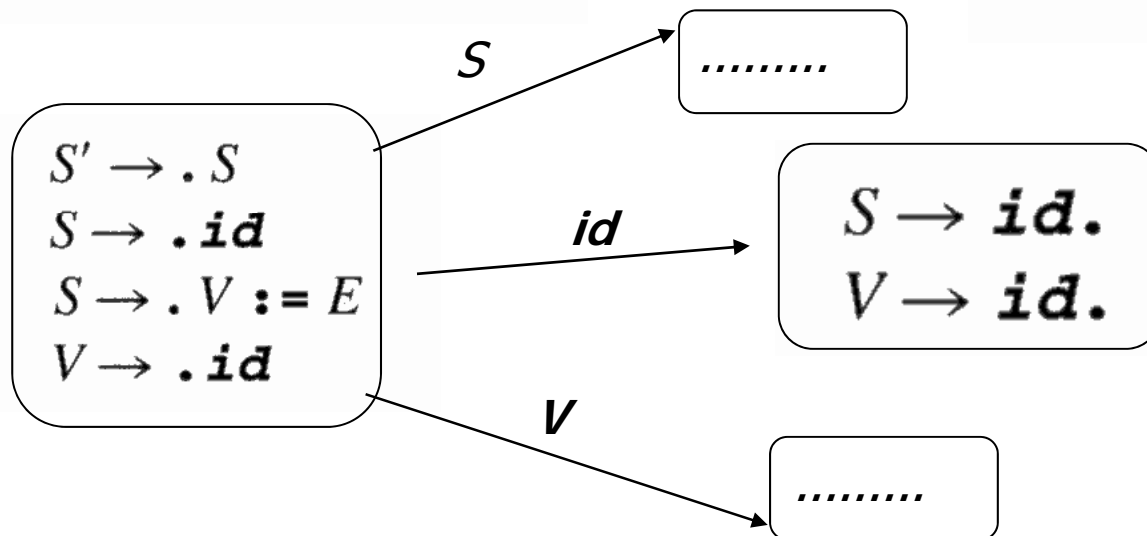


# Grammatikk som ikke er SLR(1)

Men denne viser det seg vi kan løse ved å være nøyaktigere med "etterfølgermengder" i LR(0)-DFA'en *under konstruksjonen*

$stmt \rightarrow call-stmt \mid assign-stmt$   
 $call-stmt \rightarrow identifier$   
 $assign-stmt \rightarrow var := exp$   
 $var \rightarrow var [ exp ] \mid identifier$   
 $exp \rightarrow var \mid number$

$S \rightarrow id \mid V := E$   
 $V \rightarrow id$   
 $E \rightarrow V \mid n$



Har har reduser/reduser-konflikt for input = \$

|   | First | Follow |
|---|-------|--------|
| S | id    | \$     |
| V | id    | :=, \$ |
| E | Id, n | \$     |

# Oppgaver til INF 5110, kapittel 4, med svarforslag

Gjennomgås tirsdag 23. febr. 2010

Oppgave 1: Sjekk om grammatikken  $S \rightarrow ( S ) S \mid \varepsilon$  er LL(1)

Oppgave 2: Gitt gram.:  $exp \rightarrow exp + exp \mid (exp) \mid \text{if } exp \text{ then } exp \text{ else } exp \mid \text{var}$

- Lag en entydig grammatikk for dette språket, der + skal være venstreassosiativ, og der "if x then y else z+u" skal bety "if x then y else (z+u)".
- Hvorfor får vi ikke noe "dangling else"-problem her?

Oppgave 3 (Mye repetisjon. Bli ikke fullt gjennomgått, man fullt svarforslag gis på foiler):

Gitt gram.:  $exp \rightarrow exp \text{ op } exp \mid (exp) \mid \text{num}$   
 $op \rightarrow + \mid - \mid * \mid / \mid ** \mid < \mid =$

- Grammatikken over er opplagt flertydig. Lag en *entydig* grammatikk for språket ut fra at følgende tilleggsregler:
  - \*\* (opphøying) har presedens 3 (høyest) og er høyre-assosiativ
  - \* og / har presedens 2, og er venstre-assosiativ
  - + og - har presedens 1 og er venstre-assosiativ
  - < og = har presedens 0, og er ikke-assosiativ
- Se på grammatikken du fant under a), og skriv et syntaksdiagram (med løkker der det passer) for hver ikke-terminal. Del opp "op"-terminalene på hensiktsmessig måte.
- Lag recursive-descent prosedyrer for å sjekke programmet (med while-setninger der det passer) ut fra grammatikken fra b). Du kan bruke både "match(token)" og "getToken()" fra boka (som begge setter neste symbol inn i variabelen "token").
- Ut fra svaret på c), legg til trebyggings-setninger i prosedyren som behandler en sekvens av \*\* slik at treet får riktig høyre-assosiativ form.
- Ta hele grammatikken fra a), og gjør den fri for venstreassosiativitet, og gjør all mulig venstrefaktorisering (men behold entydighet).
- Sjekk om grammatikken fra e) er LL(1).

# Oppgave 1

Sjekk om grammatikken " $S \rightarrow ( S ) S \mid \epsilon$ " er LL(1)

|   | First        | Follow |
|---|--------------|--------|
| S | $\epsilon$ ( | ) \$   |

Tabellen for valg av alternativ blir dermed:

|   | (                       | )                        | \$                       |
|---|-------------------------|--------------------------|--------------------------|
| S | $S \rightarrow ( S ) S$ | $S \rightarrow \epsilon$ | $S \rightarrow \epsilon$ |

Og denne tabellen er entydig, altså er den LL(1).

En recursive descent prosedyre kunne bli (ikke spurt om i oppgaven):

```
procedure S() {  
  if token = "(" then {  
    getToken();  
    S();  
    match( ")" );  
    S();  
  } else {  
    // ingen ting  
  }  
}
```

## For interesserte:

Grammatikken:  $S \rightarrow S ( S ) \mid \epsilon$   
(som gir samme språk) er derimot *ikke* LL(1). Vi får her:

|   | First        | Follow   |
|---|--------------|----------|
| S | $\epsilon$ ( | ) \$ ( ) |

Dermed blir det konflikt for "("

**DESSUTEN er den altså venstrekursiv**, så vi kunne egentlig umiddelbart sagt at den ikke er LL(1)!



## Oppgave 2

---

**Oppgave:** Gitt gram.:  $\text{exp} \rightarrow \text{exp} + \text{exp} \mid (\text{exp}) \mid \text{if exp then exp else exp} \mid \text{var}$

- a) Lag en entydig grammatikk for dette språket, der + skal være venstreassosiativ, og der "if x then y else z+u" skal bety "if x then y else (z+u)".

$\text{exp} \rightarrow \text{exp} + \text{exp1} \mid \text{exp1}$

$\text{exp1} \rightarrow \text{if exp then exp else exp} \mid (\text{exp}) \mid \text{var}$

Denne *er* entydig (den er SLR(1), som vi kommer til). Merk at vi f.eks. kan ha setningen:

$a + b + \text{if } c \text{ then } d \text{ else } e + f$ . Denne vil bli tolket slik:

$(a + b) + (\text{if } c \text{ then } d \text{ else } (e + f))$ .

Setningen:

$(a + b) + (\text{if } c \text{ then } d \text{ else } (\text{if } g \text{ then } h \text{ else } (e + f)))$  får den betydningen som angitt om den skrives helt uten parenteser.

- b) Hvorfor får vi ikke noe "dangling else"-problem her?

Det kommer av at det ikke er noe tvil om det skal være med en *else* eller ikke til en *if-then*. Det skal *alltid* være med en else!



# Oppgave 3a

Gitt grammatikken:

$$\begin{aligned} \text{exp} &\rightarrow \text{exp op exp} \mid (\text{exp}) \mid \text{num} \\ \text{op} &\rightarrow + \mid - \mid * \mid / \mid ** \mid < \mid = \end{aligned}$$

- a) Grammatikken over er opplagt flertydig. Lag en entydig grammatikk for språket ut fra at følgende tilleggsregler:

\*\* (opphøying) har presedens 3 (høyest) og er høyre-assosiativ  
\* og / har presedens 2, og er venstre-assosiativ  
+ og - har presedens 1 og er venstre-assosiativ  
< og = har presedens 0, og er ikke-assosiativ

## Svarforslag:

Vi må lage en ny ikke-terminal for hvert presedens-nivå. Vi velger fra laveste til høyeste:

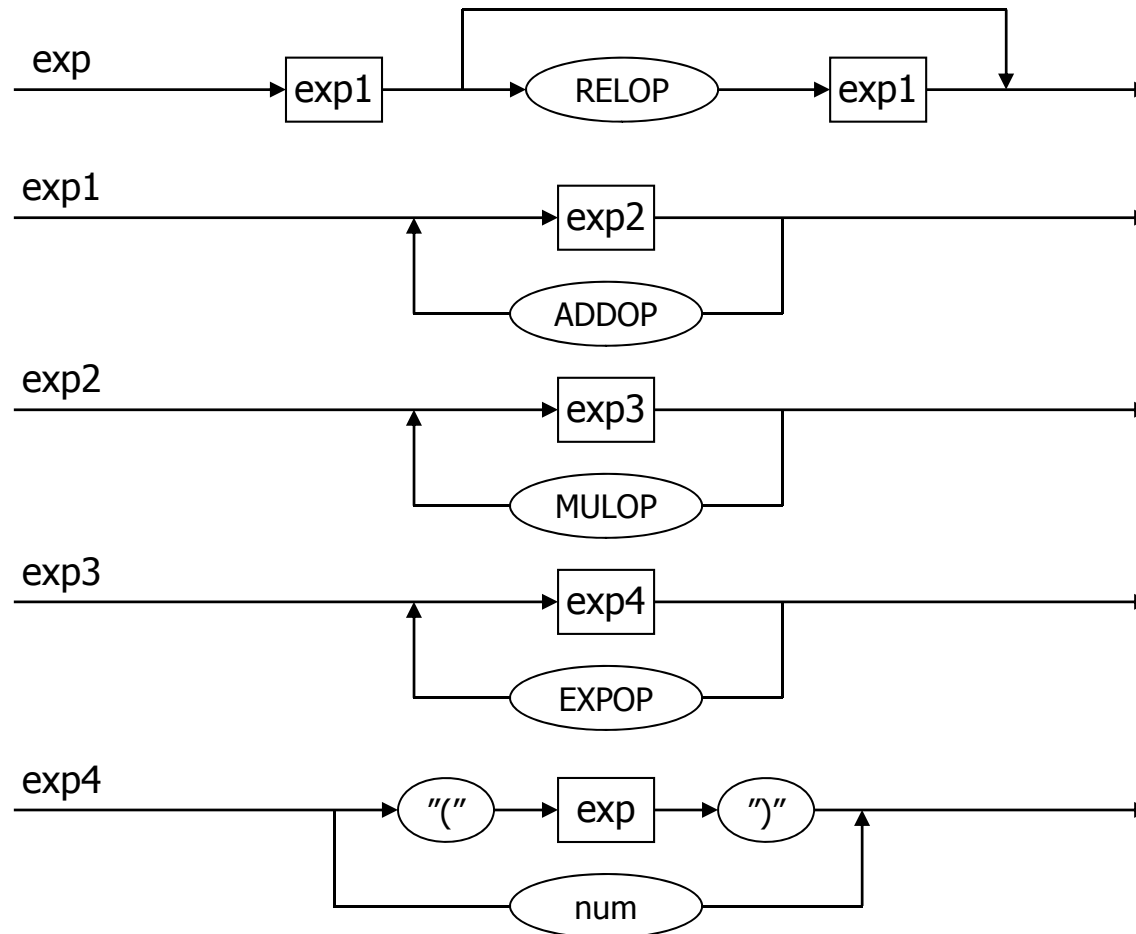
|      |   |             |
|------|---|-------------|
| exp  | som den er i oppgaven                     |             |
| exp1 | for operander foran og bak < og =         |             |
| exp2 | for operander mellom, foran og bak + og - | (term)      |
| exp3 | for operander mellom, foran og bak * og / | (faktor)    |
| exp4 | for operander mellom, foran og bak **     | (opphøying) |

## Grammatikken blir:

|      |   |   |
|------|---|---|
| exp  | $\rightarrow \text{exp1 RELOP exp1} \mid \text{exp1}$ | RELOP dekker < og =, men kommer som samme token |
| exp1 | $\rightarrow \text{exp1 ADDOP exp2} \mid \text{exp2}$ | Tilsvarende for + og -                          |
| exp2 | $\rightarrow \text{exp2 MULOP exp3} \mid \text{exp3}$ | Tilsvarende for * og /                          |
| exp3 | $\rightarrow \text{exp4 EXPOP exp3} \mid \text{exp4}$ | Tilsvarende, men snudd, for **                  |
| exp4 | $\rightarrow (\text{exp}) \mid \text{num}$            |   |

## Oppgave 3b

**Oppgaven:** Se på grammatikken du fant under a), og skriv et syntaksdiagram (med løkker der det passer) for hver ikke-terminal. Del opp "op"- terminalene på hensiktsmessig måte.



**Merk:**  
Assosiativitet  
kommer ikke  
fram her. Det  
må eventuelt  
legges inn i  
trebyggingen i  
spørsmål d)



## Oppgave 3c

c)

**Oppgaven:** Lag recursive-descent prosedyrer for å sjekke programmet (med while-setninger der det passer) ut fra grammatikken fra b). Du kan bruke både "match(token)" og "getToken()" fra boka (som begge setter neste symbol inn i variabelen "token").

```
procedure exp( ) {
  exp1( );
  if token = RELOP then {
    getToken( )
    exp1( );
  }
}
```

```
procedure exp1( ) {
  exp2;
  while token = ADDOP do {
    getToken( );
    exp2( );
  }
}
```

Både exp2 og exp3 blir helt tilsvarende til exp1( )

```
procedure exp4( ) {
  if token = LPAR then {
    getToken( );
    expr( );
    match( RPAR )
  } else {
    match( NUM );
  }
}
```

---

Om "exp" er det faktisk **ytterste** startsymbolet (som ofte heter "program"), så legger man gjerne på en ytterste rec.descent-prosedyre som får det hele riktig i gang, og som sjekker at det ikke er noe grums **etter** programmet. Den kan f.eks. være slik:

```
procedure expression( ) {
  getToken( );
  exp( );
  if token != "$" then {error("grums etter
  uttrykket");}
}
```


## Oppgave 3d (se alternativ løsning på neste foil)

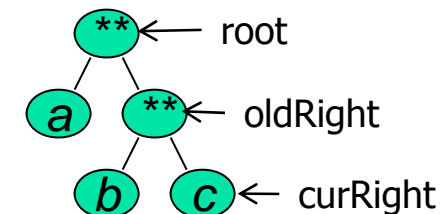
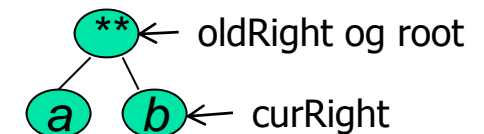
**NB:** Det var noe feil på denne foilen i det som er delt ut tidligere! Er rettet her og i de endelige foiler som legges ut.

**Oppgaven:** Ut fra svaret på c), legg til trebyggings-setninger i prosedyren som behandler en sekvens av \*\*, slik at treet får riktig høyre-assosiativ form.

```
procedure exp3(): TreeNode {  
    TreeNode oldRight, curRight, newRight, newOp;  
  
    curRight = exp4(); root = curRight; oldRight = null;  
    while token = EXPOP do {  
        getToken();  
        newRight = exp4();  
        newOp = new OpNode("**"); newOp.right = newRight;  
        if (oldRight == null) {  
            newOp.left = curRight ; root = newOp;  
        } else {  
            newOp.left = oldRight.right ; oldRight.right = newOp;  
        }  
        oldRight = newOp; curRight = newRight;  
    }  
    return root;  
}
```

$a ** b ** c$

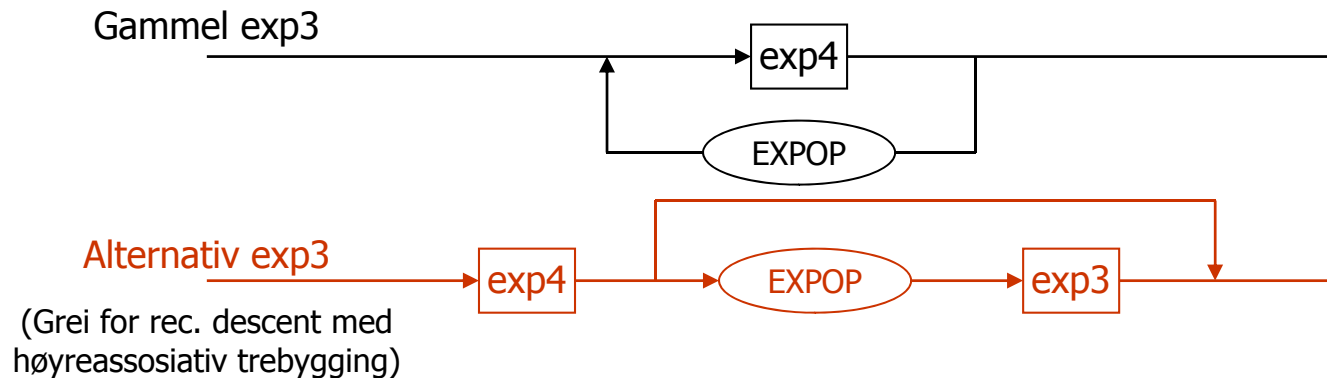
Før while-setn:   $oldRight == null$



$newRight$  og  $newOP$  er alltid ute av bruk mellom iterasjonene



## Oppgave 3d alternativ løsning



Over er angitt et alternativt syntaksdiagram for exp3. Dette kan gi en rec. desc. prosedyre, med greiere høyreassosiativ trebygging. Det kan være slik:

### Uten trebygging:

```
procedure exp3() {
  exp4;
  if token = EXPOP then {
    getToken();
    exp3();
  }
}
```

### Med trebygging:

```
procedure exp3(): TreeNode {
  TreeNode root; OpNode opNode;
  root = exp(4);
  if token = EXPOP then {
    getToken();
    newRight = exp3();
    root = new OpNode("***", root,
                      newRight);
  }
  return root;
}
```



## Oppgave 3e

---

- e) Ta hele grammatikken fra a), og gjør den fri for venstreassosiativitet, og gjør all mulig venstrefaktorisering (men behold entydighet).

$\text{exp} \rightarrow \text{exp1 RELOP exp1} \mid \text{exp1}$   
 $\text{exp1} \rightarrow \text{exp1 ADDOP exp2} \mid \text{exp2}$   
 $\text{exp2} \rightarrow \text{exp2 MULOP exp3} \mid \text{exp3}$   
 $\text{exp3} \rightarrow \text{exp4 EXPOP exp3} \mid \text{exp4}$   
 $\text{exp4} \rightarrow (\text{exp}) \mid \text{num}$

RELOP dekker  $<$  og  $=$ , men er samme token  
Tilsvarende for  $+$  og  $-$   
Tilsvarende for  $*$  og  $/$   
Tilsvarende for  $**$

$\text{exp} \rightarrow \text{exp1 expx}$   
 $\text{expx} \rightarrow \text{RELOP exp1} \mid \varepsilon$   
 $\text{exp1} \rightarrow \text{exp2 exp1x}$   
 $\text{exp1x} \rightarrow \text{ADDOP exp2 exp1x} \mid \varepsilon$   
 $\text{exp2} \rightarrow \text{exp3 exp2x}$   
 $\text{exp2x} \rightarrow \text{MULOP exp3 exp2x} \mid \varepsilon$   
 $\text{exp3} \rightarrow \text{exp4 exp3x}$   
 $\text{exp3x} \rightarrow \text{EXPOP exp3} \mid \varepsilon$   
 $\text{exp4} \rightarrow (\text{exp}) \mid \text{num}$

At vi faktisk beholder entydighet er ikke uten videre greit å se, men det blir klart i oppgave f), siden vi der finner at grammatikken er LL(1). Alle grammatikker som er LL(1) er entydige.

# Oppgave 3f

f) Sjekk om grammatikken fra e) er LL(1). Vi beregner først First og Follow (Fi og Fo). FiU er Fi uten  $\epsilon$ . Regner her RELOP, ADOP, MULOP og EXPOP som teminal-symboler. Vi gjentar de røde aksjonene til det stabiliserer seg.

|                                       |   |
|---------------------------------------|---|
| exp → exp1 expx                       | Legger Fo(exp) inn i Fo(expx) og inn i Fo(exp1). Legger FiU(expx) inn i Fo(exp1)                |
| expx → RELOP exp1   $\epsilon$        | Legger Fo(expx) inn i Fo(exp1). Legger RELOP inn i Fi(expx)                                     |
| exp1 → exp2 exp1x                     | Legger Fo(exp1) inn i Fo(exp1x) og inn i Fo(exp2). Legger FiU(exp1x) inn i Fo(exp2)             |
| exp1x → ADDOP exp2 exp1x   $\epsilon$ | Legger Fo(exp1x) inn i Fo(exp2). Legger FiU(exp1x) inn i Fo(exp2). Legger ADDOP inn i Fi(exp1x) |
| exp2 → exp3 exp2x                     | Legger Fo(exp2) inn i Fo(exp2x) og inn i Fo(exp3). Legger FiU(exp2x) inn i Fo(exp3)             |
| exp2x → MULOP exp3 exp2x   $\epsilon$ | Legger Fo(exp2x) inn i Fo(exp3). Legger FiU(exp2x) inn i Fo(exp3) Legger MULOP inn i Fi(exp2x)  |
| exp3 → exp4 exp3x                     | Legger Fo(exp3) inn i Fo(exp3x) og inn i Fo(exp4). Legger FiU(exp3x) inn i Fo(exp4)             |
| exp3x → EXPOP exp3   $\epsilon$       | Legger Fo(exp3x) inn i Fo(exp3). Legger EXPOP inn i Fi(exp3x)                                   |
| exp4 → ( exp )   num                  | Legger "(" inn i Fo(exp). Legger ")" og num inn i Fi(exp4)                                      |

|       | <b>First</b>     | <b>Follow</b>                |  |
|-------|------------------|------------------------------|--|
| exp   | num (            | \$ )                         | Legger først \$ inn i Follow(exp) (siden exp er startsymbolet) |
| expx  | $\epsilon$ RELOP | \$ )                         |  |
| exp1  | num (            | \$ ) RELOP                   |  |
| exp1x | $\epsilon$ ADDOP | \$ ) RELOP                   |  |
| exp2  | num (            | \$ ) RELOP ADDOP             |  |
| exp2x | $\epsilon$ MULOP | \$ ) RELOP ADDOP             |  |
| exp3  | num (            | \$ ) RELOP ADDOP MULOP       |  |
| Exp3x | $\epsilon$ EXPOP | \$ ) RELOP ADDOP MULOP       |  |
| exp4  | num (            | \$ ) RELOP MULOP ADDOP EXPOP |  |

|       | <b>RELOP</b> | <b>ADDOP</b>     | <b>MULOP</b>     | <b>EXPOP</b> | <b>num</b> | <b>(</b>   | <b>)</b>   | <b>\$</b>  |
|-------|--------------|------------------|------------------|--------------|------------|------------|------------|------------|
| exp   |              |                  |                  |              | exp1 expx  | exp1 expx  |            |            |
| expx  | RELOP exp1   |                  |                  |              |            |            | $\epsilon$ | $\epsilon$ |
| exp1  |              |                  |                  |              | exp2 exp1x | exp2 exp1x |            |            |
| exp1x | $\epsilon$   | ADDOP exp2 exp1x |                  |              |            |            | $\epsilon$ | $\epsilon$ |
| exp2  |              |                  |                  |              | exp3 exp2x | exp3 exp2x |            |            |
| exp2x | $\epsilon$   | $\epsilon$       | MULOP exp3 exp2x |              |            |            | $\epsilon$ | $\epsilon$ |
| exp3  |              |                  |                  |              | exp4 exp3x | exp4 exp3x |            |            |
| exp3x | $\epsilon$   | $\epsilon$       | $\epsilon$       | EXPOP exp3   |            |            | $\epsilon$ | $\epsilon$ |
| exp4  |              |                  |                  |              | num        | ( exp )    |            |            |

Dermed, siden det ikke er konflikter: Den er LL(1)! Og videre: Dermed også entydig.