Oppgaver til kap. 4 og 5 , med svarforslag

Gjennomgås onsdag 5. mars, 2014

Oppgavene til kapittel 4:

Oppgave 1: Sjekk om grammatikken ”S → (S) S | ε” er LL(1)

Oppgave 2: Gitt gram.: exp → exp **+** exp | **(**exp**)** | **if** exp **then** exp **else** exp | **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)”.

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

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

Gitt gram.: exp → exp op exp | (exp) | **num**
op → + | - | * | / | ** | < | =

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-assossiativ
* og / har presedens 2, og er venstre-assosiaiv
+ og - har presedens 1 og er venstre-assosiativ
< og = har presedens 0, og er ikke-assosiativ

b) 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.

c) 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 variablen ”token”).

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

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

f) Sjekk om grammatikken fra e) er LL(1).

Oppgave 1

Sjekk om grammatikken ”S → (S) S | ε” er LL(1)

	First	Follow
S	ε (	$)

Tabellen for valg av alternativ blir dermed:

	(	)	$
S	S →(S) S	S → ε	S → ε

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 → S (S) | ε (som gir samme språk) er derimot *ikke* LL(1). Vi får her:

	First	Follow
S	ε (	$ ( )

Dermed blir det konflikt for ”(”

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

Oppgave 2

Oppgave: Gitt gram.: exp → exp **+** exp | **(**exp**)** | **if** exp **then** exp **else** exp | **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)”.

exp → exp + exp1 | exp1
exp1 → if exp then exp else exp | (exp) | **var**

Denne *er* entydig (den viser seg å være SLR(1)). Vi kan f.eks. ha setningen:

a + b + if c then d else e + f. Denne vil bli tolket slik:

(a + b) + (if c then d else (e + f)).

Setningen:

(a + b) + (if c then d else (if g then h else (e + f)))

får den betydningen som angitt også om den skrives helt uten parenteser.

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

Det kommer av at det aldri 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: exp → exp op exp | (exp) | **num**
op → + | - | * | / | ** | < | =

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-assossiativ
* og / har presedens 2, og er venstre-assosiaiv
+ 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:

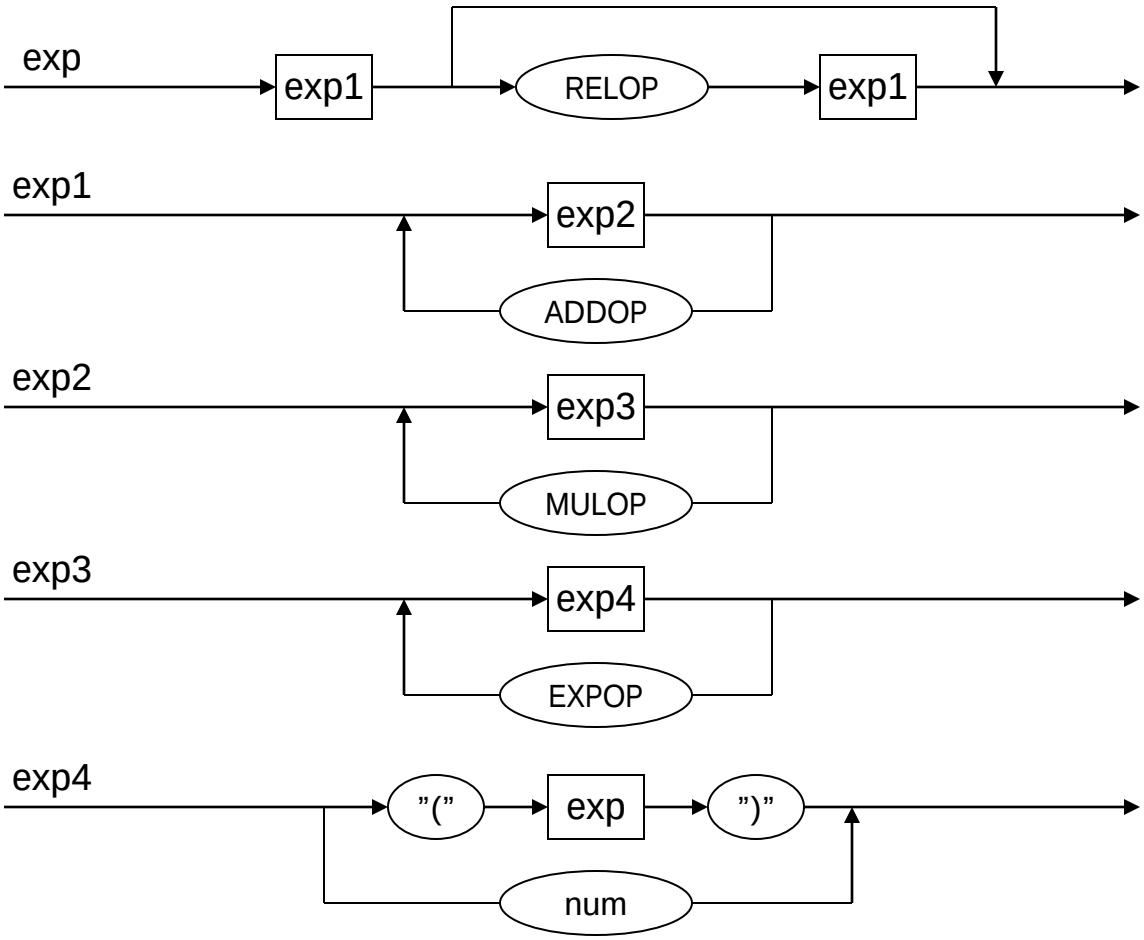
exp	som den er i oppgaven	
exp1	for operander foran og bak < og =	
exp2	for operander mellom, foran og bak + og –	(term)
exp3	for operander mellom, foran og bak * og /	(faktor)
exp4	for operander mellom, foran og bak **	(opphøying)

Grammatikken blir:

exp → exp1 RELOP exp1 | exp1	RELOP dekker < og =, men kommer som samme token
exp1 → exp1 ADDOP exp2 | exp2	Tilsvarende for + og -
exp2 → exp2 MULOP exp3 | exp3	Tilsvarende for * og /
exp3 → exp4 EXPOP exp3 | exp4	Tilsvarende, men snudd, for **
exp4 → (exp) | **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 variablen "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( ).

Høyere-assosiativiteten til exp3 kommer altså ikke fram her.

```
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 litt spesielll 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( );   //   Gjøres bare ev denne ytterste
                   //   rec.desc.- prosedyren
    exp( );
    if token != "$" then {error("grums etter
                                 uttrykket");}
}
```

Oppgave 3d (se alternativ løsning på neste foil)

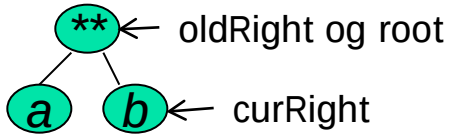
Oppgaven: Ut fra svaret på c), legg til trebyggings-setninger i prosedyren som behandler en sekvens av **, slik at treet får riktig høyre-assosiativ form.

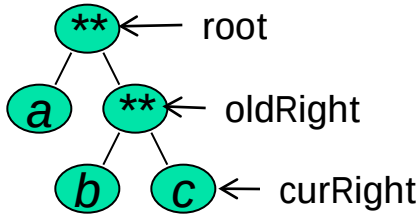
a ** b ** c

```
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;
}
```

oldRight == null

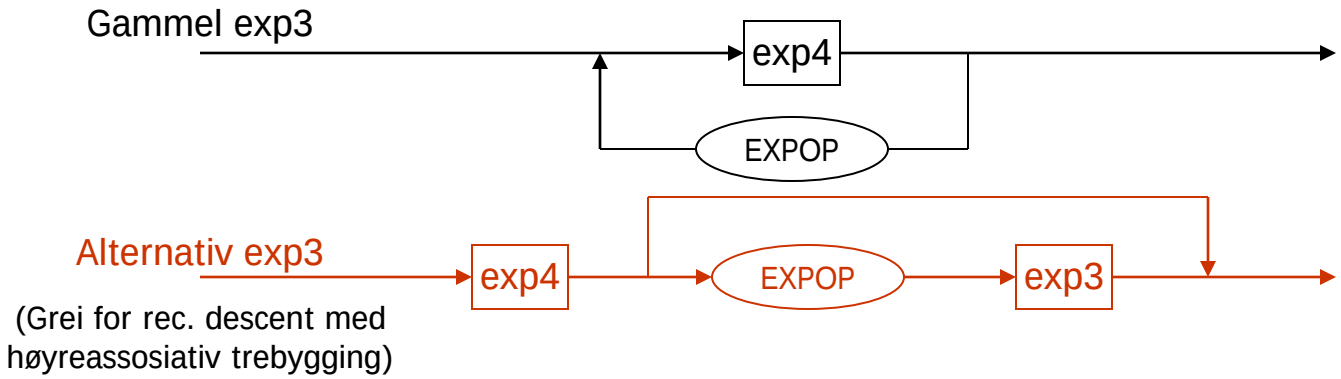
Før while-setn:





newRight og newOP er alltid ute av bruk mellom iterasjonene

Oppgave 3d alternativ løsning



(Grei for rec. descent med høyreassosiativ trebygging)

Over er angitt et alternativt syntaksdiagram for exp3. Dette kan gi en rec. desc. prosedyre, med greiere høyreassoiativ 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).

exp → exp1 RELOP exp1 | exp1 — RELOP dekker < og =, men er samme token
exp1 → exp1 ADDOP exp2 | exp2 — Tilsvarende for + og -
exp2 → exp2 MULOP exp3 | exp3 — Tilsvarende for * og /
exp3 → exp4 EXPOP exp3 | exp4 — Tilsvarende for **
exp4 → (exp) | **num**

exp → exp1 expx — Her gjør vi venstrefaktorisering. Ingen venstrerekuirsivitet
expx → RELOP exp1 | ε
exp1 → exp2 exp1x — Fjerning av venstrerekursivitet
exp1x → ADDOP exp2 exp1x | ε
exp2 → exp3 exp2x — Fjerning av venstrerekursivitet
exp2x → MULOP exp3 exp2x | ε
exp3 → exp4 exp3x — Her gjør vi venstrefaktorisering. Ingen venstrerekuirsivitet
exp3x → EXPOP exp3 | ε
exp4 → (exp) | num — Behøver ingen omskriving

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 ε. 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 | ε	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 | ε	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 | ε	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 | ε	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)

	First	**Follow**	
exp	num (	$)	Legger først $ inn i Follow(exp) (siden exp er startsymbolet)
expx	ε RELOP	$)	
exp1	num (	$) RELOP	
exp1x	ε ADDOP	$) RELOP	
exp2	num (	$) RELOP ADDOP	
exp2x	ε MULOP	$) RELOP ADDOP	
exp3	num (	$) RELOP ADDOP MULOP	
Exp3x	ε EXPOP	$) RELOP ADDOP MULOP	
exp4	num (	$) RELOP MULOP ADDOP EXPOP	

	RELOP	**ADDOP**	**MULOP**	**EXPOP**	**num**	**(**	**)**	**$**
exp					exp1 expx	exp1 expx		
expx	RELOP exp1						ε	ε
exp1					exp2 exp1x	exp2 exp1x		
exp1x	ε	ADDOP exp2 exp1x					ε	ε
exp2					exp3 exp2x	exp3 exp2x		
exp2x	ε	ε	MULOP exp3 exp2x				ε	ε
exp3					exp4 exp3x	exp4 exp3x		
exp3x	ε	ε	ε	EXPOP exp3			ε	ε
exp4					num	(exp)		

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

Oppgaver til kapittel 5

- **Fra boka:** 5.3

  5.11 Vi har tidligere sett på: A -> (A) | a

  5.18 Forsøk også sette alternativet A -> A A til slutt

- **Utvid** grammatikken på den foilen (i Kap 5, del 2) som ser på den flertydige grammatikken:

  E' -> E E -> E + E | E * E | n

  med høyreassosiativ opphøying slik:

  E' -> E E -> E + E | E * E | E ** E | n

  og avgjør hvordan konfliktene da skal løses.

- **Oppgave 2** fra Eksamen 2006 (se neste side).

Eksamen 2006, oppgave 2 (minus ett punkt)

Betrakt følgende grammatikk G, hvor S og T er ikke-terminaler, # og a er terminalsymboler, og S er startsymbolet.

$S \rightarrow T\ S$

$S \rightarrow T$

$T \rightarrow \#\ T$

$T \rightarrow a$

a) Finn First og Follow-mengdene til T og S (og la $ betegne 'end-of-file' som i boka).

b) Formulér med dine egne ord hvilke sekvenser av terminalsymboler du kan lage ut fra S'.

c) Avgjør om du kan lage et regulært uttrykk som uttrykker disse sekvensene av # og a som du

kan utlede fra S, og hvis svaret er 'ja', gi et slikt regulært uttrykk.

d) Innfør et nytt start-symbol S' → S og lag LR(0)-DFA-en for G rett fra denne grammatikken. Nummerér tilstandene.

f) Lag parseringstabellen for G ut fra den typen grammatikk den er.

g) Vis hvordan setningen: "a#a" vil bli parsert ved å skrive opp, som i boka, stakk-innholdet og input for hver av skift- eller reduser-operasjon

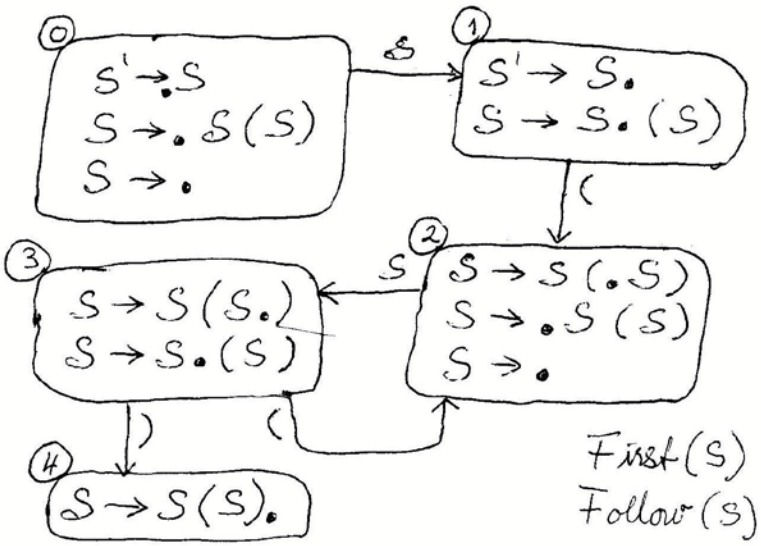
Fra boka: Oppgave 5.3

Kommentar: Denne er, i motsetning til den med S-> (S)S, ikke LL(1) (se kommentar i forbindelse med en av oppgavene til kap 4)

5.3 Consider the following grammar:

$$S \rightarrow S\ (\ S\ )\ |\ \varepsilon$$

a. Construct the DFA of LR(0) items for this grammar.
b. Construct the SLR(1) parsing table.
c. Show the parsing stack and the actions of an SLR(1) parser for the input string (()()).
d. Is this grammar an LR(0) grammar? If not, describe the LR(0) conflict. If so, construct the LR(0) parsing table, and describe how a parse might differ from an SLR(1) parse.



First(S) = {ε, (}
Follow(S) = {$, (,)}

	(	)	$	S	
0	r(S→ε)	r(S→ε)	r(S→ε)	1	accept!
1	s2		r(S'→S)		
2	r(S→ε)	r(S→ε)	r(S→ε)	3	
3	s2	s4			
4	r(S→S(S))	r(S→S(S))	r(S→S(S))		

SLR(1)-tabell

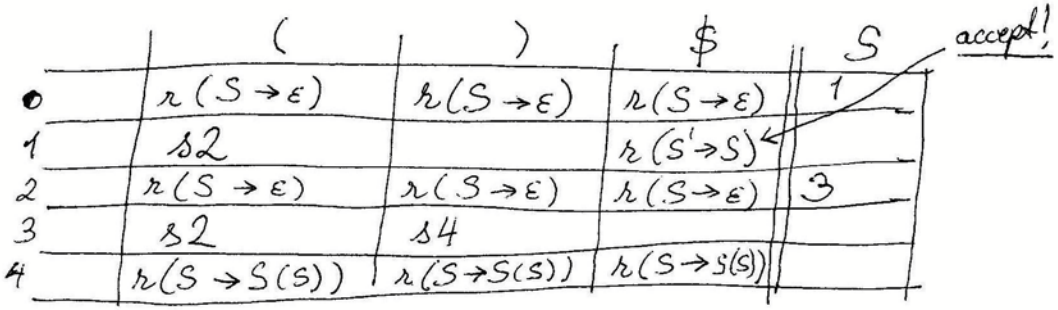
Den er ikke LR(0) på grunn av tilst. 1.

Merk at tilst. 0 og 2 ikke er problematiske siden det ikke er lovlig å skifte for noe terminalsymbol. Redusering er altså eneste mulighet

Den er SLR(1) fordi i tilstand 1 er red(S' -> S) bare aktuelt for "$", mens skift bare er aktuelt for "(".

Man kan også ekvivalent si at den er SLR(1) fordi tabellen ble entydig.

Oppgave 5.3, fortsatt



Grammatikk:

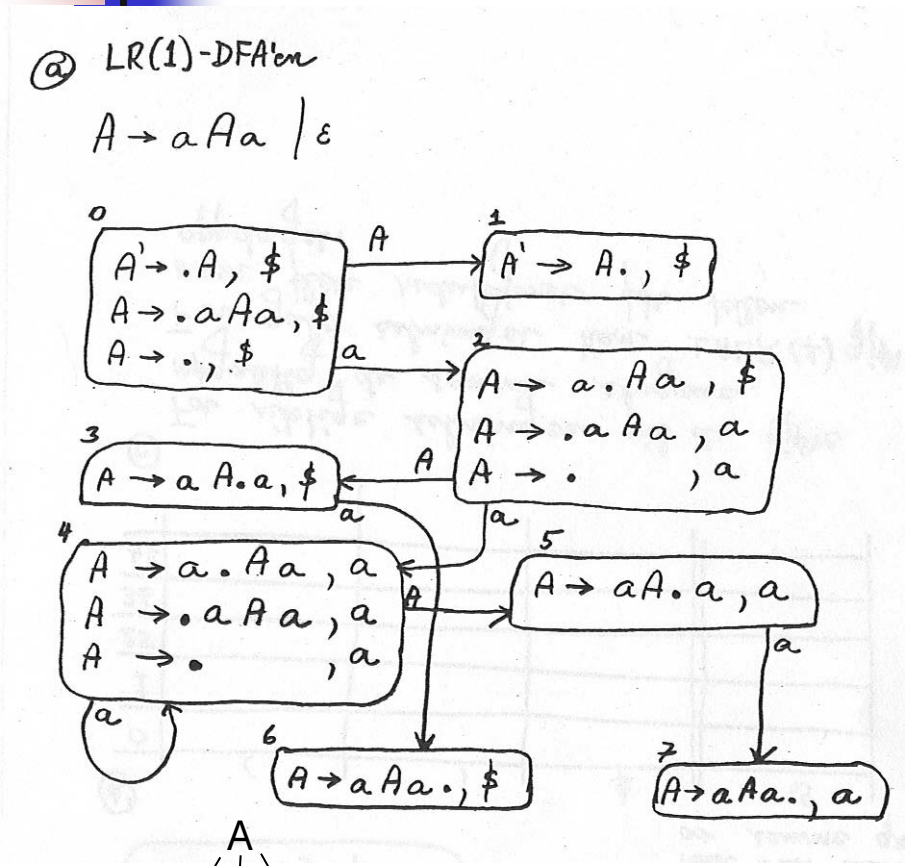
S -> S (S) | ε

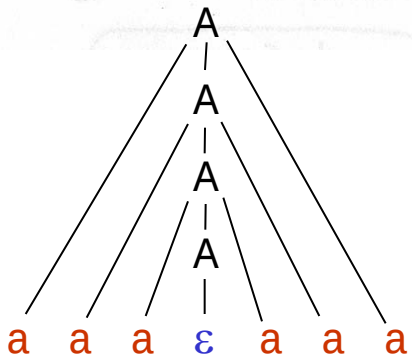
$0	(( ) ( )) $
$0 S 1	(( ) ( )) $
$0 S 1 (2	( ) ( )) $
$0 S 1 (2 S 3	( ) ( )) $
$0 S 1 (2 S 3 (2	) ( )) $
$0 S 1 (2 S 3 (2 S 3	) ( )) $
$0 S 1 (2 S 3 (2 S 3) 4	( )) $
$0 S 1 (2 S 3	( )) $
$0 S 1 (2 S 3 (2	)) $
$0 S 1 (2 S 3 (2 S 3	)) $
$0 S 1 (2 S 3 (2 S 3) 4	) $
$0 S 1 (2 S 3	) $
$0 S 1 (2 S 3) 4	$
$0 S 1	$

accept!

To røde streker under betyr at reduksjone med S -> ε *er utført*. En strek under betyr at det *skal* reduseres med det understrekede, mens piler betyr skift.

5.11 – a og b





For både tilstand 2 og 4 gjelder at på input 'a' så "foreslås" det både å skifte og å redusere med $A \rightarrow \varepsilon$. Altså er grammatikken ikke LR(1).

(b) Grammatikken genererer alle strenger med et partall antall 'a'-er, og den er entydig siden den bare kan gjøre dette på en måte (se figur).

Ekstra: Med denne grammatikken må vi imidlertid lese helt til slutten av setningen for å finne når vi skal gå over fra å skifte til å redusere. Det skal skje på midten.

Vi kan dermed se at grammatikken ikke er LR(k) for noen k.

Andre grammatikker som gir de samme setninger, og som helt kurant er SLR(1)

er: $A \rightarrow A\ a\ a \mid \varepsilon$

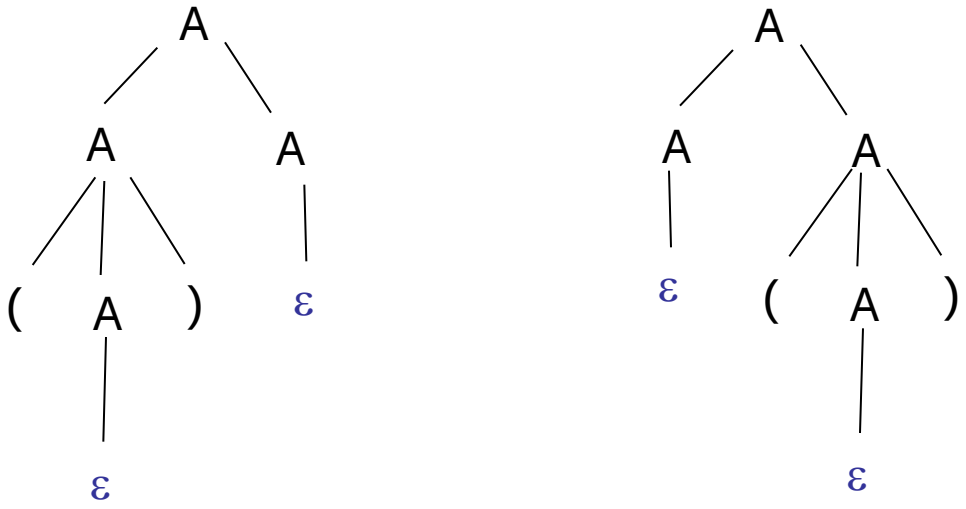
eller: $A \rightarrow a\ a\ A \mid \varepsilon$

Oppgave 5.18

Vi ser på grammatikken:
A -> A A | (A) | ε

Denne grammatikken genererer samme språk som de velkjente:
S -> (S) S | ε og S -> S (S) | ε
Nemlig: Alle korrekte parentesstrukturer.

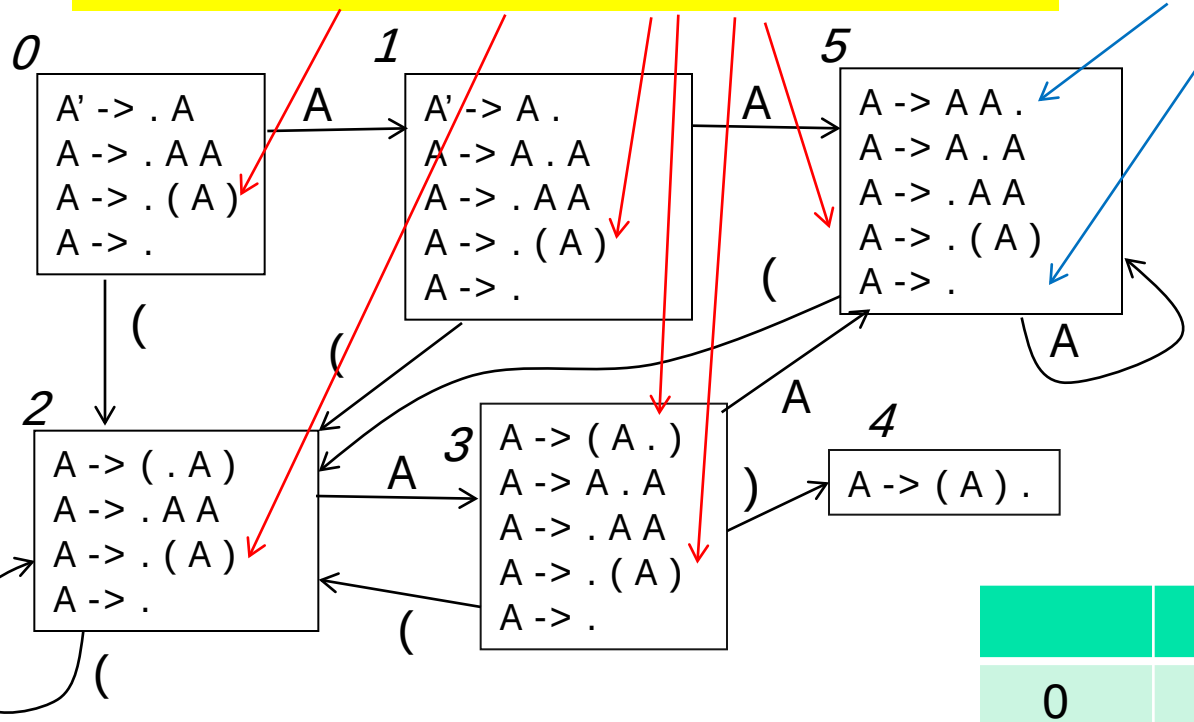
Men den nye er ”opplagt” flertydig! Vis det.
Svar: Her er to forskjellige trær for strengen: ” ( ) ”



Oppgave 5.18, side 2

Her kan man skifte for ”(”, og noen steder også for ”)” Velger alltid dette heller enn å redusere

Red./red.-konflikt.
YACC (og CUP?) velger den som står først, altså prod. (1)



Vi ser på grammatikken:

(1) A -> A A
(2) A -> (A)
(3) A -> ε

Follow(A) = { (,), $ }

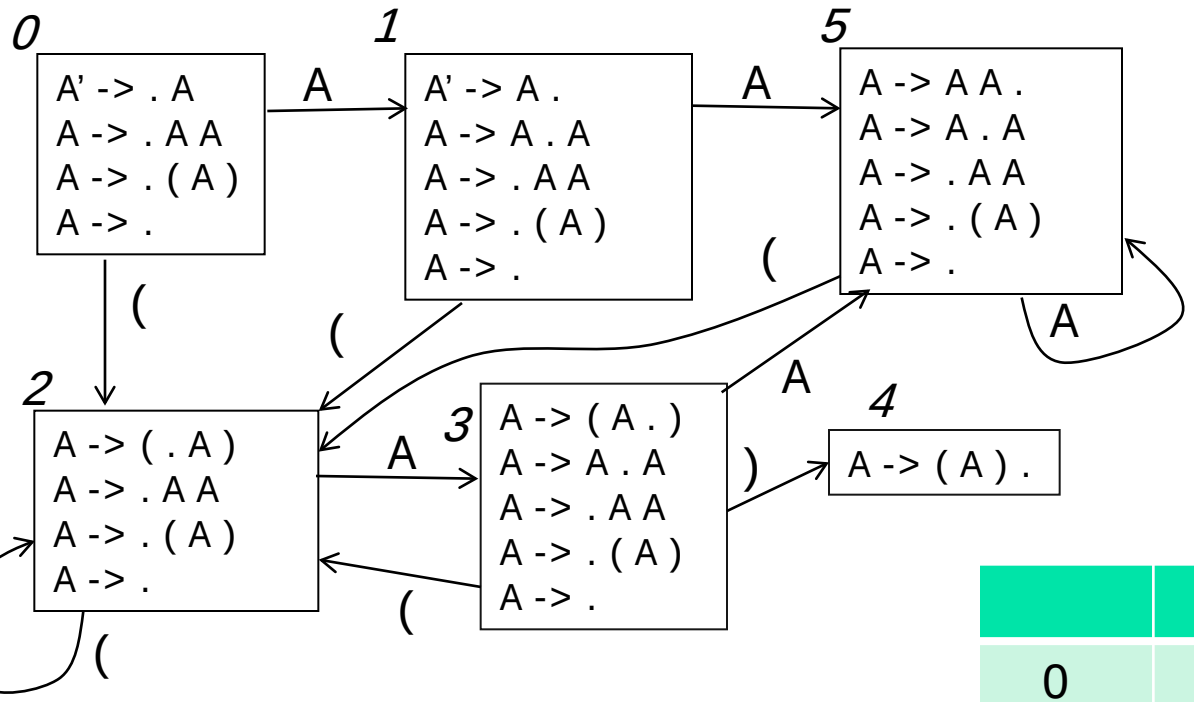
	(	)	$	A
0	s2	r(3)		1
1	s2		acc.	5
2	s2	r(3)	r(3)	3
3	s2	s4		5
4	r(2)	r(2)	r(2)	
5	s2	r(1)	r(1)	5

Velger å redusere om skift ikke er mulig. Det gir de fleste steder reduksjon med A -> ε. Om vi hadde byttet om prod. (1) og (3) ville vi også valgt å redusere med A -> ε for ”)” og ”$” i tilstand 5. Men, automaten ville da ikke virke siden den aldri ville redusere med A -> A A.

Oppgave 5.18, side 3:

Vil denne LR-tabellen godkjenne alle setninger i språket??

Vi så jo på forrige foil (nederst til venstre) at uheldige valg ødela automaten!



Vi ser på grammatikken:

(1) A -> A A

(2) A -> (A)

(3) A -> ε

Follow(A) = { (,), $ }

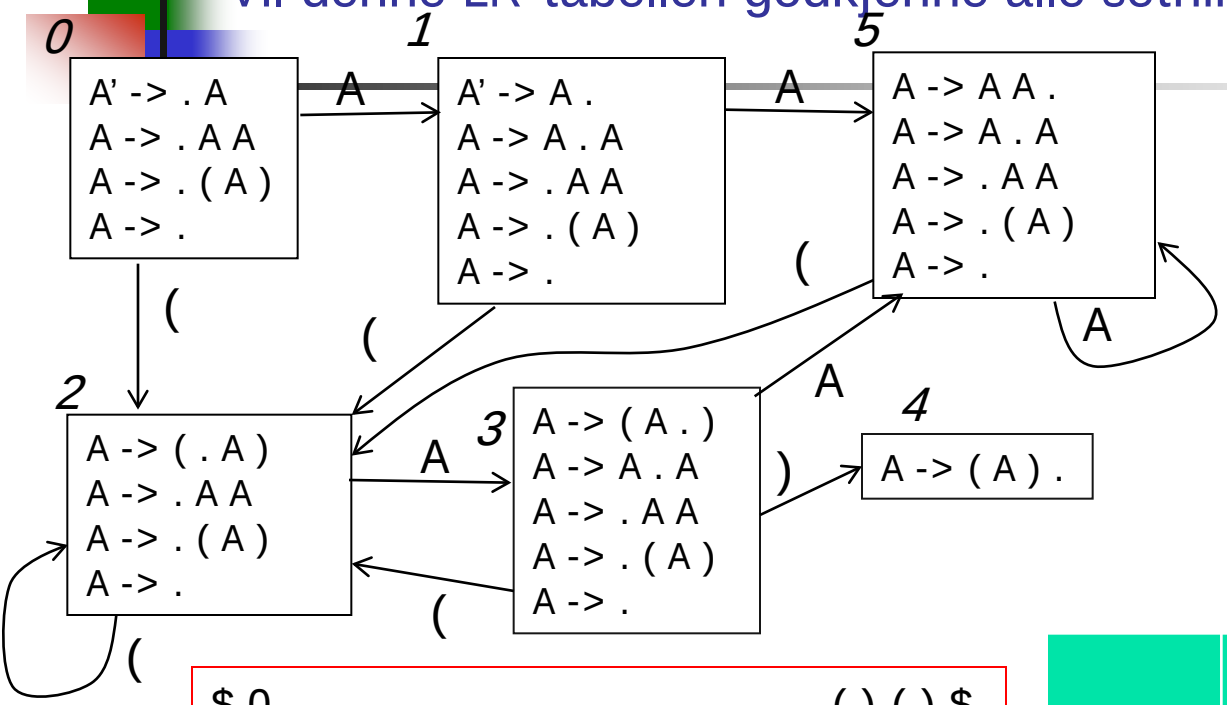
	(	)	$	A
0	s2	r(3)		1
1	s2		acc.	5
2	s2	r(3)	r(3)	3
3	s2	s4		5
4	r(2)	r(2)	r(2)	
5	s2	r(1)	r(1)	5

Vil denne tabellen godkjenne akkurat de samme setningen som den gitte flertydige grammatikken?

Har ingen metode for å avgjøre dette, og må stole på å se på å studere eksempler. Se neste side.

Oppgave 5.18, side 4:

Vil denne LR-tabellen godkjenne alle setninger i språket??



Vi ser på grammatikken:

(1) A -> A A
(2) A -> (A)
(3) A -> ε

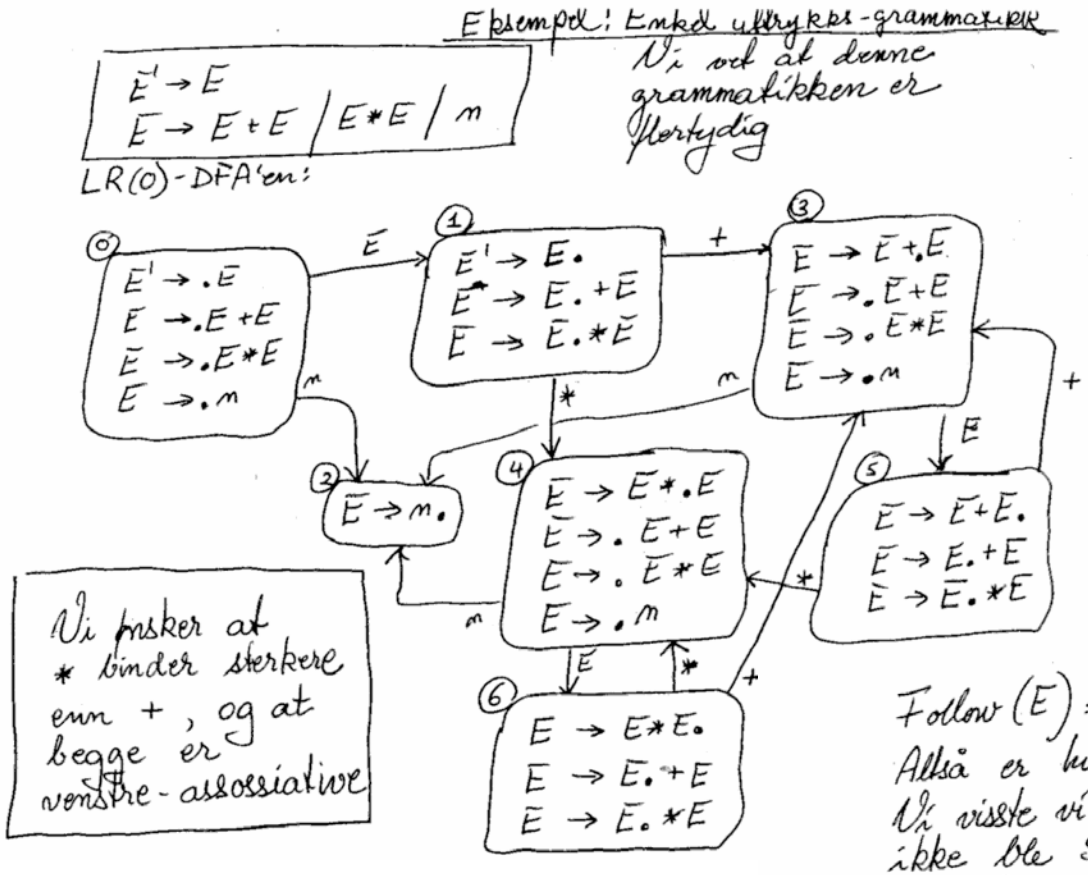
Follow(A) = { (,), $ }

	(	)	$	A
0	s2	r(3)		1
1	s2		acc.	5
2	s2	r(3)	r(3)	3
3	s2	s4		5
4	r(2)	r(2)	r(2)	
5	s2	r(1)	r(1)	5

$ 0	( ) ( ) $
$ 0 (2	) ( ) $
$ 0 (2 A 3	) ( ) $
$ 0 (2 A 3) 4	( ) $
$ 0 A 1	( ) $
$ 0 A 1 (2	) $
$ 0 A 1 (2 A 3	) $
$ 0 A 1 (2 A 3) 4	$
$ 0 A 1 A 5	$
$ 0 A 1	$

Det ser i hvert fall *lovende* ut mht. å klare alle setninger! Hva om vi brukte r(3) ved [5, $] ?

Gammel foil: Innfør her ** (høyeste presedens, og høyreassossitiv)



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 med å angi presedens, assosiativitet, m.m. Dette kan angis f.eks. 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 +

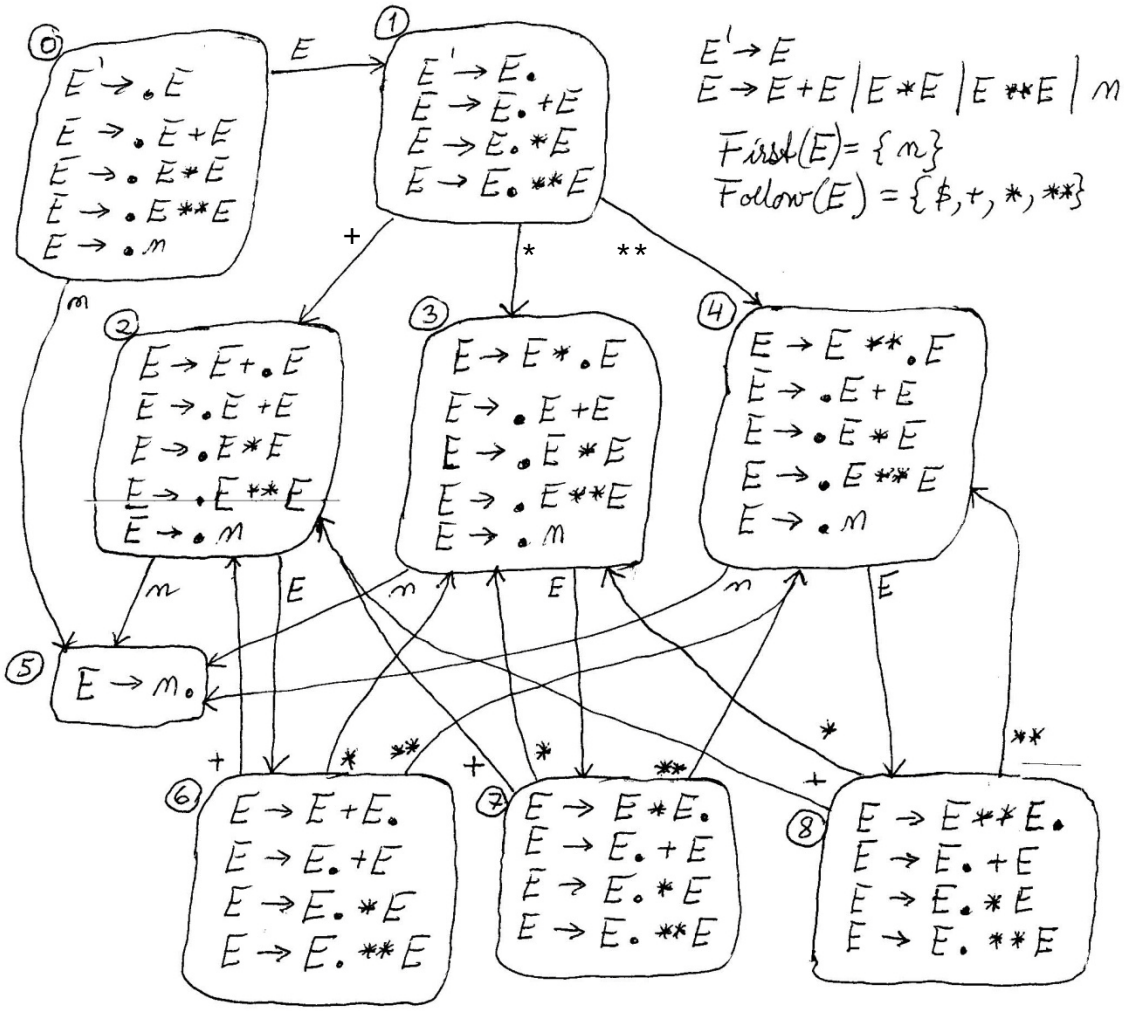
Tilstand 6: Stakk= .....E*E Input=
$: reduser, fordi skift ikke lovlig for $
+: reduser, fordi * har presedens over +
*: reduser, fordi * er venstreassossiativ)

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

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

	n	+	*	$	E
0	s2				1
1		s3	s4	accept	
2		r(E→n)	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)	

Innføring av høyre-assosiativ ** med høyeste presedens



Her måtte det bli konflikter, siden grammatikken er flertydig, og konfliktene opptrer i tilstandene 6, 7 og 8. Ut fra presedens og assosiativitet løser vi dette slik:

Tilstand 6 (E+E øverst på stakken)

\+ red(E-> E+E) (venstre-ass.)

* skift 3

** skift 4

Tilstand 7 (E*E øverst på stakken)

\+ red(E -> E*E)

* red(E -> E*E) (venstr-ass.)

** skift 4

Tilstand 8 (E**E øverst på stakken)

\+ red(E -> E**E)

* red(E -> E**E)

** skift 4 (høyre-ass.)

NB: Dette skjer altså automatisk i CUP når man har oppgitt presedens og assosiativitet riktig.

Eksamen 2006, oppgave 2 (minus ett punkt)

Betrakt følgende grammatikk G, hvor S og T er ikke-terminaler, # og a er terminalsymboler, og S er startsymbolet.

$S \rightarrow T\ S$

$S \rightarrow T$

$T \rightarrow \#\ T$

$T \rightarrow a$

a) Finn First og Follow-mengdene til T og S (og la $ betegne 'end-of-file' som i boka).

b) Formulér med dine egne ord hvilke sekvenser av terminalsymboler du kan lage ut fra S'.

c) Avgjør om du kan lage et regulært uttrykk som uttrykker disse sekvensene av # og a som du

kan utlede fra S, og hvis svaret er 'ja', gi et slikt regulært uttrykk.

d) Innfør et nytt start-symbol S' → S og lag LR(0)-DFA-en for G rett fra denne grammatikken. Nummerér tilstandene.

f) Lag parseringstabellen for G ut fra den typen grammatikk den er.

g) Vis hvordan setningen: "a#a" vil bli parsert ved å skrive opp, som i boka, stakk-innholdet og input for hver av skift- eller reduser-operasjon

Eksamen 2006, oppgave 2

2a

S → T S	Gjør at Fi(S) skal ha alle fra Fi(T), og at Fo(T) skal ha alle fra Fi(S)
S → T	Gjør, som over, at Fi(S) skal ha alle fra Fi(T), og at Fo(T) skal ha alle fra Fo(S)
T → # T	Gjør at Fi(T) skal inneholde #
T → a	Gjør at Fi(T) skal inneholde a

	First	Follow
S	**a #**	**$**
T	**a #**	**a # $**

2b

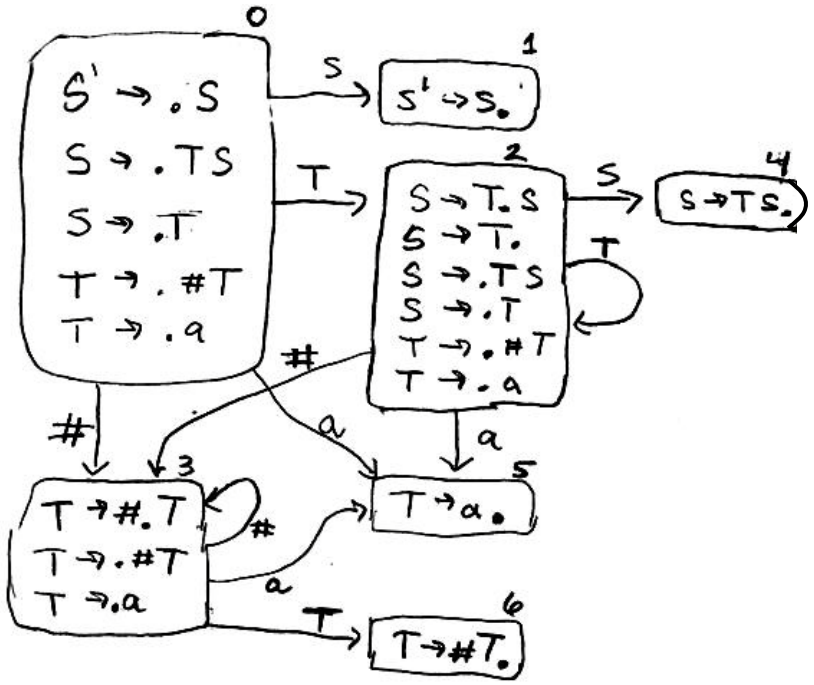
Vi kan fra S generere en-eller-flere ’a’-er hvor hver a har null eller flere # foran seg.

2c

Vi har flg. regulære uttrykk { {#}* a}+

Eksamen 2006, oppgave 2

2d LR(0)-DFA'en blir som følger:



S → T S
S → T
T → # T
T → a

Eksamen 2006, oppgave 2

2e

Den er ikke LR(0) fordi det ikke er entydig hvilken regel vi skal bruke etter f.eks å ha lest '#' (vi må se på nest symbol for å se hvilken produksjon vi skal nytte – kommer det en ny # eller 'a'?) Det er altså shift/red konflikt i tilstand 2.

Den er SLR(1) fordi den skift-reduser konflikten vi har i tilstand 2 løses ved at Followmengden til S ikke inneholder # eller a (dvs.: har vi # eller a som neste symbol, gjøres skift med overgang til 3 eller 5, mens er det $ gjør vi reduksjon med S -> T og går tilbake til 0 og så 1 og accept).

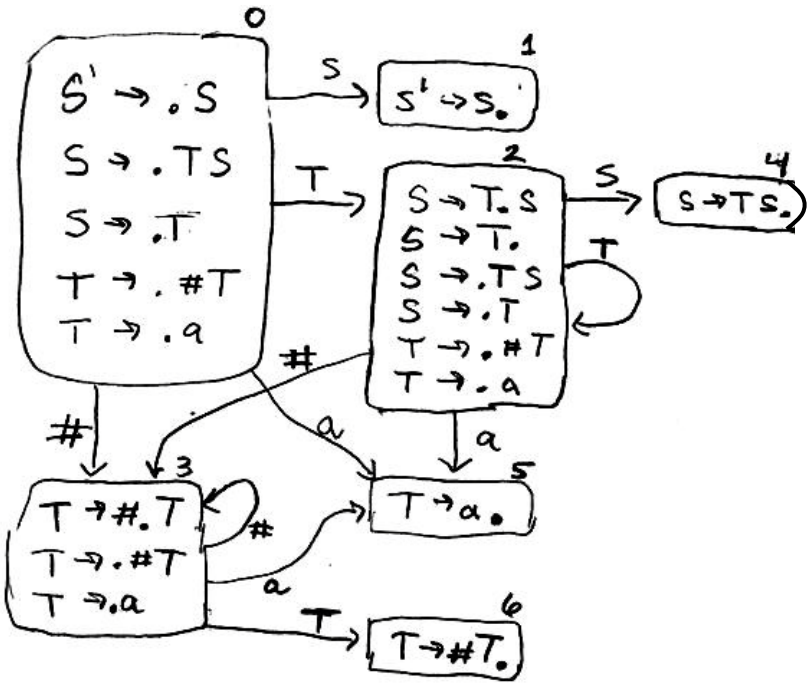
Siden den er SLR(1), er den også LALR(1) og LR(1). Den er følgelig også entydig.

2f

	a	#	$	S	T
0	s5	S3		1	2
1			accept		
2	s5	s3	r(S->T)	4	2
3	s5	s3			6
4			r(T->TS)		
5	r(T->a)	r(T->a)	r(T->a)		
6	r(T->#T)	r(T->#T)	r(T->#T)		

(Merk at i tilstand 2 har vi både skift og reduksjon, valgt ut fra look-ahead-symbolet)

	First	Follow
S	**a #**	**$**
T	**a #**	**a # $**



Eksamen 2006, oppgave 2

2g

	a	**#**	**$**	**S**	**T**
0	s5	S3		1	2
1			accept		
2	s5	s3	r(S->T)	4	2
3	s5	s3			6
4			r(T->TS)		
5	r(T->a)	r(T->a)	r(T->a)		
6	r(T->#T)	r(T->#T)	r(T->#T)		

Analyse av a # a :

$ 0	a # a $
$ 0 a 5	# a $
$ 0 T 2	# a $
$ 0 T 2 # 3	a $
$ 0 T 2 # 3 a 5	$
$0 T 2 # 3 T 6	$
$0 T 2 T 2	$
$ 0 T 2 S 4	$
$ 0 S 1	$
accept	