Med rettelser til oppgave 5.18, gjort 3/3

INF5110, 29/2-2012

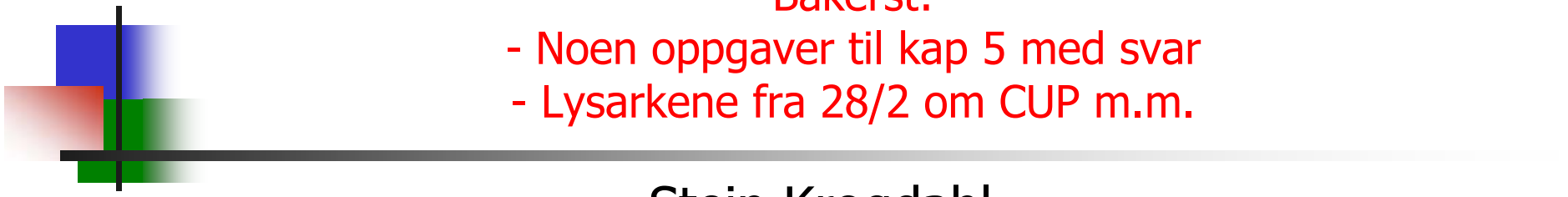
Her er også alt fra 28/2

Kap. 5, Del 3:

Litt om LR(1)- og LALR(1)-grammatikker

Bakerst:

- Noen oppgaver til kap 5 med svar
- Lysarkene fra 28/2 om CUP m.m.



Stein Krogdahl,

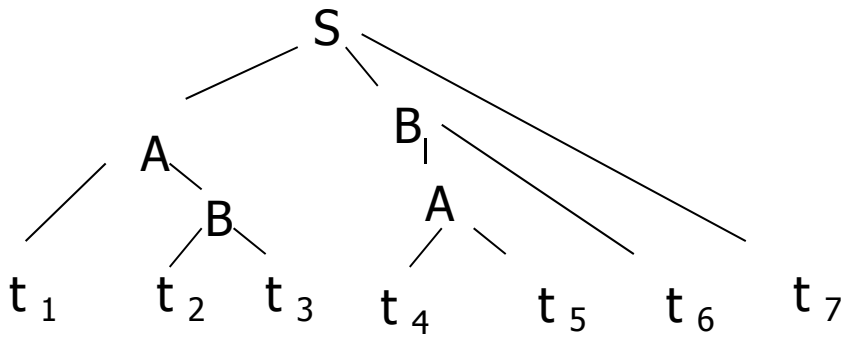
Ifi, UiO

I dag:

Første time: Om feilbehandling under LR-parsering, og flere oppgaver

Siste time: Mer om Oblig1

Typer av LR-parsering



LR-parsering 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 lookahead-mengder
- LR(1) Det teoretisk sterkeste, om man vil se på ett lookahead-symbol

Merk: - Alle metodene setter opp en tabell av samme type (selv om den for LR(0) kan gjøres litt enklere)
- De tre første får like store tabeller, mens den til LR(1) blir mye større
- Når tabellen først er satt opp er parserings-algoritmen akkurat den samme for alle

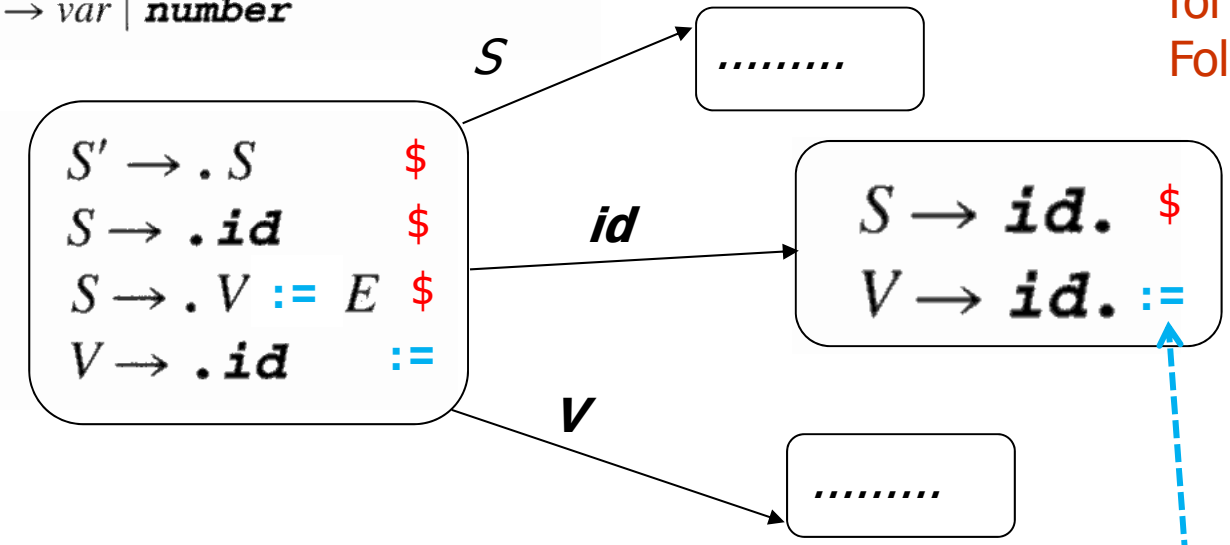
Grammatikk som ikke er SLR(1)

Men den *er* LALR(1)!

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

$S \rightarrow \mathbf{id} \mid V\ \mathbf{:=}\ E$
$V \rightarrow \mathbf{id}$
$E \rightarrow V \mid \mathbf{n}$

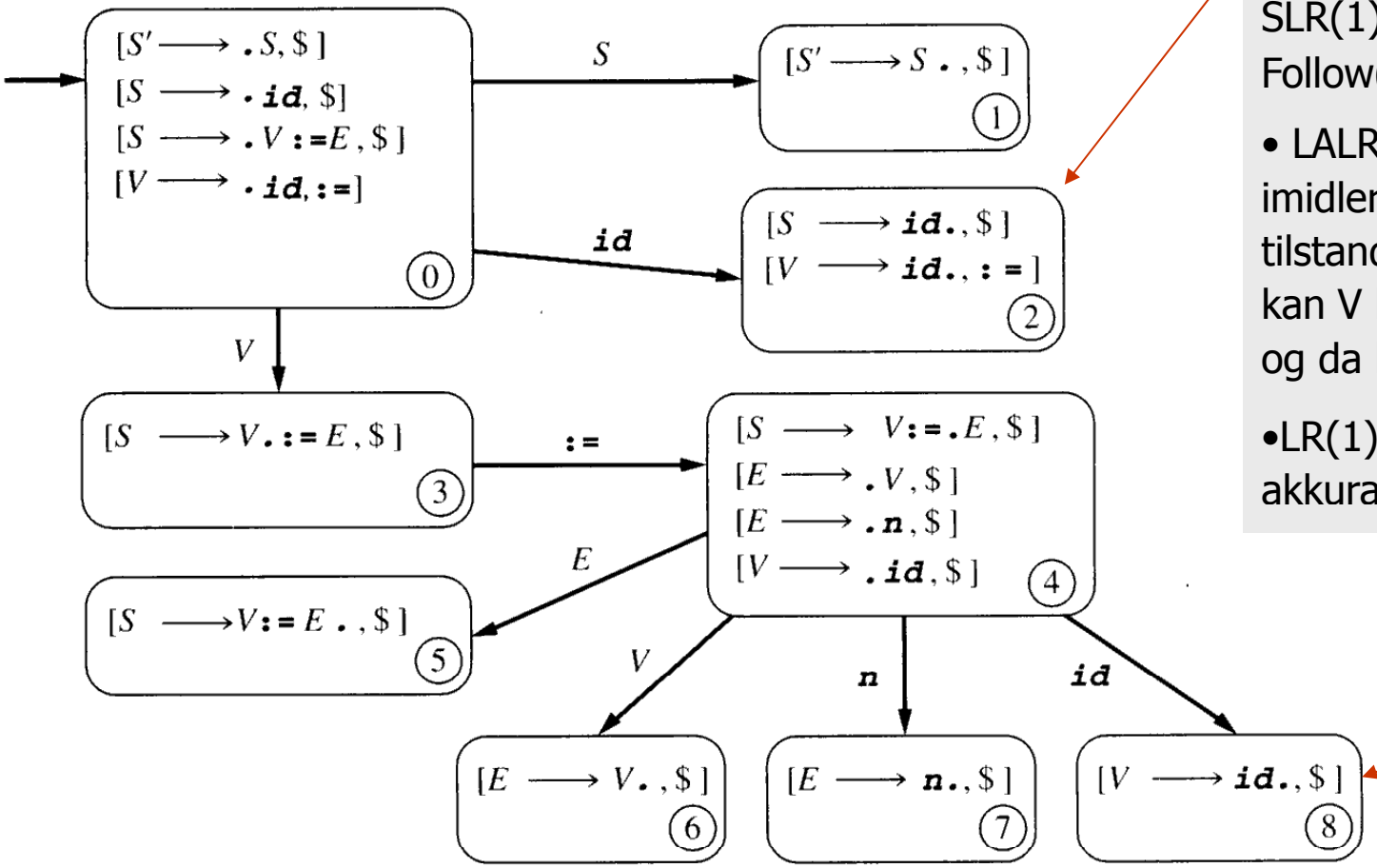
SLR(1): Gir her reduser/reduser-konflikt for input = $. Se First og Follow under.



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

- Men: Det viser seg at vi kan løse denne ved å være nøyere med "etterfølgermengder" *under konstruksjon* av DFA'en. Da får vi her bare := som etterfølge-mengde for V -> id. Over, og da er det ingen konflikt!
- Vi sier da at vi finner *LALR(1)-DFA'en*

LALR(1)-DFA (og LR(1)-DFA) for gram. på forrige side. Da løser det seg i tilstand 2, og i alle andre tilstander.



- Her var det konflikt ved SLR(1), siden Follow(V) = { :=, $}.
- LALR(1)-betraktning viser imidlertid at i denne tilstanden (sammenhengen) kan V bare følges av { :=}, og da har vi ikke problemer.
- LR(1)-betrakning gir akkurat den samme DFA'en

Her er situasjonen hvor V kan følges av $, men her er det ikke problematisk.

Grammatikken *er* altså LALR(1) (og dermed også LR(1)!)

Full LR(1)-parsering

(det beste vi kan få til med én "Lookahead")

Hovedidé: Vi vil, *under oppsett av NFA og DFA*, ha LR(1)-itemer som fra starten inneholder et etterfølger-symbol.

OG: Tilstander er bare like om alle itemene også har samme etterfølger-symbol. Derfor MANGE tilstander.

a = "*look-ahead*"

LR(1)-itemer:

[A →α . β , **a**]

Angir at **a** kan komme etter A→αβ i den aktuelle sammenhengen. Altså at **a** kan komme bak *hele* αβ, (og *ikke* bak punktumet, med mindre β = ε)

Definition of LR(1) transitions (part 1). Given an LR(1) item $[A \to \alpha . X \gamma, a]$, where X is any symbol (terminal or nonterminal), there is a transition on X to the item $[A \to \alpha X . \gamma, a]$.

NFA: [A →α . **X**γ, **a**] —**X**→ [A →α**X** . γ, **a**]

Vi flytter oss ett hakk framover i høyresiden, men produksjonen forblir i samme sammenheng

Definition of LR(1) transitions (part 2). Given an LR(1) item $[A \to \alpha . B \gamma, a]$, where B is a nonterminal, there are ε-transitions to items $[B \to .\beta, b]$ for every production $B \to \beta$ and *for every token* b in First(γa).

NFA: [A →α . **B**γ, **a**] —ε→ [B → . β, **b**]

For alle
B →β1 | β2 | ..
og alle
b ∈ First(γa)

Spesialtilfelle, inkludert i det over (γ = ε):

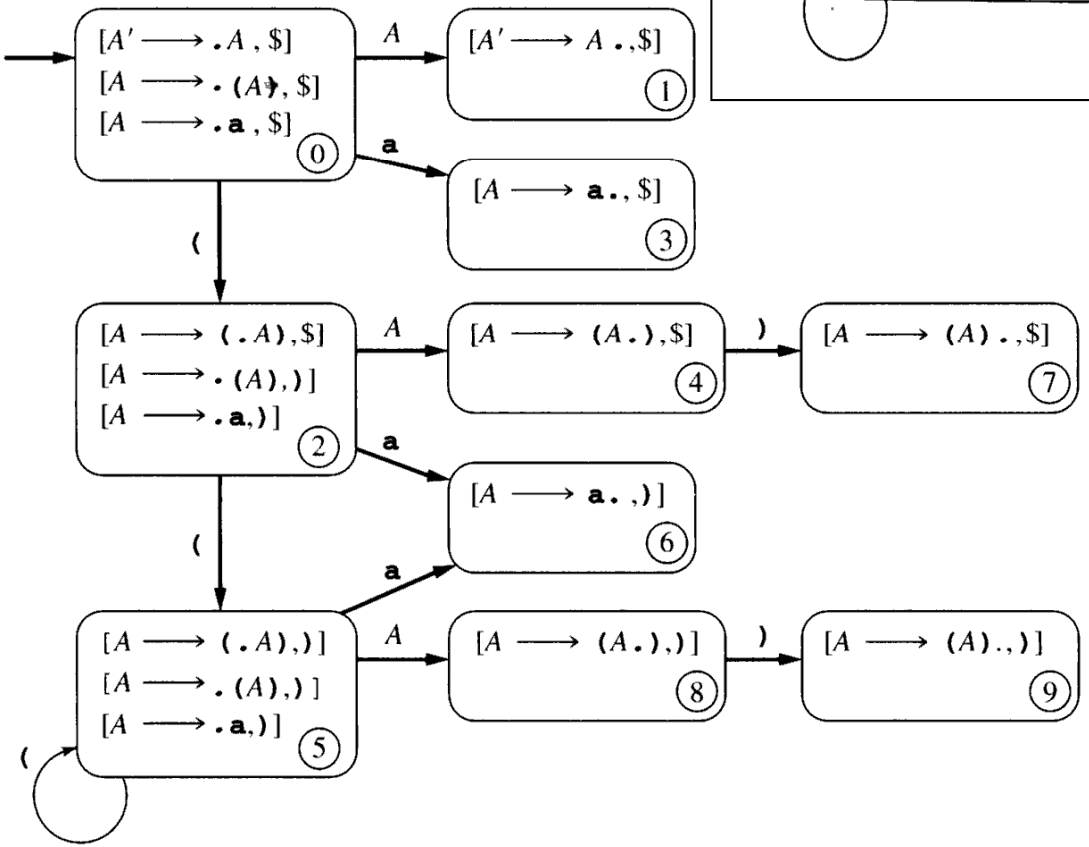
[A →α . **B**, **a**] —ε→ [B → . β, **a**]

For alle B → β1 | β2 | ..

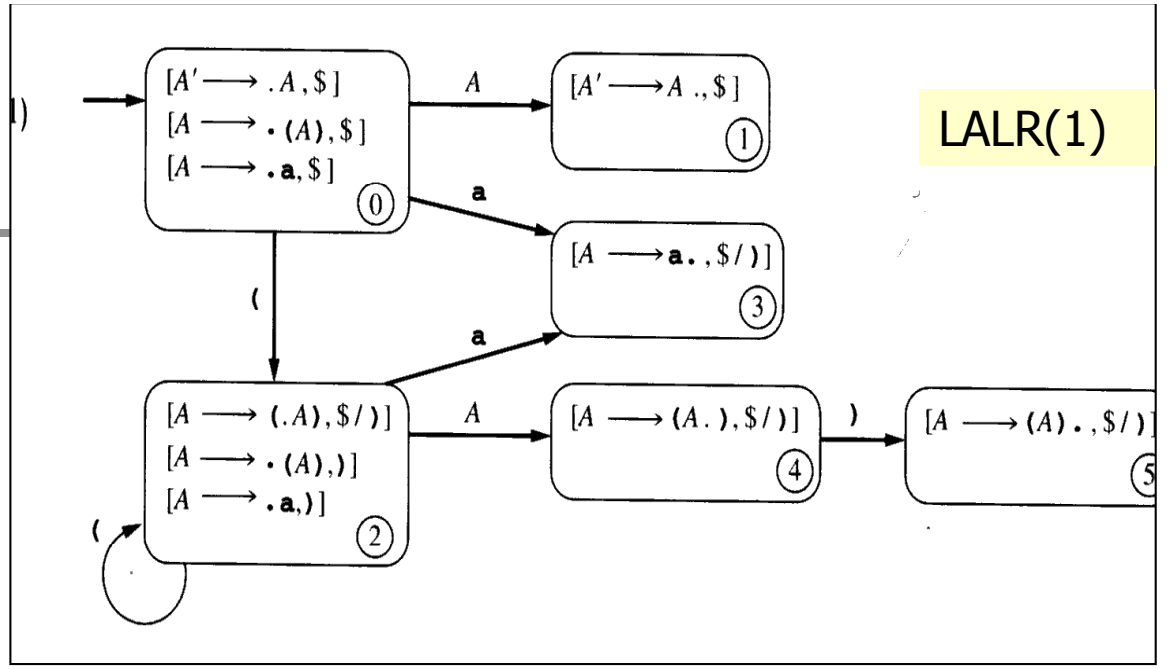
Sammenligning av LR(1)- og LALR(1)-DFA'er for samme grammatikk:

$$A \rightarrow (\; A \;) \mid \mathbf{a}$$

LR(1)



LALR(1)



Oppsett av LALR(1)-DFA'en ut fra LR(1)-DFA'en

Brukes ikke i praksis!

- "Core" (kjerne) av en LR(1)-tilstand:
  - Mengden av LR(0)-itemer, når man ser bort fra "look-ahead" itemene.
  - Merk at vi kan ha både **[A** $\rightarrow\alpha.\beta$**, a]** og **[A** $\rightarrow\alpha.\beta$**, b]** . Dette gir bare ett LR(0)-item i kjernen, nemlig : **A** $\rightarrow\alpha.\beta$

- Observasjoner:
  - Kjernen i alle LR(1)-DFA-tilstander er en LR(0)-DFA-tilstand (for samme grammatikken)
  - To LR(1)-tilstander med samme kjerne har samme kanter ut, og tilsvarende ut-kanter fører til tilstander med samme kjerne.

- LALR(1)-DFA'en:
  - I LR(1)-DFA'en slår vi sammen alle tilstander med samme kjerne.
  - Ut fra observasjonen 2 over får vi da også konsistente kanter mellom disse tilstandene.
  - Vi sitter rett og slett med LR(0)-DFA'en
  - Men med det tillegg at det etter hvert item er satt på lookahead-symboler som er *unionen av* det som var i tilstandene som ble slått sammen.
  - Det *kan* da altså hende at lookahead-mengden i et slutt-item **[A** $\rightarrow\alpha.$ **, a b c ...]** i LALR(1)-DFA'en er mindre enn etterfølgermengden til **A**, og dette gir større muligheter til å løse konflikter enn ved SLR(1)-betrakninger.

Oppsummering om LR-parsering (bottom-up)

OG: Det som står her om LALR(1) og LR(1) er det man skal vite om dette.

- Vi formulerer vår grammatikk som basal BNF
- Konflikter (f.eks. pga. flertydighet) kan løses med:
  - omredigere BNF'en (dog slik at den produserer samme språk!)
  - eller ved direktiver til CUP/Yacc/Bison (assosiativitet, presedens, etc.)
  - eller løse det senere i semantisk analyse (er: "(a and b) + 3" en semantisk feil?)
  - NB: Ikke **alle** konflikter *kan* løses selv av LR(1) – skriv om! (eks: A → a | a A a)
- De forskjellige varianter av LR-grammatikker, den ene sterkere enn den andre:

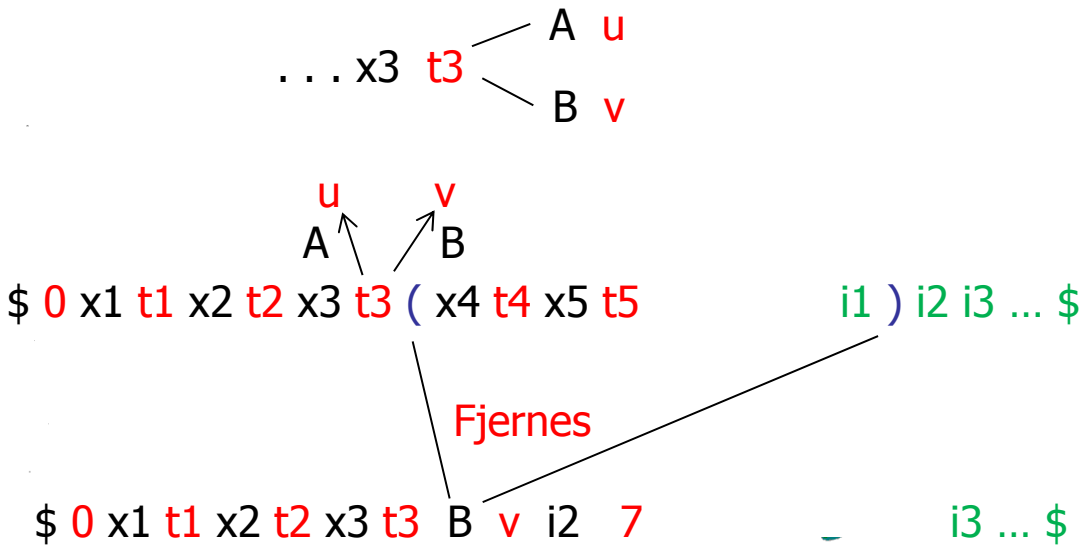
	Fordeler	**Annet**
LR(0)	Definerer DFA-tilstander som brukes av SLR og LALR	Mange (unødige) konflikter (red/red og skift/red) oppstår. Brukes neppe.
SLR(1)	Klar forbedring av LR(0), selv om den bare bruker det samme antall tilstander. Tabell typisk 50K felter	Ikke så god som LALR(1), men OK til det meste. Grei om man vil hånd-lage parser for liten grammatikk
LALR(1)	Nesten like bra som LR(1), men antall tilstander bare som for LR(0)	Brukes i de aller fleste automatiserte LR-parsere
LR(1)	Klarer alle grammatikker som teoretisk lar seg analysere ved én lookahead.	Svært mange tilstander (typisk tabell opp mot 1M felter for et reelt språk). Brukes?

Husk: Når tabellen først er satt opp, er algoritmen for parsering alltid den samme

Ved syntaksfeil:

"Panic-mode" for LR-parsering (det eneste vi skal se på)

Syntaksfeil:
fordi ruten $[t_5, i_1]$ er tom



		i1		i2			A		B
t3							u		v
t4		-					-		-
t5		-					-		-
u		-		r..					
v		-		s7					

1. Pop states from the parsing stack until a state is found with nonempty Goto entries. — finner t3
2. If there is a legal action on the current input token (i_1) from one of the Goto states, push that state onto the stack and restart the parse. If there are several such states, prefer a shift to a reduce. Among the reduce actions, prefer one whose associated nonterminal is least general. — (u og v)
3. If there is no legal action on the current input token from one of the Goto states, advance the input until there is a legal action or the end of the input is reached.

Velger til slutt å pushe på B, som etter å ha fjernet i1 gir tilstand v

Ta vekk én og én input, og gjenta 2

Eksempel: if-setning er mindre generell enn setning

Typisk Yacc-produsert parseringstabell

(merk påfyll av ekstra reduksjoner, som en plass-optimalisering i Yacc)

Grammatikk:

command → exp

exp → exp + term | exp – term | term

term → term * factor | factor

factor → NUMBER | (exp)

State	Input							Goto			
	NUMBER	(	+	-	*	)	$	*command*	*exp*	*term*	*factor*
0	s5	s6						1	2	3	4
1							accept				
2	r1	r1	s7	s8	r1	r1	r1				
3	r4	r4	r4	r4	s9	r4	r4				
4	r6	r6	r6	r6	r6	r6	r6				
5	r7	r7	r7	r7	r7	r7	r7				
6	s5	s6							10	3	4
7	s5	s6								11	4
8	s5	s6								12	4
9	s5	s6									13
10			s7	s8		s14					
11	r2	r2	r2	r2	s9	r2	r2				
12	r3	r3	r3	r3	s9	r3	r3				
13	r5	r5	r5	r5	r5	r5	r5				
14	r8	r8	r8	r8	r8	r8	r8				

Panic-mode for LR-parsering kan gå i evig løkke

Vi bruker følgende grammatikk:

command → exp
exp → exp + term | exp – term | term
*term → term * factor | factor*
factor → NUMBER | (exp)

samt parseringstabellen som Yacc produserte for denne (tidligere lysark)

Parsering med feil input "(n n)":

$ 0	(n n) $
$ 0 (6	n n) $
$ 0 (6 n 5	n) $
$ 0 (6 F 4	n) $
$ 0 (6 T 3	n) $
$ 0 (6 E 10	n) $
$ 0 (6 F 4	n) $

... gjentar seg selv

Feil, siden [10, n] er tomt. Videre:

- 10 har ingen goto, så E 10 poppes av
- 6 har goto for E T F
- ville gå til tilstand 10 3 4
- ved input n gir dette - r4 r6

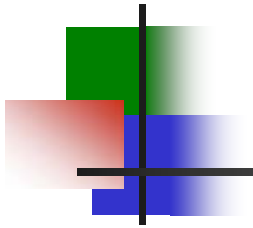
(ingen skift, dessverre!)

- Av T og F velger vi F, som er den minst generelle, og pusher derfor på F 4

Men da er vi tilbake til en tilstand vi har vært i (uten å ha lest noe input), og ting vil da bare gjenta seg selv.

Mulig løsning (meget løslig):

- Hold greie på om du kommer tilbake til samme tilstand, og gjør noe spesielt:
- Ta da mer bort fra stakken, og forsøk igjen
- Kanskje: *Forlange* en skift-mulighet for å sette i gang igjen



Oppgaver til INF 5110, kapittel 5

Gjennomgås 28. og eller 29. februar 2012

- **Fra boka:** 5.3 Vi har sett litt på denne på en forelesning
  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 ikketerminal-symboler, # 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 $S' \rightarrow S$ 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

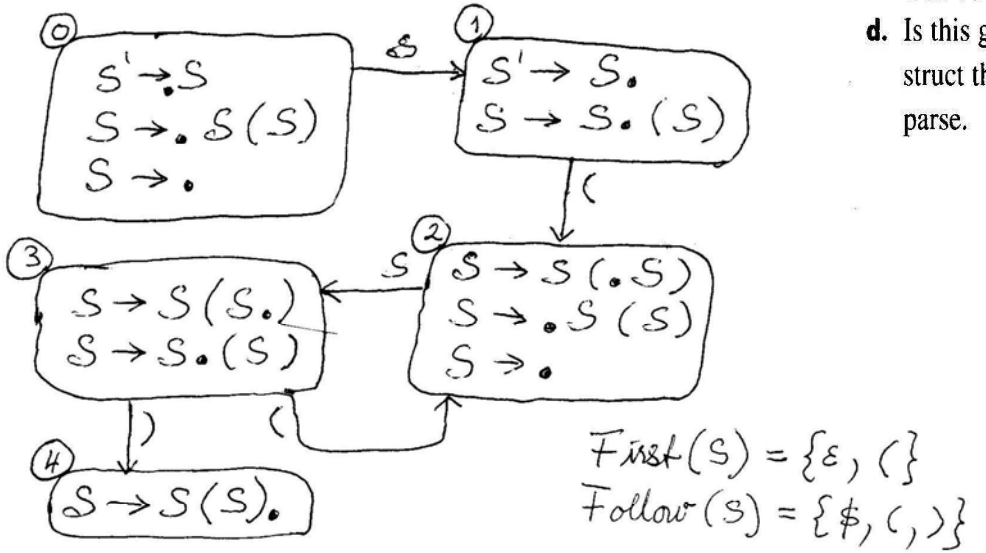
Fra boka: Oppgave 5.3

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
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))	

accept!

SLR(1)-tabell

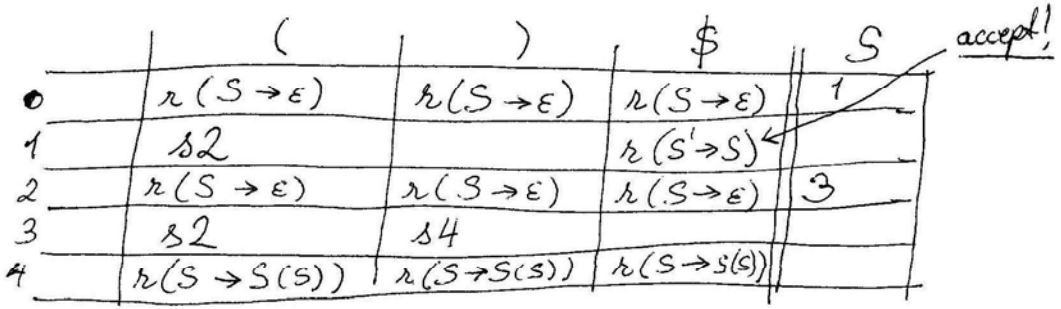
Den er ikke LR(0) på grunn av tilst. 1.

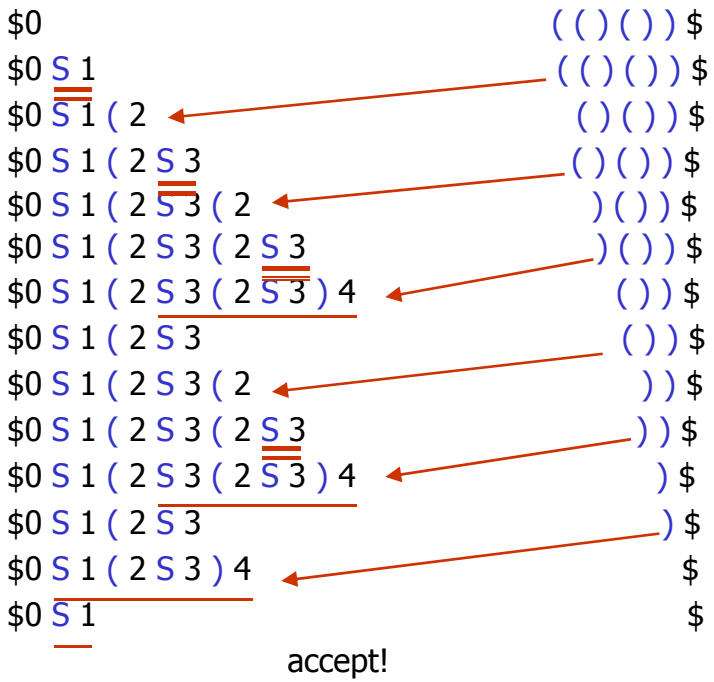
Merk at tilst. 0 og 2 ikke er problematiske siden det ikke er lovlig å skifte for noe terminalsymbol. Redusering er altså eneste mulighet

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.

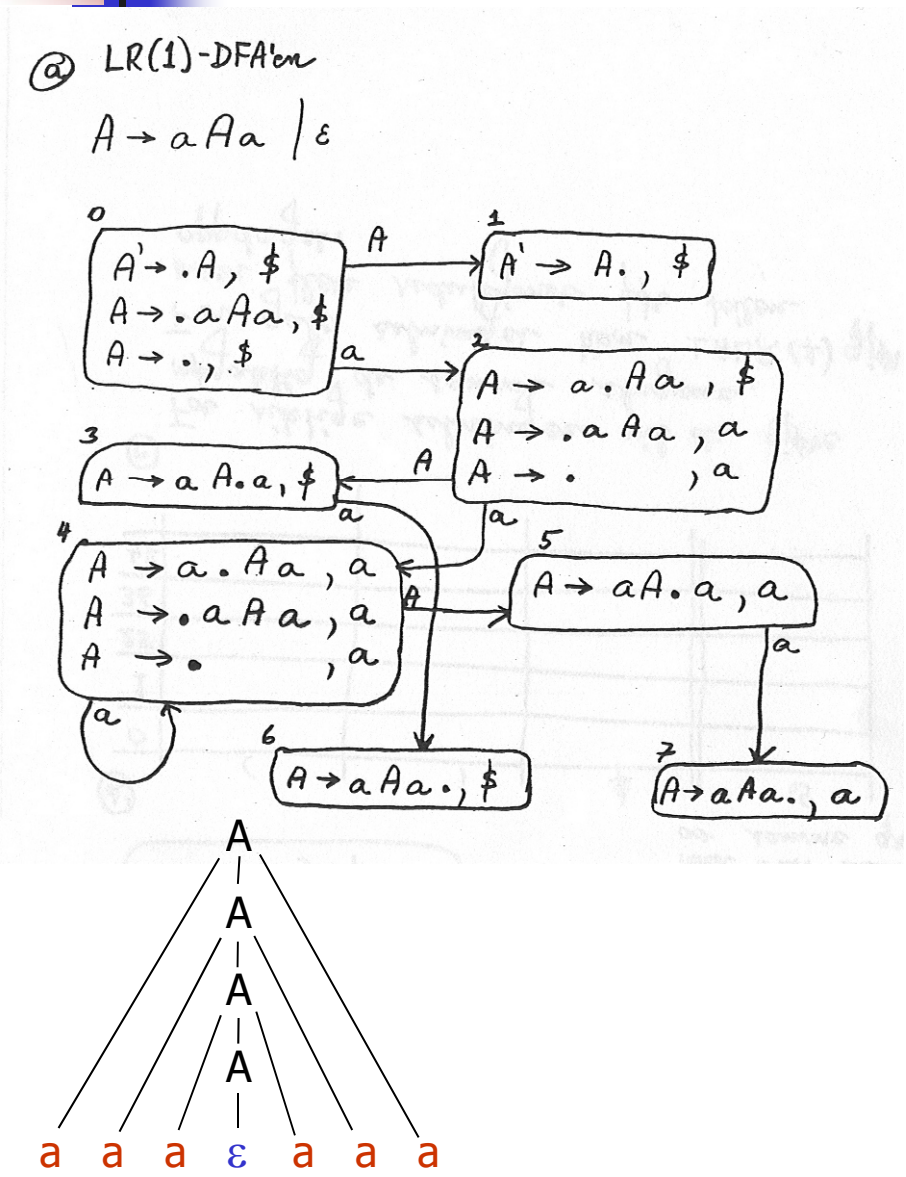
Oppgave 5.3, fortsatt





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 $A \to \varepsilon$. 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: $A \to A\ a\ a \mid \varepsilon$

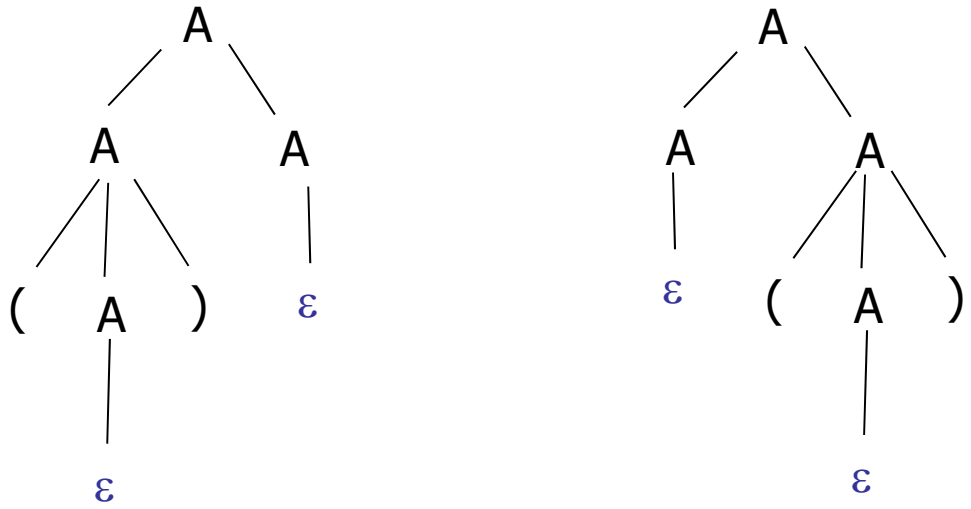
eller: $A \to a\ a\ A \mid \varepsilon$

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.

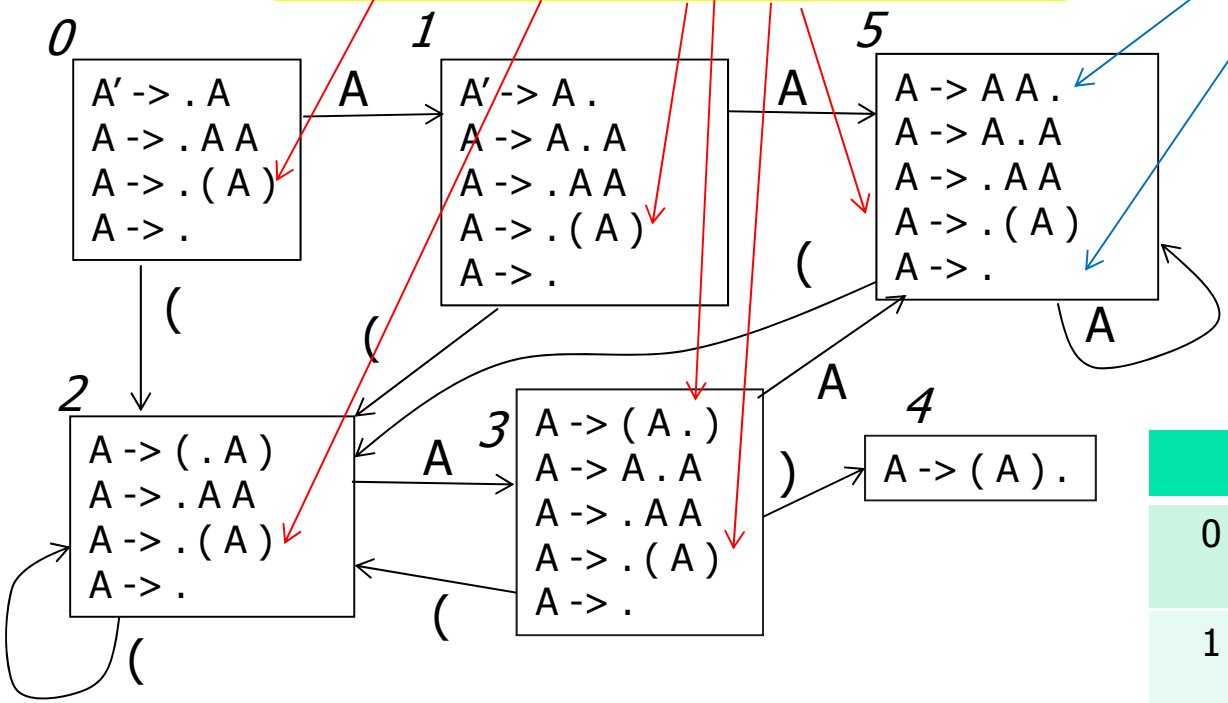
Men denne er ”opplagt” flertydig! Vis det.
Svar: Her er to forskjellige trær for strengen: ” ( ) ”



Oppgave 5.18, side 2

Her kan man skifte for ”(” eller ”)”
Velger alltid det heller enn å redusere

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



Vi ser på grammatikken:

(0) A' -> A
(1) A -> A A
(2) A -> (A)
(3) A -> ε

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

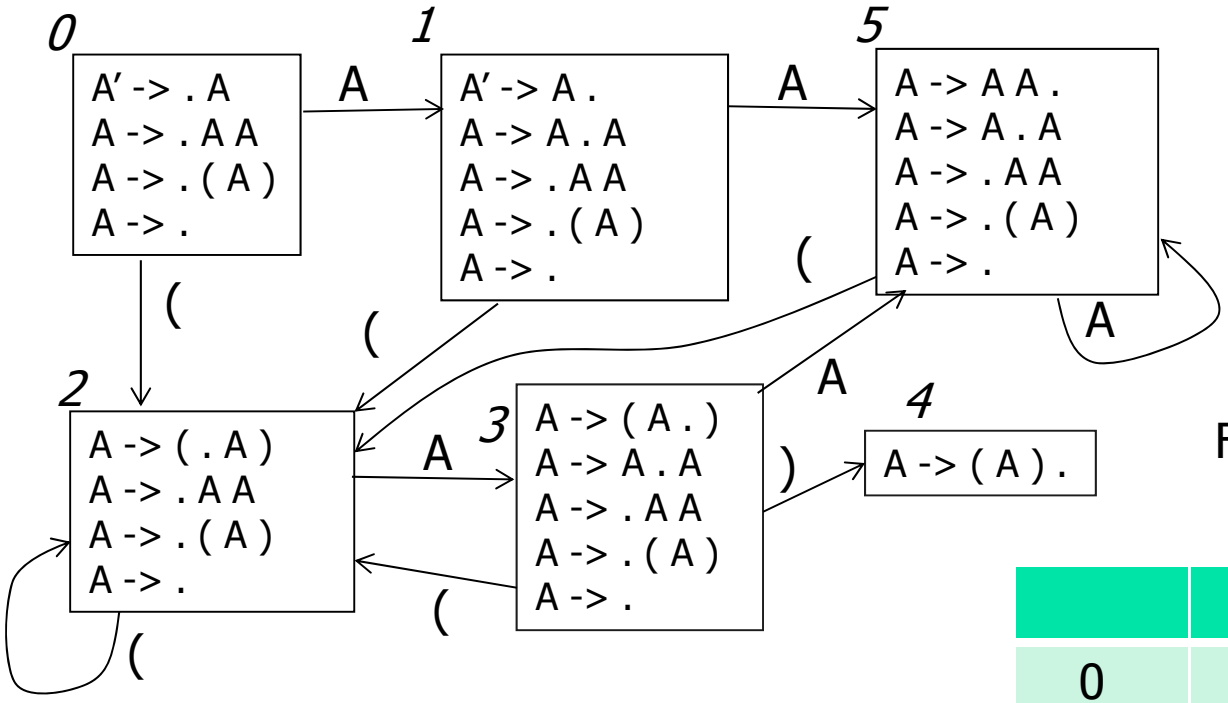
	(	)	$	A
0	r(3) s2	r(3)	r(3)	1
1	r(3) s2	r(3)	r(3) acc.	5
2	r(3) S2	r(3)	r(3)	3
3	r(3) s2	r(3) s4	r(3)	5
4	r(2)	r(2)	r(2)	
5	r(3) r(1) s2	r(3) r(1)	r(3) r(1)	5

Til høyre ser vi at SLR-tabellen blir flertydig i mange av rutene, og det måtte vi vente siden grammatikken er flertydig. CUP, YACC, etc. velger imidlertid å skifte der det er mulig, og ruten [1,$] blir nok også spesialbehandlet til ”accept” (altså r(0)). I rutene [5,)] og [5,$] blir r(1) valgt fremfor r(3) siden (1) står før (3) i grammatikken. Vi får da den entydige tabellen på neste side.

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:

(0) A' -> A
(1) A -> A A
(2) A -> (A)
(3) A -> ε

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

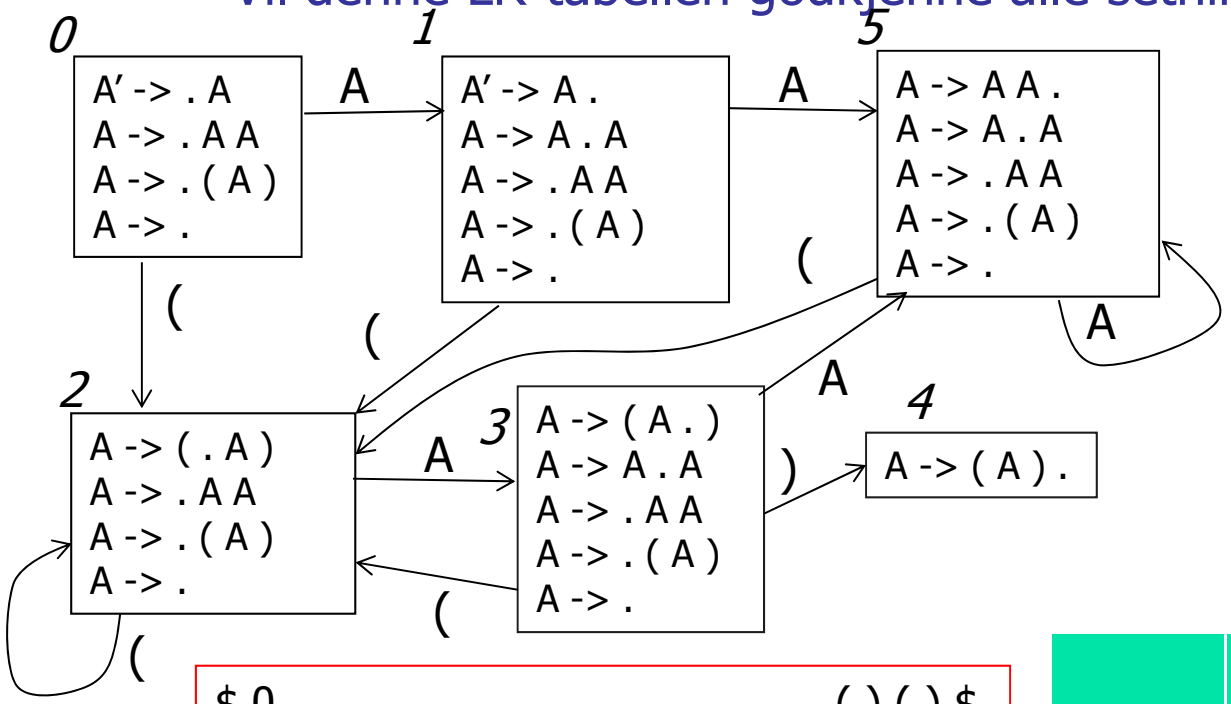
	(	)	$	A
0	s2	r(3)	r(3)	1
1	s2	r(3)	acc.	5
2	s2	r(3)	r(3)	3
3	s2	s4	r(3)	5
4	r(2)	r(2)	r(2)	
5	s2	r(1)	r(1)	5

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 studere eksempler. Se neste side.

Oppgave 5.18, side 4:

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



Vi ser på grammatikken:

(0) A' -> A
(1) A -> A A
(2) A -> (A)
(3) A -> ε

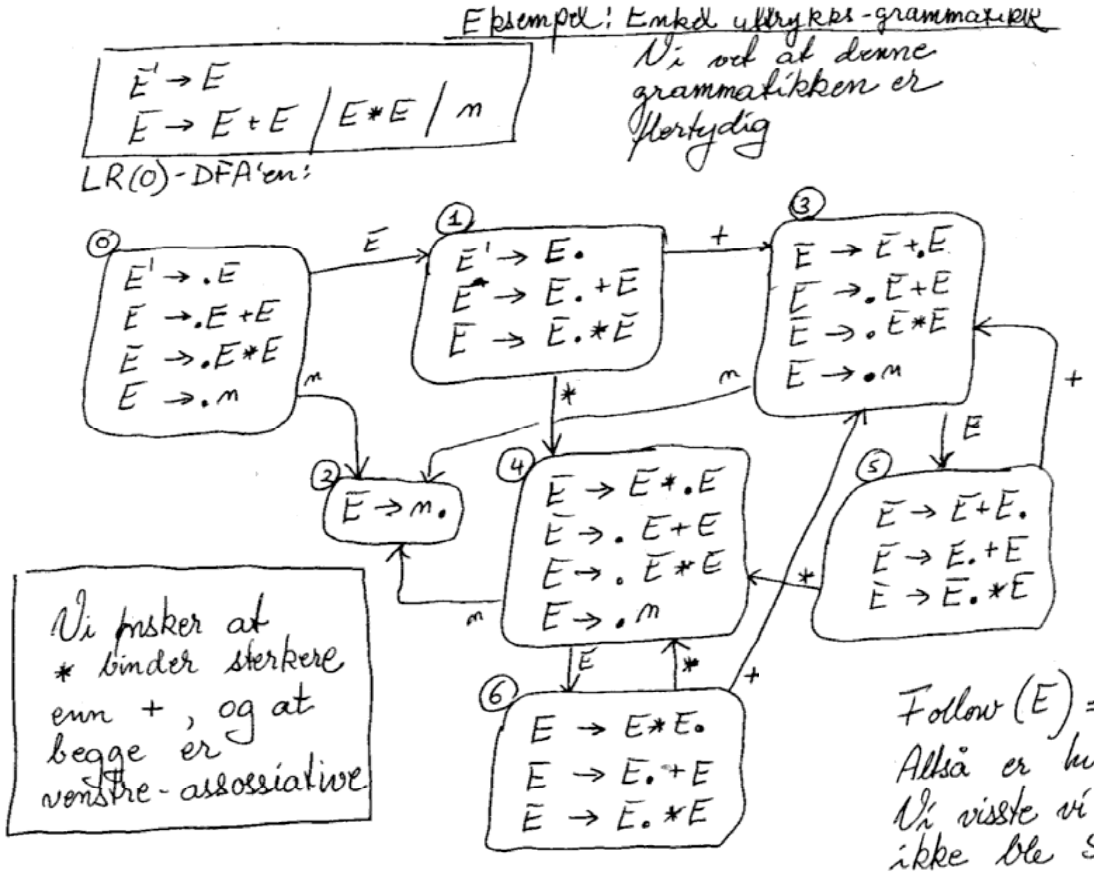
Follow(A) = { (,), $ }

$ 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	$

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

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 +

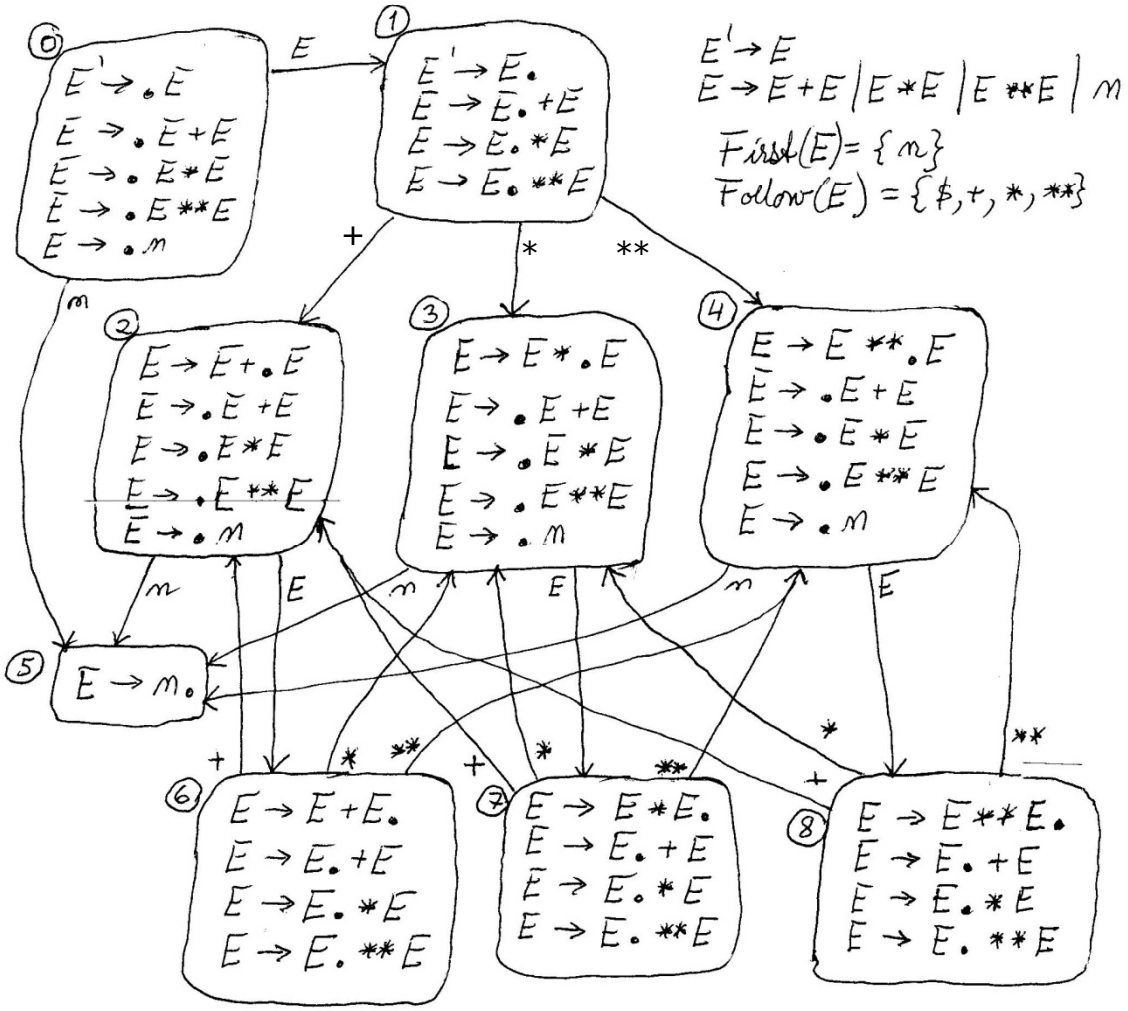
Tilstand 6: Stakk= ……E*E Input=
$: reduser, fordi skift ikke lovlig for $
+: reduser, fordi * har presedens over +
*: reduser, fordi * er venstreassossiativ)

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 – siden ingen flertydige grammatikker er SLR.
Hva skal vi så gjøre i tilstand ⑤ og ⑥?

	m	+	*	$	E
0	s2				1
1		s3	s4	accept	
2		r(E→m)	r(E→m)		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
(se undervisningsplanen 2008).

Betrakt følgende grammatikk G, hvor S og T er ikketerminal-symboler, # 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 $S' \rightarrow S$ og lag LR(0)-DFA-en for G rett fra denne grammatikken. Nummerér tilstandene.

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

f) 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

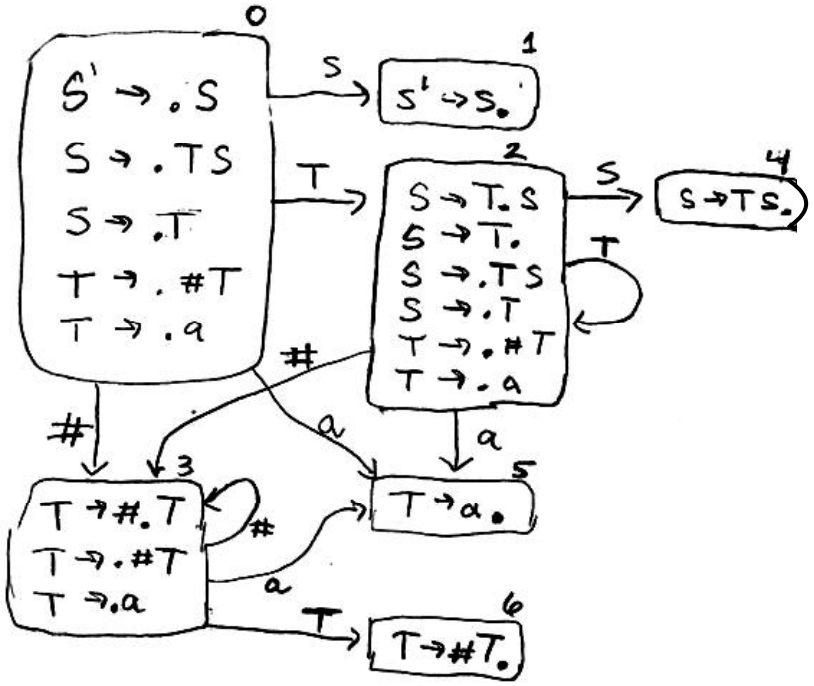
Vi kan fra S generere en-eller-flere 'a'-er hvor hver a har null eller flere # foran seg.

2c

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

Eksamen 2006, oppgave 2

2d LR(0)-DFA'en blir som følger:



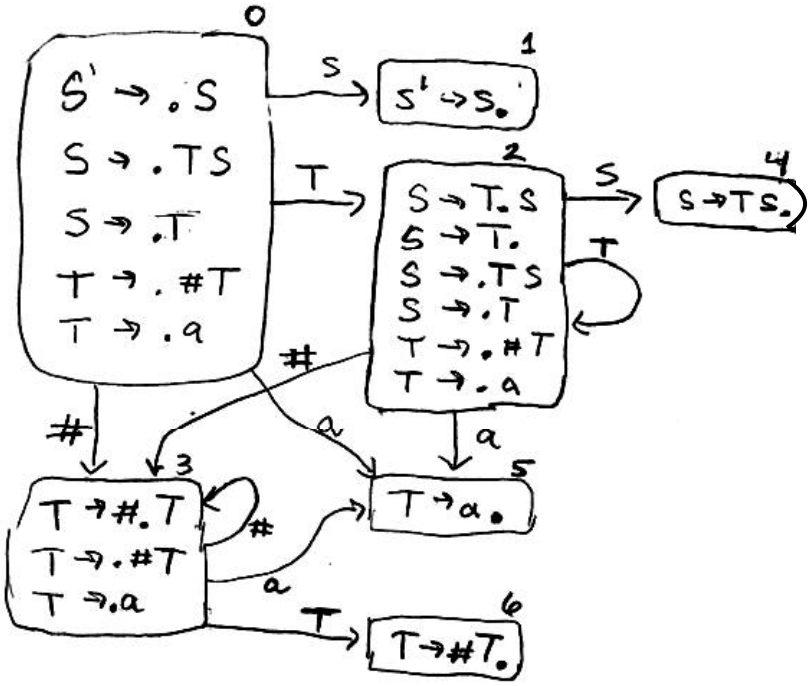
S → T S
S → T
T → # T
T → a

Eksamen 2006, oppgave 2 e

2e

	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 2f

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)		

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	

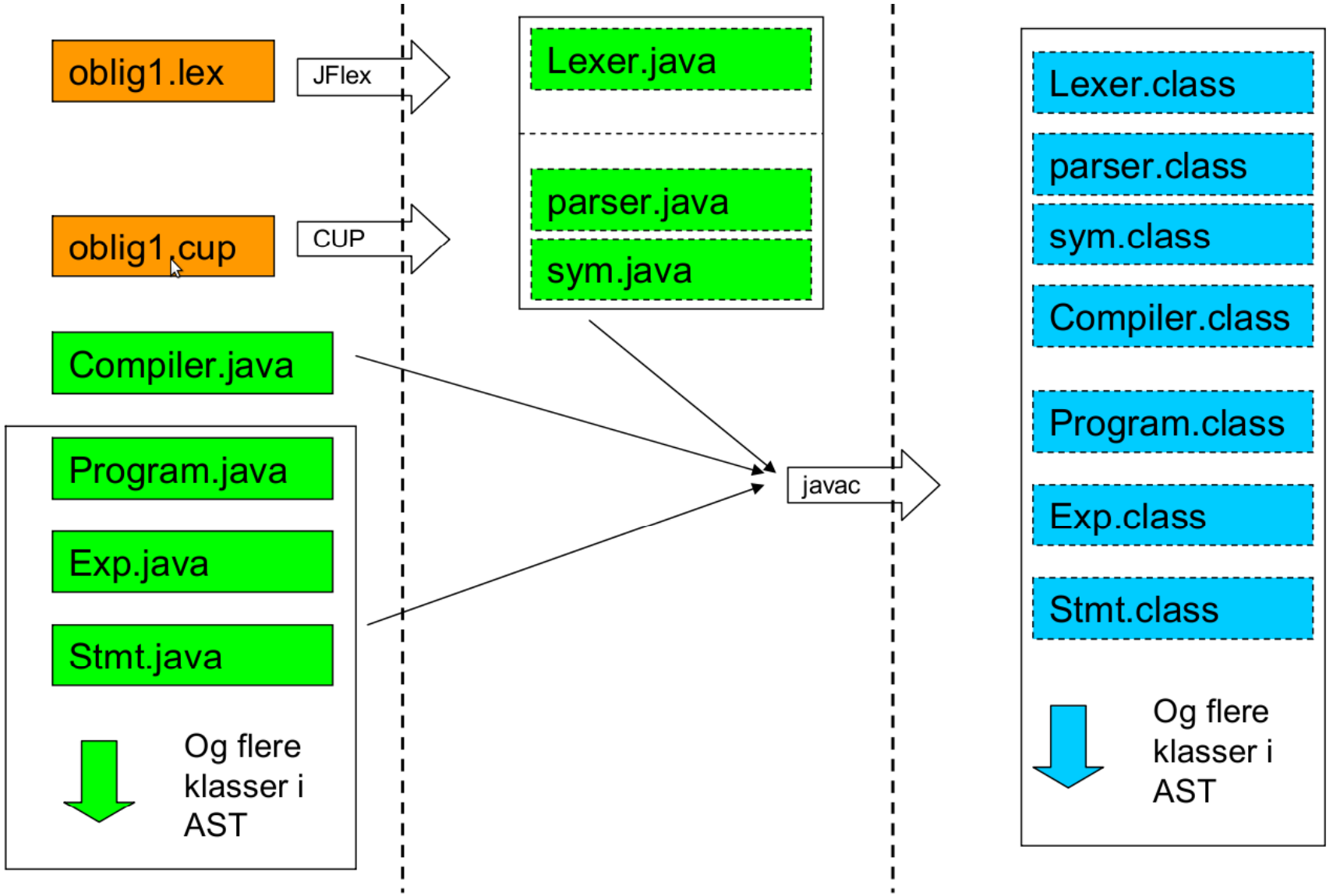
JFlex og Cup

En introduksjon for kurset INF5110
våren 2012

Mål for i dag

- *JFlex*
  - *Regulære uttrykk*
  - *Cup interaksjon, sym klassen*
  - *Tilstander*
  - *http://jflex.de/manual.html*
- *CUP*
  - *Terminaler og ikke-terminaler*
  - *Aksjonskode*
  - *Feilhåndtering*
  - *http://www2.cs.tum.edu/projects/cup/manual.html*
- *Se litt på prekoden for Oblig 1*

Filer generert av JFLex og CUP



JFlex - bruker en tredelt spesifikasjonsfil

1. *User Code:*
   - *Blir kopiert til den genererte klassen (før klassedefinisjonen).*
   - *Inneholder typisk `package` og `import` statements.*
2. *Options and Macros:*
   - *Options er sett av predefinerte egenskaper som kan inkluderes.*
   - *Makroer gjør det mulig å gi kompliserte regulære uttrykk navn.*

- *Lexical Rules:*
  - *Inneholder regulære uttrykk med tilhørende aksjonskode.*
  - *Kan definere forskjellige tilstander som hver har et sett med regulære uttrykk og aksjonskoder.*

JFlex - Options

- *Blir skrudd på med %<option_name>*
- *Vanlige options er:*
  - *%cup* - *Integrasjon med CUP*
  - *%unicode* - *Bruk Unicode*
  - *%class <class_name>* - *Navnet på generert klasse.*
  - *%line* - *JFlex vil holde rede på*
  - *%column* *linjenummer og kolonne*
  - *%state <state_name>* - *definerer en tilstand*
  - *%{ <hjelpekode> %}* - *Kopieres inn i klassen.*

JFlex - Macros

- *Har formen:*
  - `macroidentifier = regular expression`
- *Vanlig å definere whitespace og identifier som macro:*
  - `LineTerminator = \r|\n|\r\n`
  - `WhiteSpace = {LineTerminator} | [ \t\f]`
  - `Identifier = [:jletter:][:jletterdigit:]*`

JFlex - Lexical Specifications

- *Bruker regulære uttrykk til å kjenne igjen symboler.*
  1. *Bruker lengste treff.*
  2. *Mønster definert først gjelder hvis flere ekvivalent treff.*
- *Aksjonskoden i blokken etterfulgt av et regulært uttrykk er hva som blir eksekvert om uttrykket matcher.*

```
"foo"       {...}
"foobar"    {...}
{Letters}   {...}
```

```
fo      // pattern 3 [of 3]
foo     // pattern 1 [of 1,3]
foobar  // pattern 2 [of 1,2,3]
```

Expression-par eksempel

Demo 1 – Jflex

grammars/expression-par.lex

CUP

Generer en LALR parser utifra BNF grammatikk og aksjonskode.

Nodene i parseringstreet til parseren er objekter av egendefinerte klasser.

Parseren vil legge symboler fra Scanneren på stacken helt til en produksjon kan reduseres til en ikke-terminal.

- *Aksjonskoden for produksjonen vil da utføres og resultatet av den vil så legges på stacken.*
- *Redusere/shifte videre*

CUP - feilsituasjoner

To vanlige typer feil:

- *Shift-Reduce Conflict:*
  - *I denne situasjonen kan parseren både fortsette å shifte, eller gjøre en reduksjon.*
  - *CUP feilhåndtering: velger shift.*
- *Reduce-Reduce Conflict:*
  - *Kan gjøre to forskjellige reduksjoner på et gitt tidspunkt.*
  - *Som regel følge av feil med grammatikken.*
  - *CUP feilhåndtering: velger regelen som er først i filen.*
- *Mer info:*
  - *CUP Manualen*
  - *http://www.gobosoft.com/eiffel/gobo/geyacc/algorithm.html*

CUP - Spesifikasjonsfilen

Fem deler:

- *package and import specifications*
- *user code components*
- *symbol (terminal and non-terminal) lists*
- *precedence declarations*
- *the grammar*

1. *Del 1 og 2 ganske likt som i JFlex.*
2. *Del 3 er definisjoner over alle terminaler og ikke-terminaler. Her defineres også typen til terminalene.*
3. *Del 4 gjør det mulige å løse presedenskonflikter om dette ikke er gjort entydig gjennom BNF.*
4. *Del 5 er selve grammatikken med aksjonskoden.*

Expression-par eksempel

Demo 2 – CUP

grammars/expression-par.cup

Apache Ant

- *Fleksibelt bygg-verktøy for Java*
- *build.xml*

- *http://ant.apache.org/*
- *http://ant.apache.org/manual/index.html*

Apache Subversion

- *Sentralisert versjonskontroll*
- *Gratis hosting på universitetet (med tilgangskontroll)*
- *Anbefales om dere skal jobbe i team*

- *http://subversion.apache.org/*
- *http://svnbook.red-bean.com/en/1.5/index.html*
- *https://wwws.ifi.uio.no/system/svn*
- *http://www.uio.no/studier/emner/matnat/ifi/INF5750/h10/undervisningsmateriale/revision_control_and_subversion.pdf*