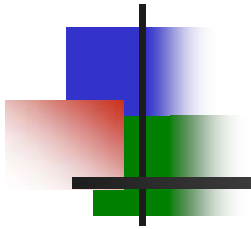


Kap. 5, del 2
LR(1)- og LALR(1)-grammatikker
INF5110 – V2008

Stein Krogdahl,
Ifi, UiO

I dag 19/2:

Time 1: Fortsette kap.5

Time 2: Hjelpelærer Fredrik Sørensen presenterer Oblig 1

Plan framovrer:

Torsdag 21/2: Som i dag

Tirsdag 26/2: Om ikke ferdig på torsdag 21/2:

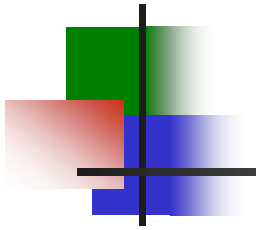
avslutning og oppgaver til kap 5.

Torsdag 28/2: Ikke undervisning

Tirsdag 4/3: Birger Møller-Pedersen starter med kap. 6

Oppgaver Kap 5:

Blir lagt ut senest onsdag morgen, med beskjed om de gjennomgås torsdag 21/2 eller tirsdag 26/2



Oppgaver som gjennomgås 26/2

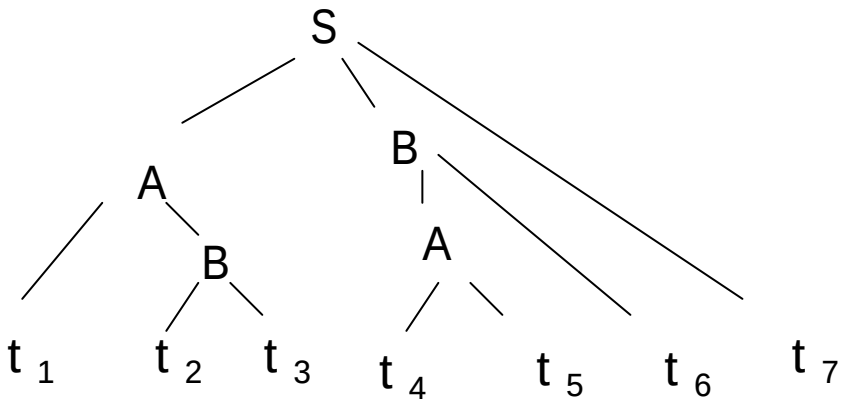
(Dette ligger på ”Undervisningsplanen” for 26/2)

Fra boka: 5.3, 5.4, 5.11, 5.12, 5.13.

Oppgave 2 fra Eksamen 2006.

Utvid grammatikken på foil 4 (Kap 5, del2) med høyreassosiativ opphøying (**), og se på hvordan konflikter da bør løses. **NB: Er i denne blitt foil 6**

”Bottom up” parsering (nedenfra-og-opp)



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
- LR(1) Det teoretisk sterkeste, om man vil se på ett lookahead-symbol
- LALR(1) ”LookAhead-LR”, veldig likt SLR, men med mer presise lookahed-mengder

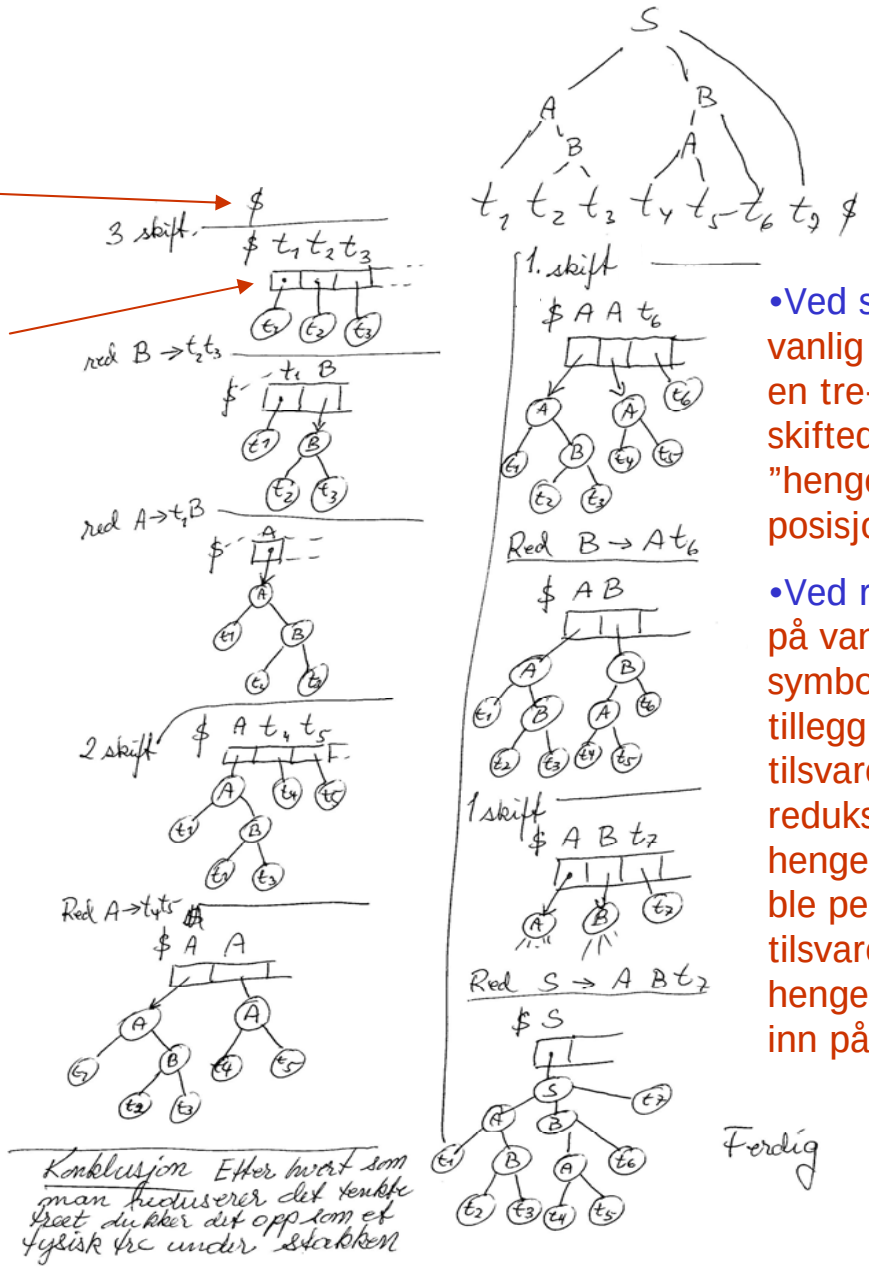
-Autimatisert:

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

Hvordan bygge et tre mens man gjør LR-parsering?

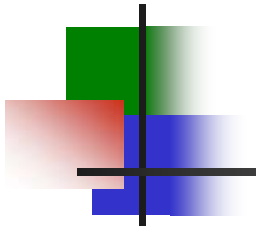
Starttilstanden, med tom stakk (pekerstakk ikke vist)

Peker-stakk som ligger ”parallelt” med symbol-stakken. I CUP, Yacc etc. er det lagt opp til å ha en slik stakk.



Merk: Her får vi et *konkret* syntaks-tre (”parserings-tre”). Dere må finne ut hvordan man kan lage et *abstrakt* parserings-tre

- Ved skift: Man skifter på vanlig måte, men lager i tillegg en tre-node som tilsvarer det skiftede terminalsymbolet, og ”henger” denne i tilsvarende posisjon på peker-stakken
- Ved reduksjon: Man reduserer på vanlig måte på symbolstakken, men lager i tillegg en tre-node som tilsvarer venstresiden i reduksjonen. Under denne henger man subtrærne som ble pekt på fra peker-stakken tilsvarende høyresiden, og henger til slutt den nye noden inn på stakken



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

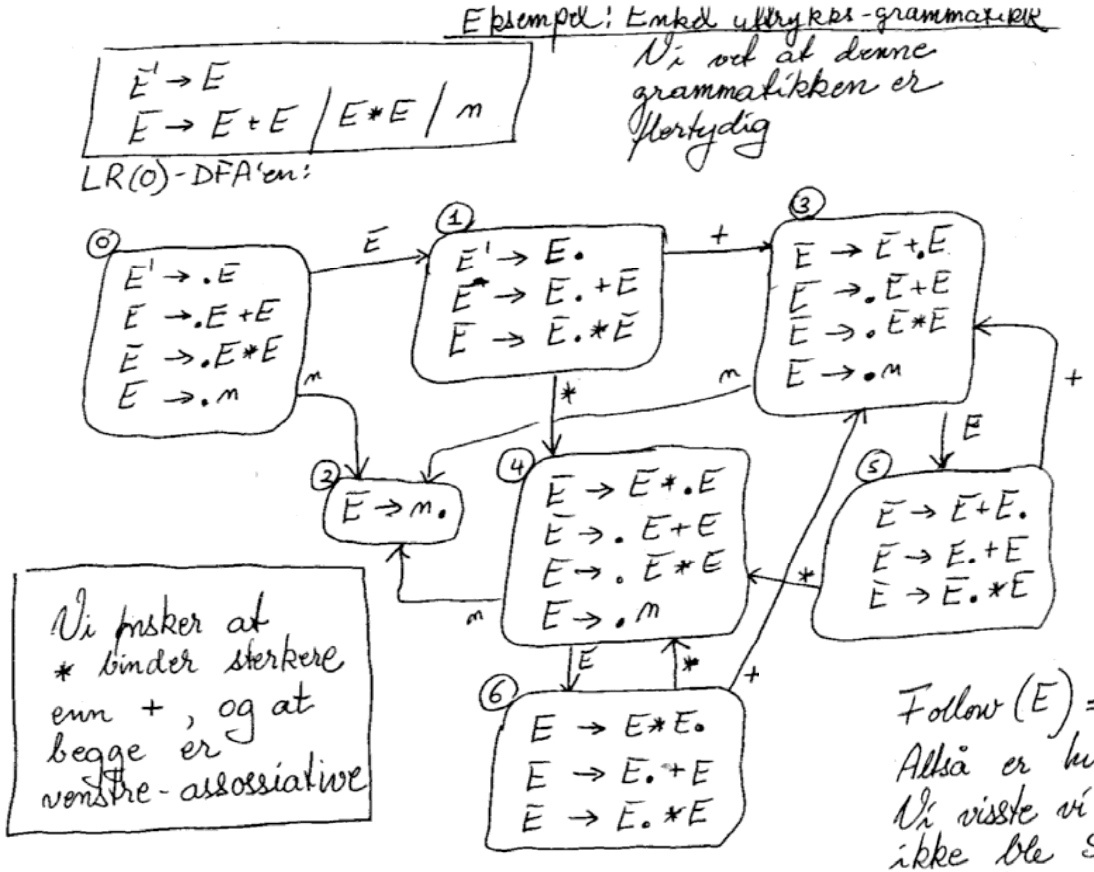
5.3.4 SLR(k) Grammars

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 First_k and 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.

Mer bruk av flertydige grammatikker ved LR-parsering (ikke i boka!)



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)	

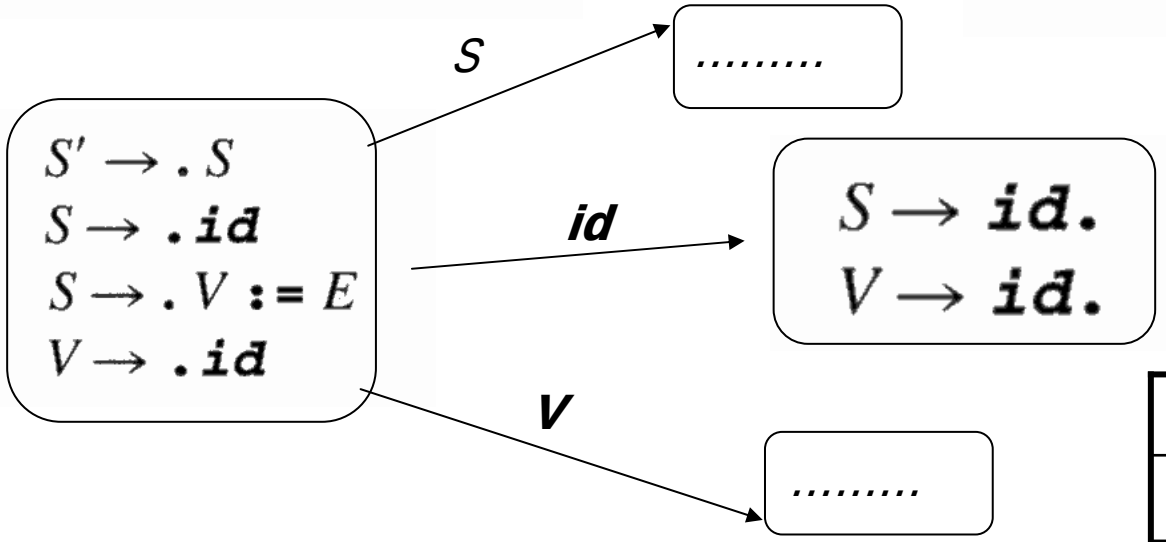
Grammatikk som ikke er SLR(1)

Men kanskje LALR(1) eller LR(1)?

- Denne grammatikken viser det seg vi kan løse ved å være nøyere med ”etterfølgermengder” i LR(0)-DFA’en *under konstruksjonen*

stmt → *call-stmt* | *assign-stmt*
call-stmt → **identifier**
assign-stmt → *var* **:=** *exp*
var → *var* **[** *exp* **]** | **identifier**
exp → *var* | **number**

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



Har har reduser/reduser-konflikt for input = $

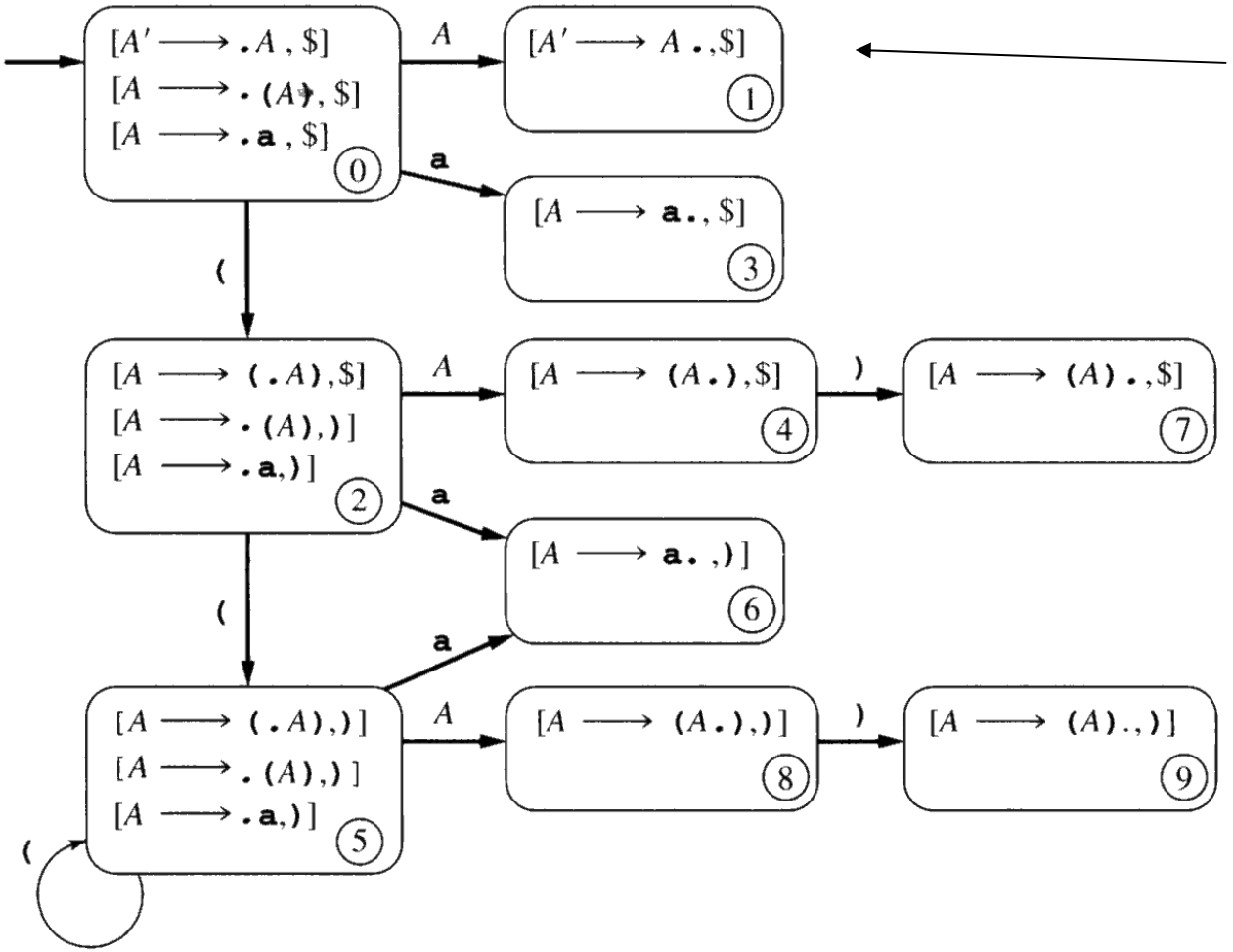
	First	Follow
S	id	$
V	id	:=, $
E	Id, n	$

Oppsett av LR(1)-DFA, sterkere enn LALR(1)

(Går ikke innom LR(1)-NFA !)

Starttilstanden er tillukningen av LR(1)-itemet: [A' → . A, $]

LR(1)-item har lookahead-symbol her



Grammatikk:

A' → A

A → (A)

A → a

A' → A. opptrer bare med look-ahead $ (angir aksept)

To tilstander i LR(1)-DFA'en regnes bare som like om de har nøyaktig den samme mengde av LR(1)-itemer (med-regnet look-ahead symbolene).

Tilstand 2 og 5 er dermed f.eks. ikke like.

LR(1)-parsering (mer generelt enn LALR(1))

a = "*look-ahead*"

Hovedidé: Vil holde greie på under oppsett av NFA og DFA hva som kan stå bak en produksjon i den sammenhengen den står

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

NFA: $[A \rightarrow \alpha . X\gamma, a]$ —**X**→ $[A \rightarrow \alpha X . \gamma, a]$

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 $[A \rightarrow \alpha . B\ \gamma, a]$, where B is a nonterminal, there are ε-transitions to items $[B \rightarrow . \beta, b]$ for every production $B \rightarrow \beta$ and *for every token* b in First(γa).

NFA: $[A \rightarrow \alpha . B\gamma, a]$ —ε→ $[B \rightarrow . \beta, b]$

For alle:
$B \rightarrow \beta_1 \mid \beta_2 \mid ..$
og alle
$b \in$ First(γa)

Spesialtilfelle, inkludert i det over ($\gamma = \varepsilon$):

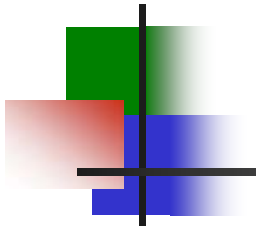
$[A \rightarrow \alpha . B, a]$ —ε→ $[B \rightarrow . \beta, a]$

For alle $B \rightarrow \beta_1 \mid \beta_2 \mid ..$

LR(1)-itemer:

$[A \rightarrow \alpha . \beta , a]$

Angir at **a** kan komme etter $A \rightarrow \alpha\beta$ i den aktuelle sammenhengen. Altså at **a** kan komme bak *hele* $\alpha\beta$, (og *ikke* bak punktumet, med mindre $\beta = \varepsilon$)



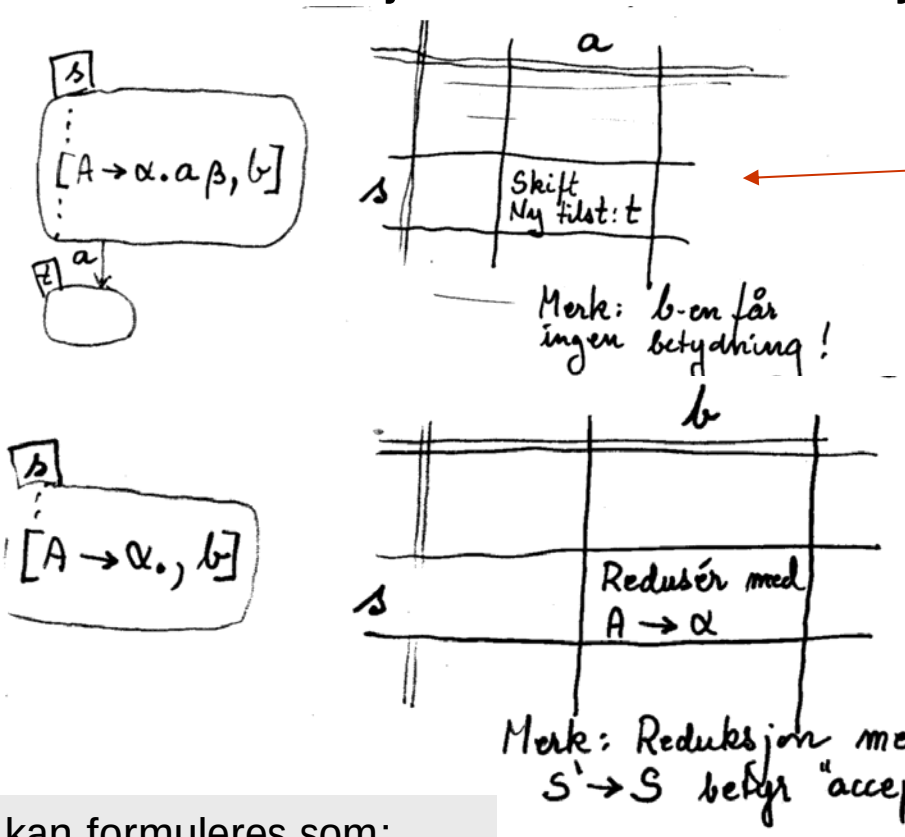
Bokas definisjon av algoritmen ved LR(1)-parsering. (samme svakheter som for LR(0) og SLR(1), men ikke rettet her)

The General LR(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 LR(1) item of the form $[A \to \alpha . X\, \beta, a]$, 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 LR(1) item $[A \to \alpha\, X . \beta, a]$.
2. If state s contains the complete LR(1) item $[A \to \alpha ., a]$, and the next token in the input string is a, then the action is to reduce by the rule $A \to \alpha$. A reduction by the rule $S' \to S$, where S is the start state, is equivalent to acceptance. (This will happen only if the next input token is \$.) In the other cases, the new state is computed as follows. Remove the string α and all of its corresponding states from the parsing stack. Correspondingly, back up in the DFA to the state from which the construction of α began. By construction, this state must contain an LR(1) item of the form $[B \to \alpha . A\, \beta, b]$. Push A onto the stack, and push the state containing the item $[B \to \alpha\, A . \beta, b]$.
3. If the next input token is such that neither of the above two cases applies, an error is declared.

Er grammatikken LR(1)?

- Metode som før: Sjekk om du får en entydig parseringstabell.



Merk: b'en få ingen betydning her. Den har bare betydning i slutt-itemer, og er bare med for at oppsett av automaten skal bli riktig

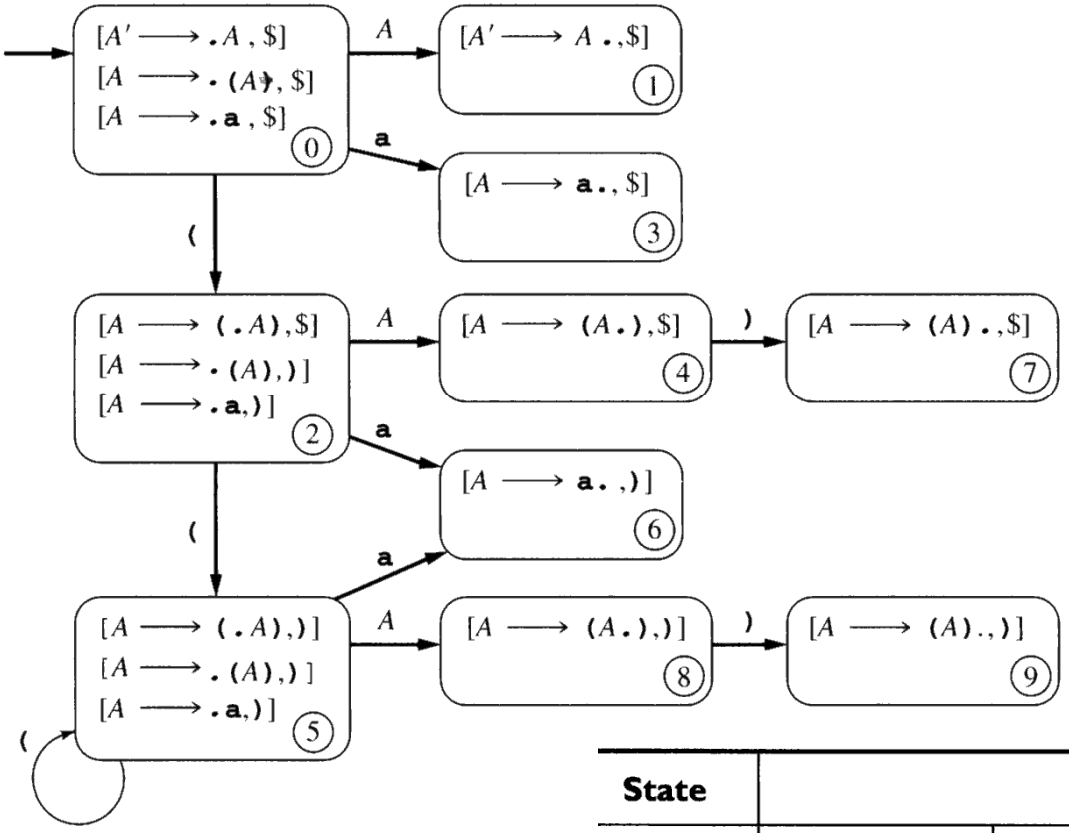
Siden $S' \rightarrow S\cdot$ alltid bare opptrer med look-ahead = $

LR(1) – kravet kan formuleres som:

1. For any item $[A \rightarrow \alpha . X\ \beta, a]$ in s with X a terminal, there is no item in s of the form $[B \rightarrow \gamma ., X]$ (otherwise there is a shift-reduce conflict).
2. There are no two items in s of the form $[A \rightarrow \alpha ., a]$ and $[B \rightarrow \beta ., a]$ (otherwise, there is a reduce-reduce conflict).

Samme som *a* over

Samme som *b* over



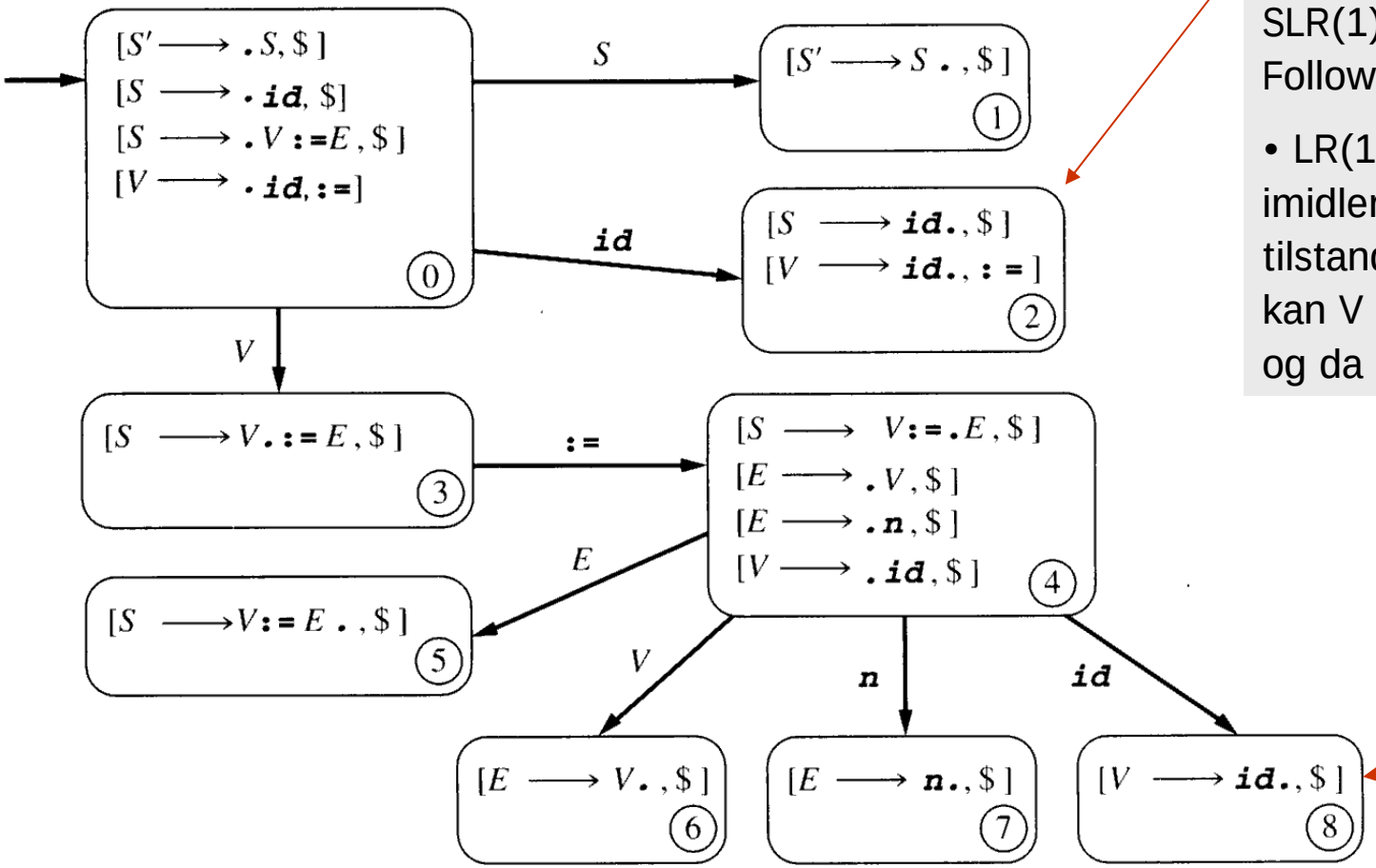
LR(1) – tabell for:

$$A \rightarrow (A) \mid \mathtt{a}$$

- Tabellen ble entydig, derfor er grammatikken LR(1).
- Dette er ingen overraskelse, siden vi visste at grammatikken er LR(0)!

State	Input				Goto
	(	a	)	$	A
0	s2	s3			1
1				accept	
2	s5	s6			4
3				r2	
4			s7		
5	s5	s6			8
6			r2		
7				r1	
8			s9		
9			r1		

LR(1) –DFA for problem-grammatikken som ikke var SLR(1)



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

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

Grammatikken *er* altså LR(1)

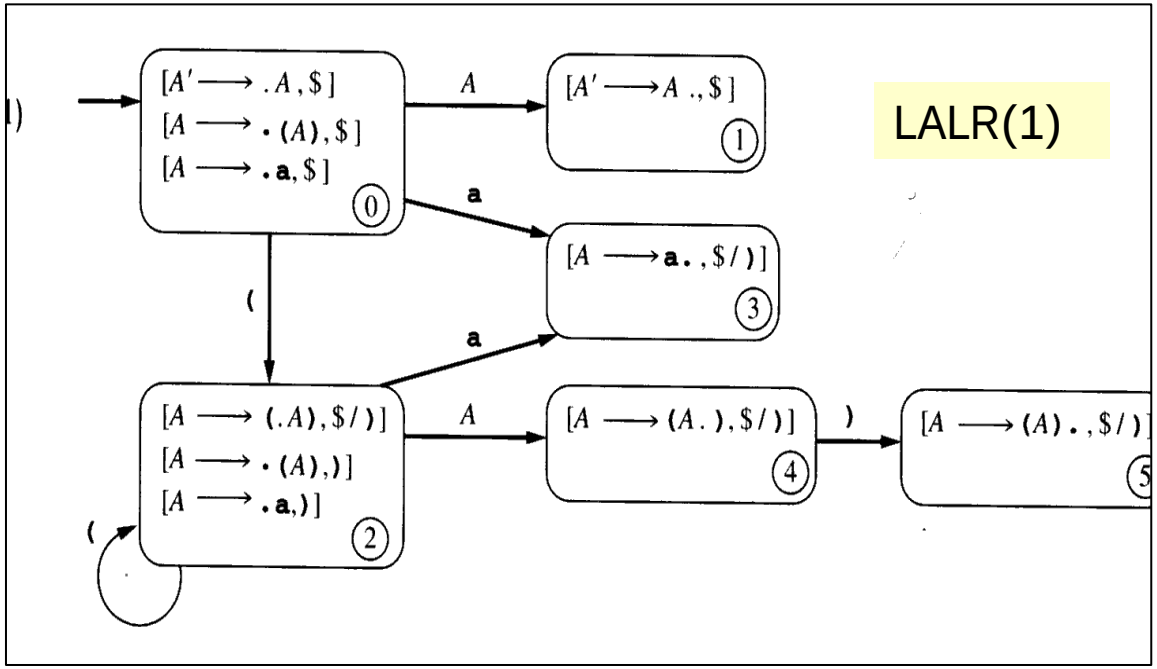
Overgang fra LR(1) til LALR(1)

- ”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 laget fra 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 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.

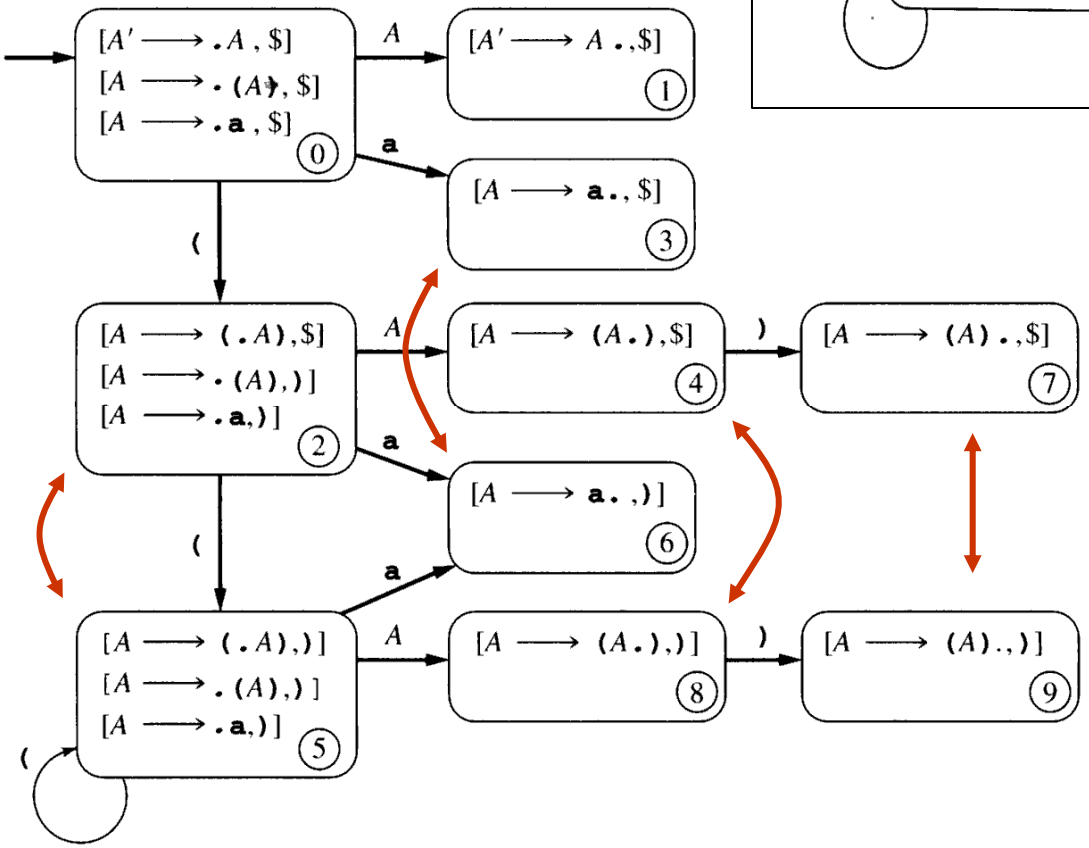
Sammenligning av LR(1)- og LALR(1)-DFA'er for samme grammatikk:

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

LALR(1)



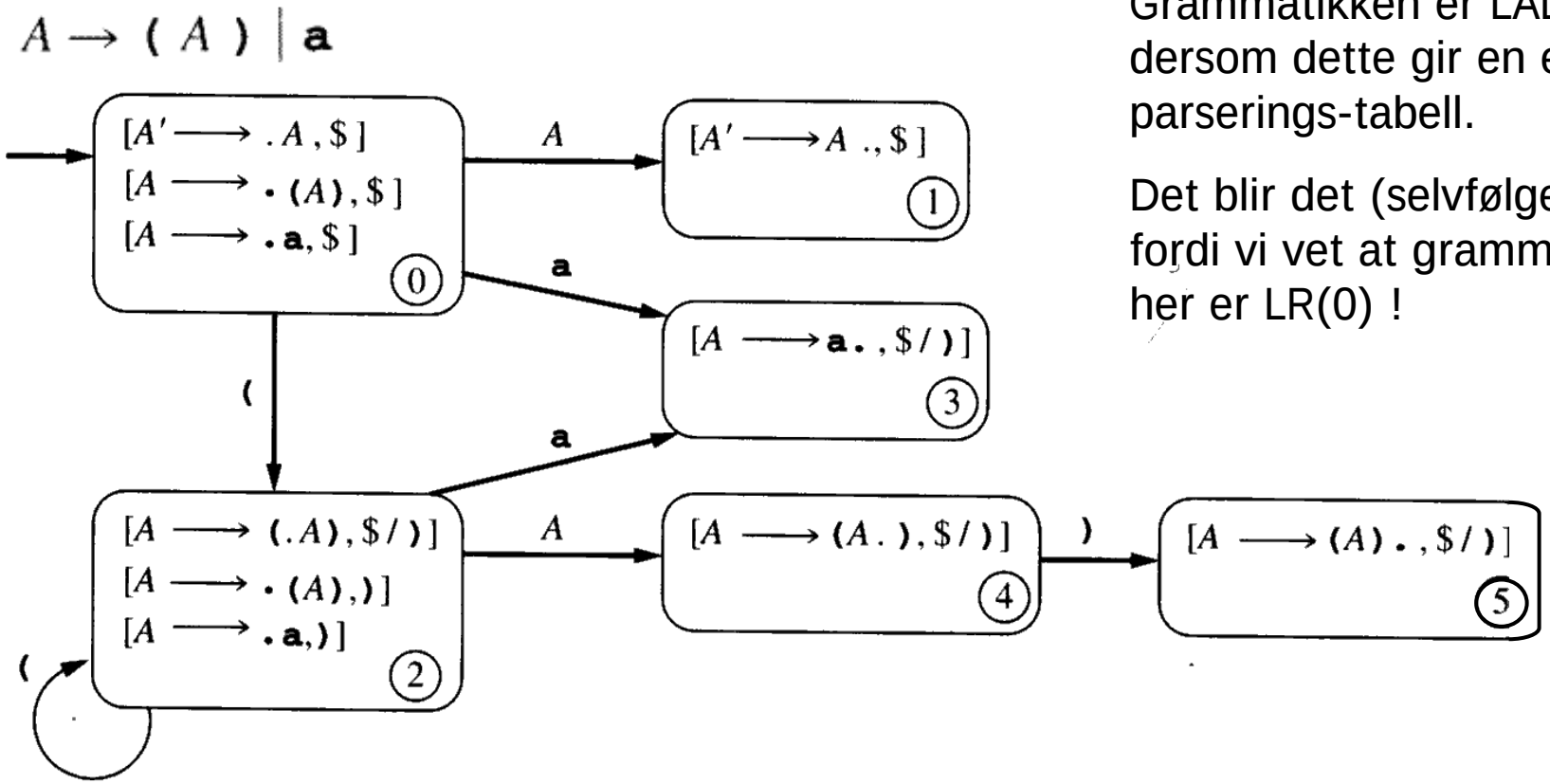
LR(1)



- Her får vi ingen hjelp ved å gå over fra SLR(1) til LALR(1)
- Det er fordi Follow(A) = {$,)}, og det er også lookahead-mengdene for alle slutt-itemer LALR(1)-DFA'en over
- Men vi trenger da heller ikke slik hjelp, siden grammatikken allerede er LR(0)!

Kan lage LALR(1)-DFA'en direkte

- Lag LR(0)-DFA'en (og sett av plass til "look-ahead"-symboler).
- Sett inn lookahead $ på [A' →.A,], altså [A' →.A, $]
- Fyll på med det som "må med" av lookahead'er i en kompletteringsprosess, til det stopper



Grammatikken er LALR(1) dersom dette gir en entydig parserings-tabell.

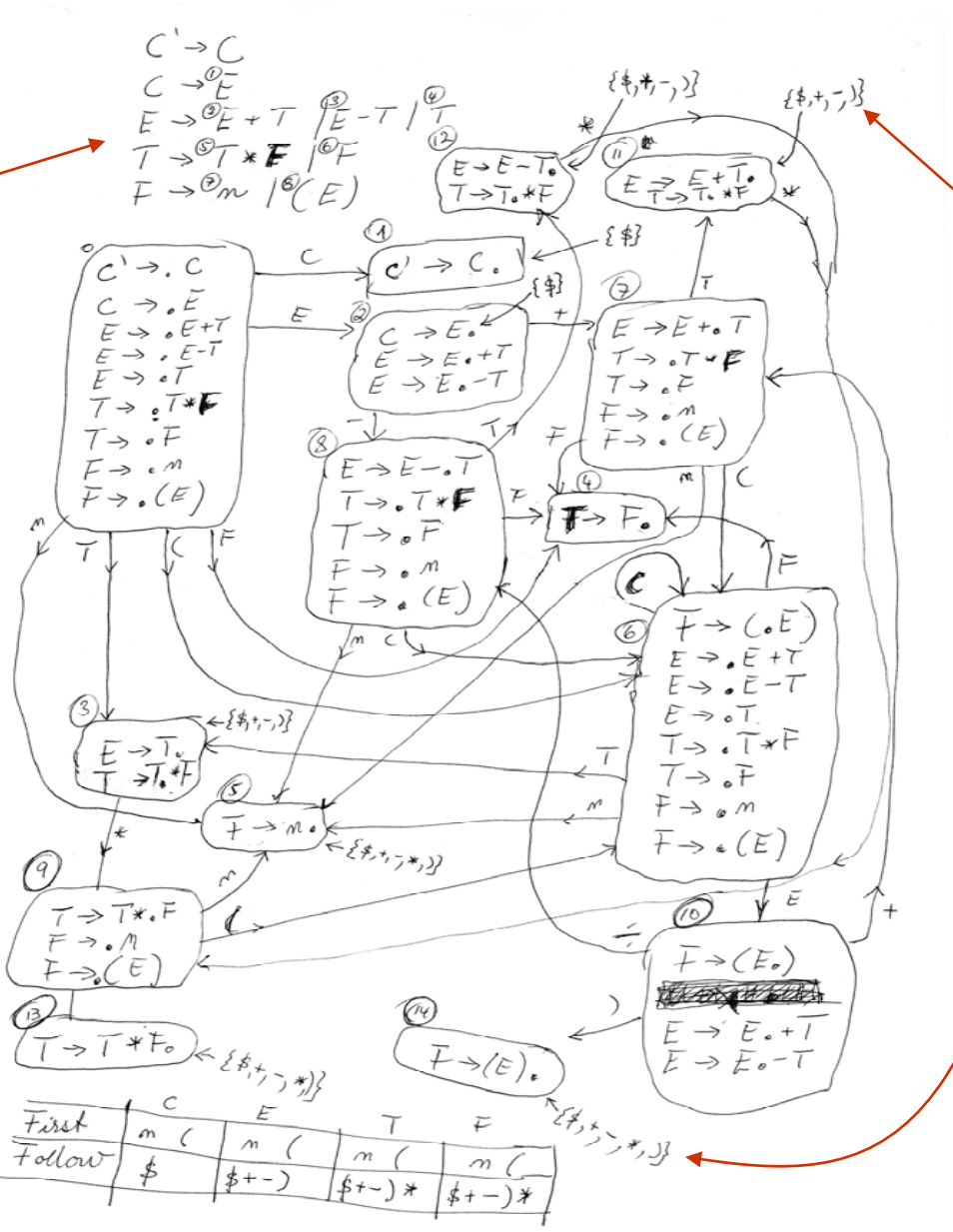
Det blir det (selvfølgelig) her fordi vi vet at grammatikken her er LR(0) !

LR(0)-DFA'en til tabellen på neste side

Dette tilsvarer utskriften fra Yacc i figur 5.12 i boka. (CUP kan lage tilsvarende)

Nummereringen av reglene (produksjonene) ble litt rar. Den er angitt ved tallene med ring rundt)

Normalt ville man slått sammen regel 2 og 3 til én regel.



Litt snudd av plasshensyn

Dette er ettfølgermengden for venstresiden av reduksjons-itemet

Typisk Yacc-produsert parseringstabell (tilsvarende for CUP)

Merk påfyll av ekstra reduksjoner, som en plass-optimalisering i Yacc. Også i CUP?

Grammatikk:

command → exp

exp → exp + term | exp – term | term

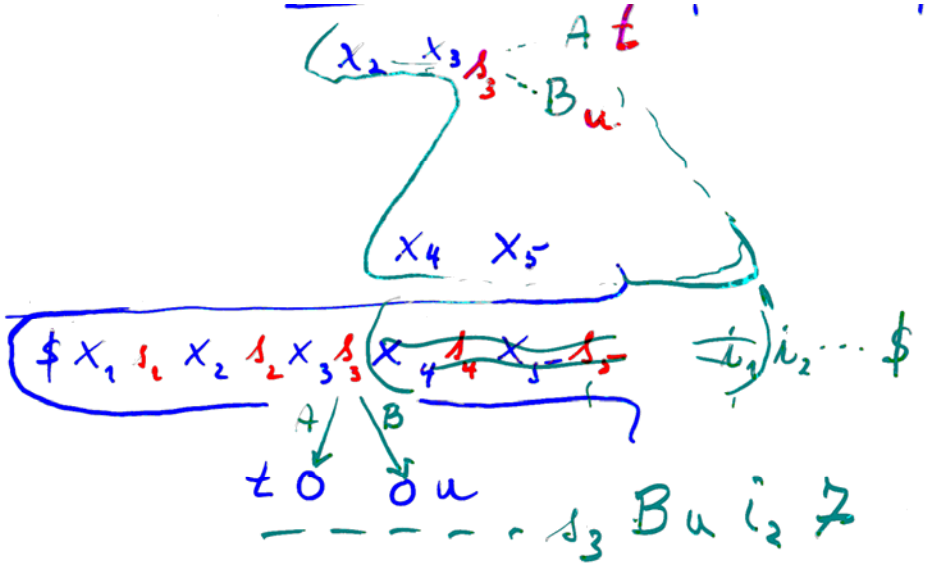
term → term * factor | factor

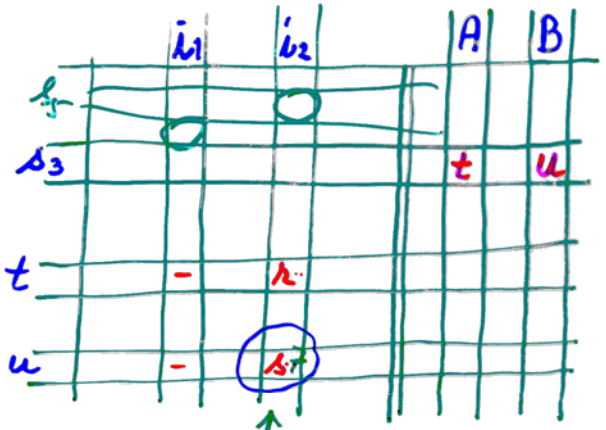
factor → NUMBER | (exp)

Tabellen under er tenkt laget fra en LALR(1)-DFA, men det blir her det samme som fra en SLR(1)-DFA (forrige foil)

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
(det eneste vi skal se på)





Velger til slutt å pushe på B, som etter å ha fjernet i_1 gir tilstand u

1. Pop states from the parsing stack until a state is found with nonempty Goto entries.
2. If there is a legal action on the current input token 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.
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.

i_1

s_3 (t og u)

} Ta vekk én og én input, og gjenta 2

Eksempel: if-setning er mindre generell enn setning

5.45

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

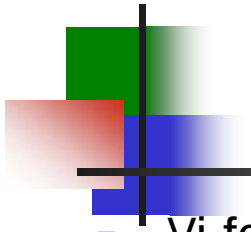
(ingen skift, dessverre!)

- 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



Oppsummering om LR-parsering (bottom-up)

- Vi formulerer vår grammatikk som basal BNF
- Konflikter kan løses med:
  - omdefinere BNF'en
  - eller ved direktiver til CUP/Yacc/Bison (assosiativitet, presedens, etc.)
  - eller løser det senere i semantisk analyse
  - NB: Ikke **alle** konflikter *kan* løses av selv LR(1) – skriv om! (eks: A → a | aAa)
- 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 100K 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)	Mest nøyaktig (færrest) konflikter. Best feilhåndtering.	Svært mange tilstander (typisk tabell på 1M felter for et reelt språk). Brukes?

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