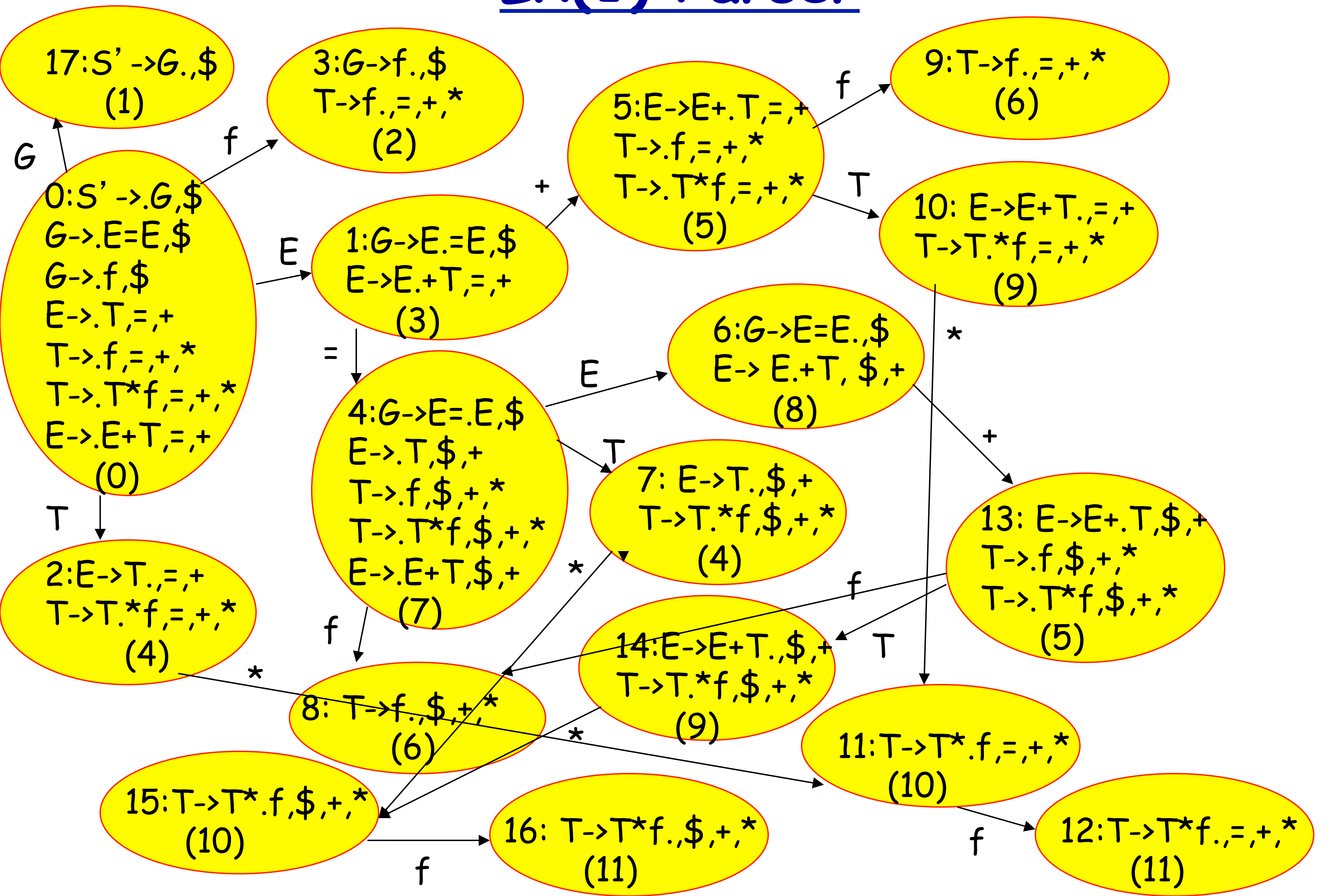
LR(1) Parser



17:S' ->G.,$
(1)
3:G->f.,$
T->f.,=,+,*
(2)
5:E->E+.T,=,+
T->.f,=,+,*
T->.T*f,=,+,*
(5)
9:T->f.,=,+,*
(6)
10: E->E+T.,=,+
T->T.*f,=,+,*
(9)
0:S' ->.G,$
G->.E=E,$
G->.f,$
E->.T,=,+
T->.f,=,+,*
T->.T*f,=,+,*
E->.E+T,=,+
(0)
1:G->E.=E,$
E->E.+T,=,+
(3)
6:G->E=E.,$
E-> E.+T, $,+
(8)
4:G->E=.E,$
E->.T,$,+
T->.f,$,+,*
T->.T*f,$,+,*
E->.E+T,$,+
(7)
7: E->T.,$,+
T->T.*f,$,+,*
(4)
13: E->E+.T,$,+
T->.f,$,+,*
T->.T*f,$,+,*
(5)
2:E->T.,=,+
T->T.*f,=,+,*
(4)
14:E->E+T.,$,+
T->T.*f,$,+,*
(9)
8: T->f.,$,+,*
(6)
11:T->T*.f,=,+,*
(10)
15:T->T*.f,$,+,*
(10)
16: T->T*f.,$,+,*
(11)
12:T->T*f.,=,+,*
(11)
G
f
E
T
+
=
*

LR(1) Parser

- LR(1) Item

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

- $A \rightarrow \alpha\beta$ is a production
- c is the lookahead

Constructing DFA for LR(1) Parser

Basic Idea of closure:

For item [A -> α.Xβ, c], where X is nonterminal and production X->δ exists, there exists a rightmost derivation

$$G \Rightarrow \varphi A u w \Rightarrow \varphi \alpha X \beta u w$$

$\Rightarrow$item [X ->. δ, v] is valid for viable prefix $\varphi\alpha$ with v in FIRST(βuw)

$\Rightarrow$If β => the empty string, then v = u.

Closure of set of Items I:

```
cset = I;
repeat
   for each item [A->  α.Xβ, a]  in cset where X is a nonterminal,
           Add all items [X-> . δ,b] for all b in FIRST(βa) to cset
                        (if not already in cset)
until no more items added;
```

GOTO(I,X): Assume [A-> α.Xβ, a] in I. Then GOTO(I,X) = closure of items [A-> αX.β, a].

Computing Closure of LR(1) items

S' -> G
G -> E = E | f
E -> T | E + T
T -> f | T * f

In LR(0) machine:

S' -> .G

G -> .E=E   G -> .f

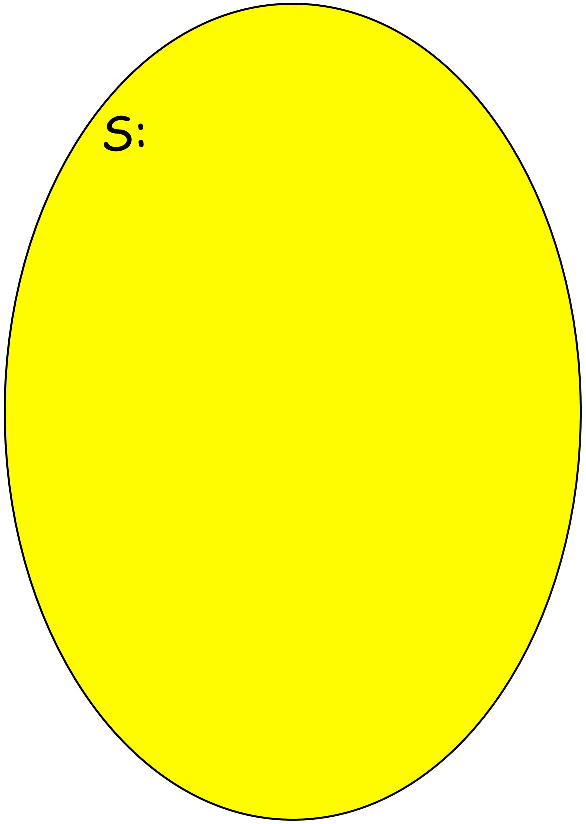
E -> .T   E -> .E+T

T -> .f   T -> .T*f

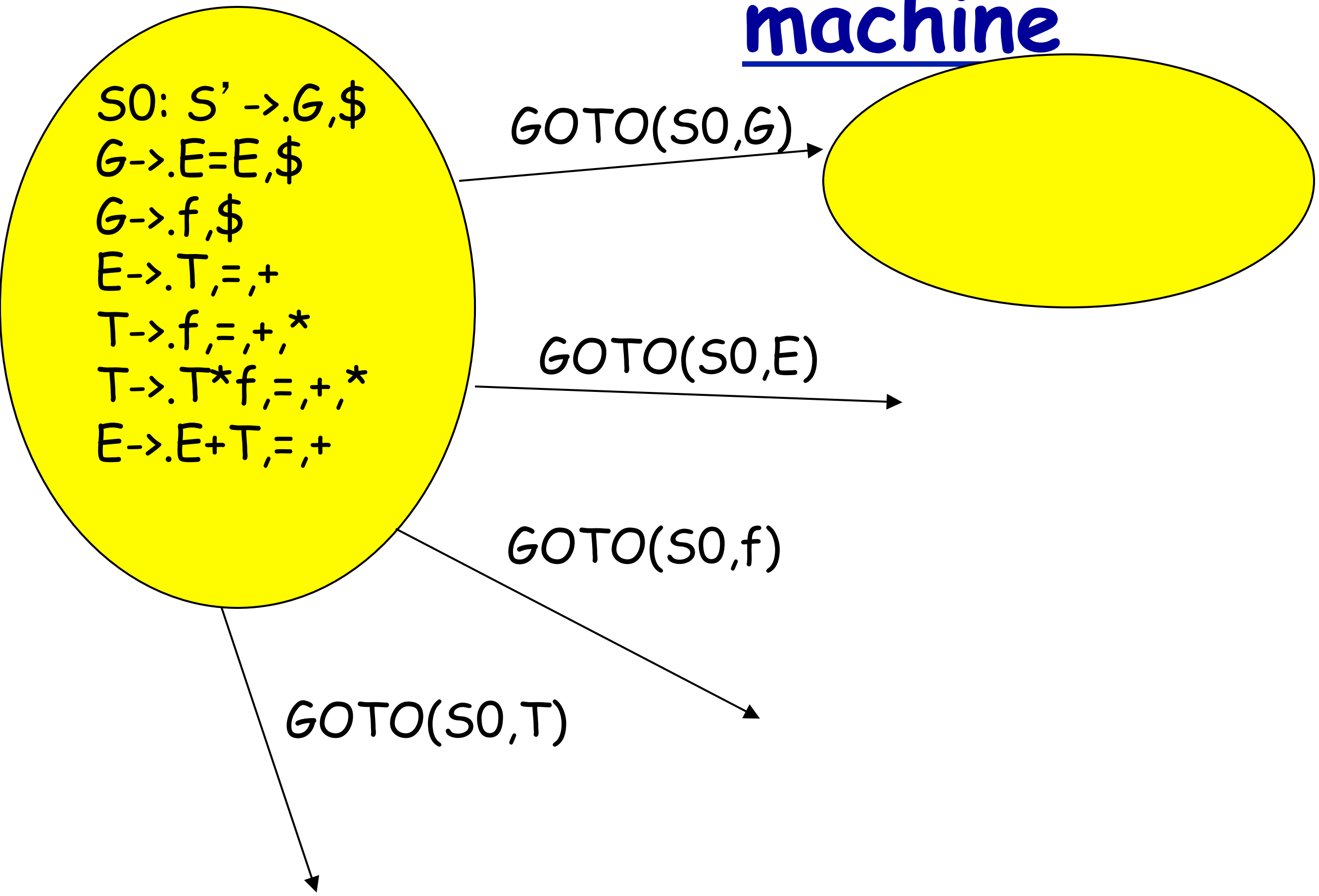
In LR(1) machine:

[S' ->.G,$]

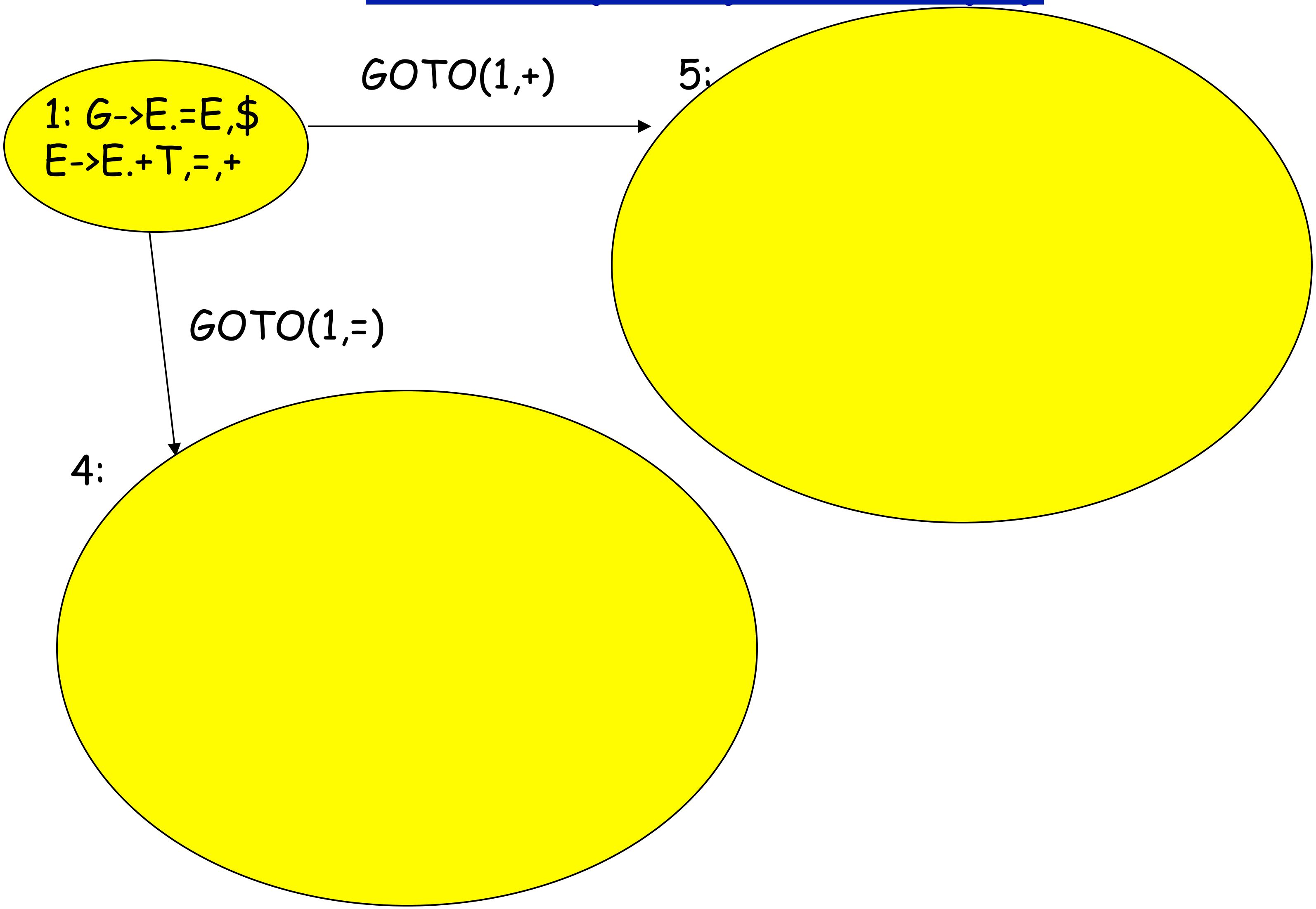
[G->.E=E,$]   [G->.f,$]



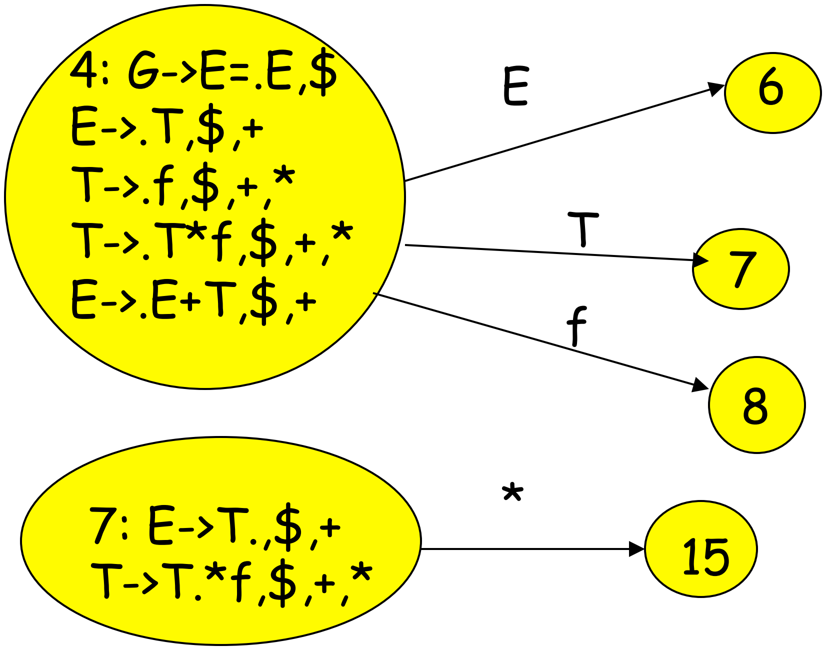
Computing GOTO(I,X) in LR(1) machine



More Extensive Example of GOTO(I,X) in LR(1)

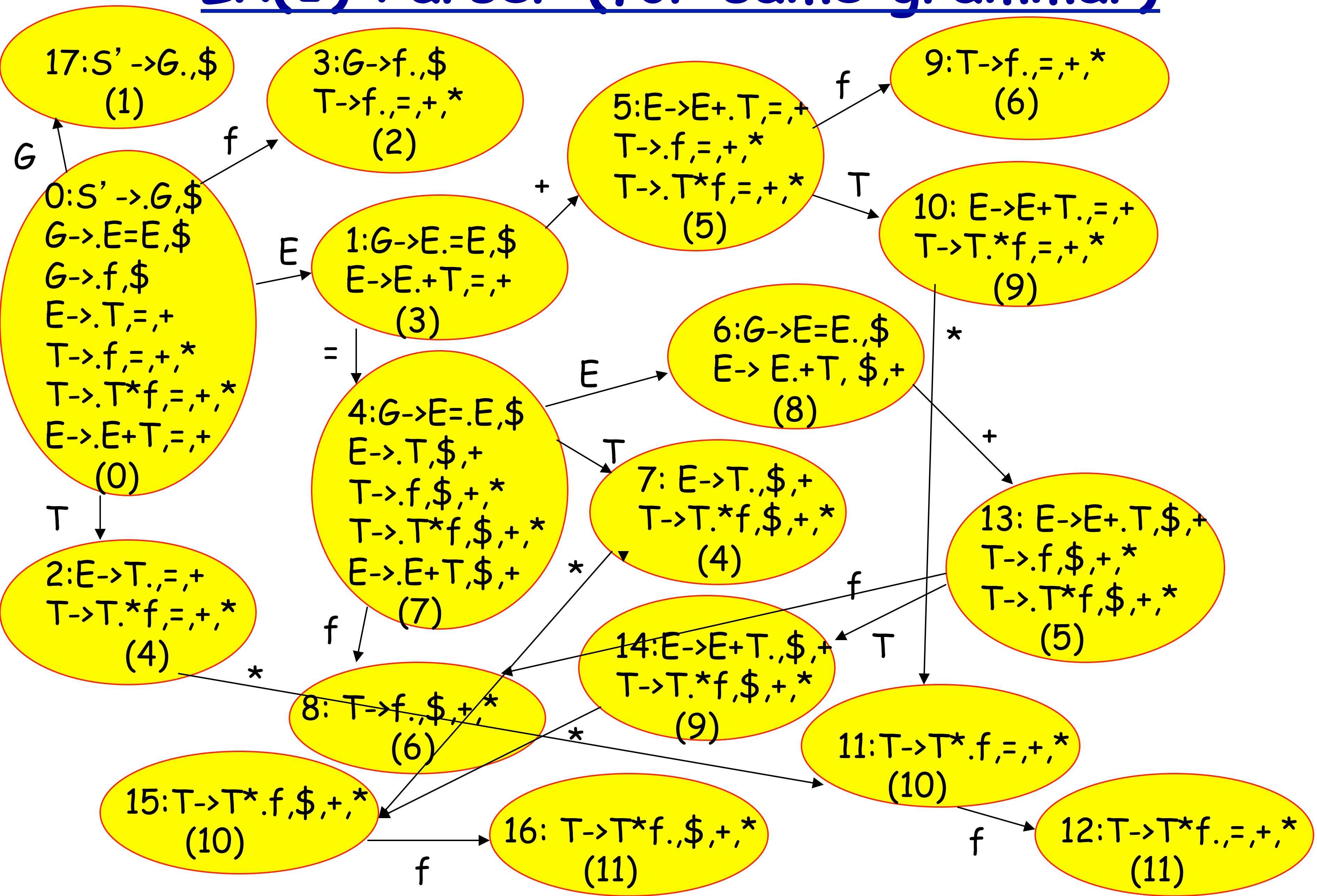


LR(1) Table Construction – Partial Example



State	Action					GOTO		
	f	=	+	*	$	G	E	T
4								
7								

LR(1) Parser (for same grammar)



LALR(1) Grammars and Parsing

Characteristic: Same number of states as SLR(1) with more power due to lookahead in states.

But, less power than canonical LR(1) because less states.

2 Approaches to Table Construction:

* Construct LR(1) sets of items (DFA) and merge states with same core.
* Construct LR(0) sets of items and generate lookahead information for each of those states.

Properties of LALR we will see:

* May perform REDUCE rather than ERROR like LR(1), but will catch error before any more input is processed.
* LALR derived from LR with no shift-reduce conflict will also have no shift-reduce conflict (Shift-reduce conflicts arise from core, not lookahead therefore merging has no effect.)
* LALR merging can create reduce-reduce conflicts not in LR from which LALR derived.

Constructing LALR from LR(1)

1. Construct LR(1) DFA.

2. Identify all sets of states with same core.

   In our example: (2,7), (5,13), (8,9), (10,14), (11,15), (12,16)

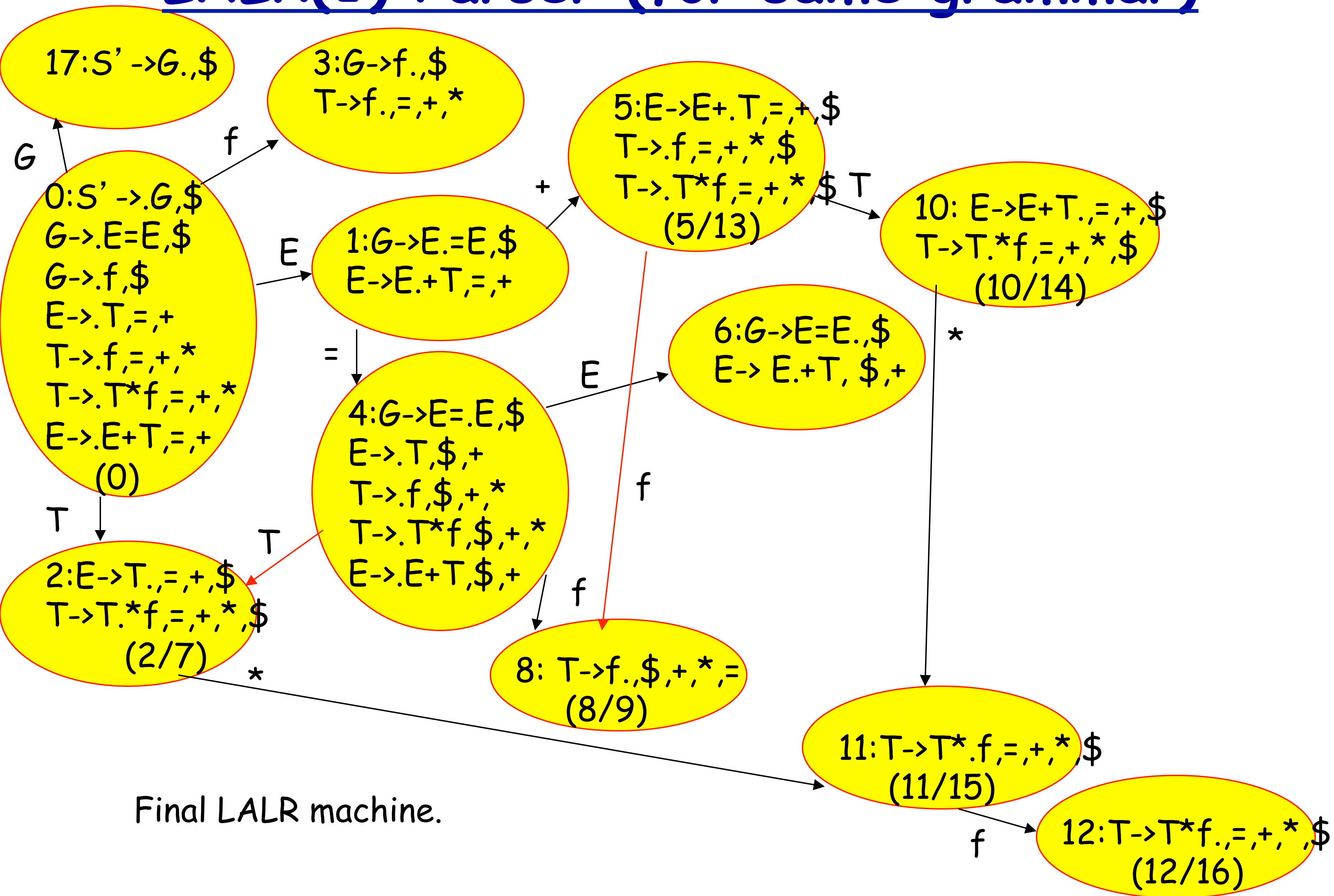
3. Merge the states with the same core into a single state in LALR:
   1. create single state with that core
   2. merge lookaheads from all LR(1) states with that core

   Example:
   LALR(1) state 2: from LR(1) states 2 and 7:

   E -> T.,
   T -> T.*f,

   Add edges to LALR machine based on edges of LR machine.

LALR(1) Parser (for same grammar)



Final LALR machine.

Is a grammar LR(1)? LALR(1)?

* Construct LR(1) (or LALR(1)) parse table using lookahead information.
If there exists any multidefined entries, then the grammar is NOT LR(1) (LALR(1)).

or

* Construct LR(1) (or LALR(1)) DFA. If there exists any inadequate states for which lookahead does not resolve the local ambiguity as below, then the grammar is NOT LR(1) (LALR(1)).

If for all states including $A \to \alpha.$, {a1,a2,....,an}
$B \to \beta.$, {b1,b2,...bm}

{a1,a2,...an} $\cap$ {b1,b2,....,bm} = 0

AND

If for all states including $A \to \alpha.$, {a1,a2,....,an}
$B \to \beta.a\delta$, {b1,b2,...bm}

{a1,a2,....,an} $\cap$ {a} = 0

then the grammar is LR(1) (LALR(1)).

Property 1: May reduce before error.

Consider string: f+f

LR(1):

Stack	Input	
0	f+f$	
0f3	+f$	
0T2	+f$	
0E1	+f$	
0E1+5	f$	
0E1+5f9	$	ERROR!

LALR(1):

0	f+f$	
0f3	+f$	
0T2	+f$	
0E1	+f$	
0E1+5	f$	
0E1+5f8	$	
0E1+5T10	$	
0E1	$	ERROR! - 2 extra reductions.

Property 2: No shift-reduce in LR => no shift-reduce conflict in LALR.

Assume when merge, we get a shift-reduce conflict in some state:

LALR state: $A \rightarrow \alpha.$, a => reduce on a
$B \rightarrow \beta.a\delta$,b => shift on a

This implies that some set Si from LR(1) machine has the item

$A \rightarrow \alpha.$, a to be included in this merge.
Since the cores of all states merged together are the same, Si must also contain

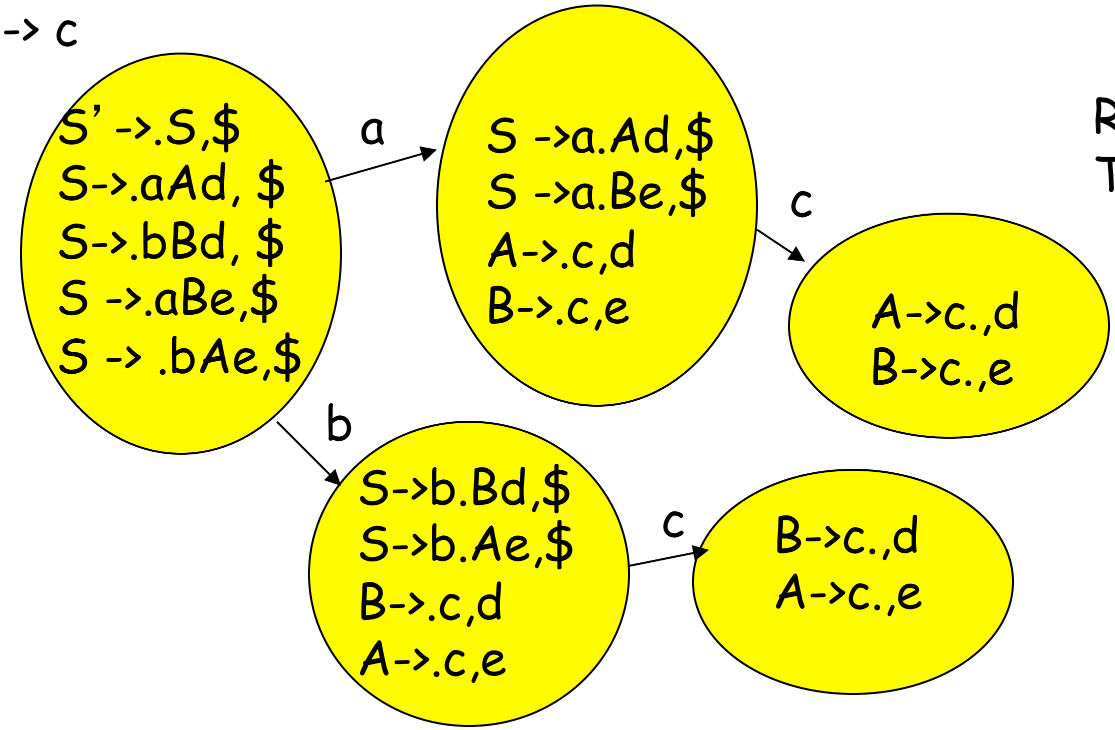
$B \rightarrow \beta.a\delta$,c for some lookahead c.

This implies that the shift-reduce conflict also must exist in Si within the LR(1) machine. Contradiction.

KEY: Merging states cannot cause shift-reduce conflicts in LALR.

Property 3: Merging states for LALR(1) can produce reduce-reduce conflicts.

Consider:
S1->S
S -> aAd | bBd | aBe | bAe
A -> c
B -> c



To create LALR,
merge:
A->c., d,e
B->c., e,d

Reduce-reduce conflict!!
Therefore,
not LALR grammar