Grammar Class Inclusion Tree

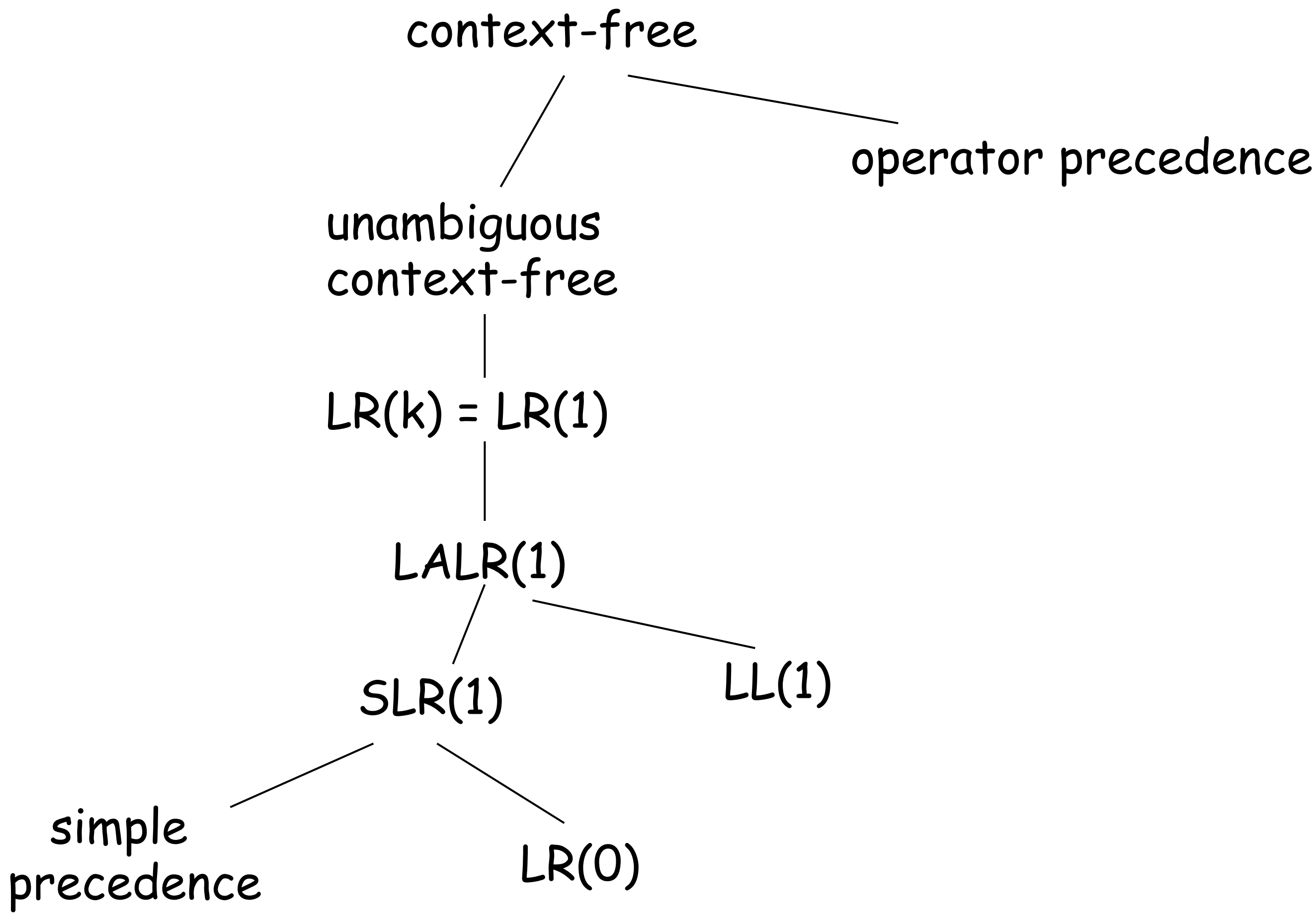


Table-driven Bottom-up Parsing

- Start at the leaves and grow toward root
- Bottom-up parsers handle a large class of grammars
- Most prevalent is based on LR(k)
- Why LR Parsing ?
  - Recognize many programming languages
  - Detect Syntax Errors
  - No backtracking

Bottom Up Parsing
Shift and Reduce

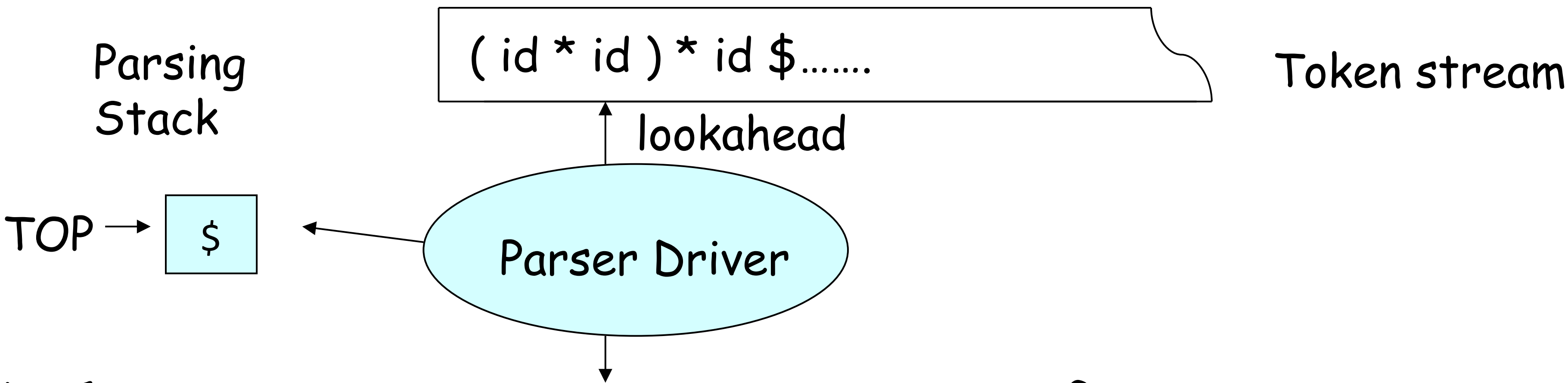
Consider: S -> bMb
M -> (L | a
L -> M a)

String: bab

<u>**Stack:**</u>	<u>**Input and Current Lookahead:**</u>
$	bab$

- Terminology: Handle: a substring β of the tree's frontier that
  - matches some production $A \rightarrow \beta$ that occurs as one step in the rightmost derivation

Table-driven Bottom-up Parsing Overview



parse states

	Action					Goto		
	(	id	)	*	$	T	T'	F
0								
1								
2								

Table[state,terminal] =

- shift token and state onto stack.
- reduce by production $A \rightarrow \beta$
  pop rhs from stack; push A; push next state
  given by Goto[exposed state,A]
- accept
- error

LR Parsing Example

1: P -> b S e
2: S -> a ; S
3: S -> b S e ; S
4: S -> ε

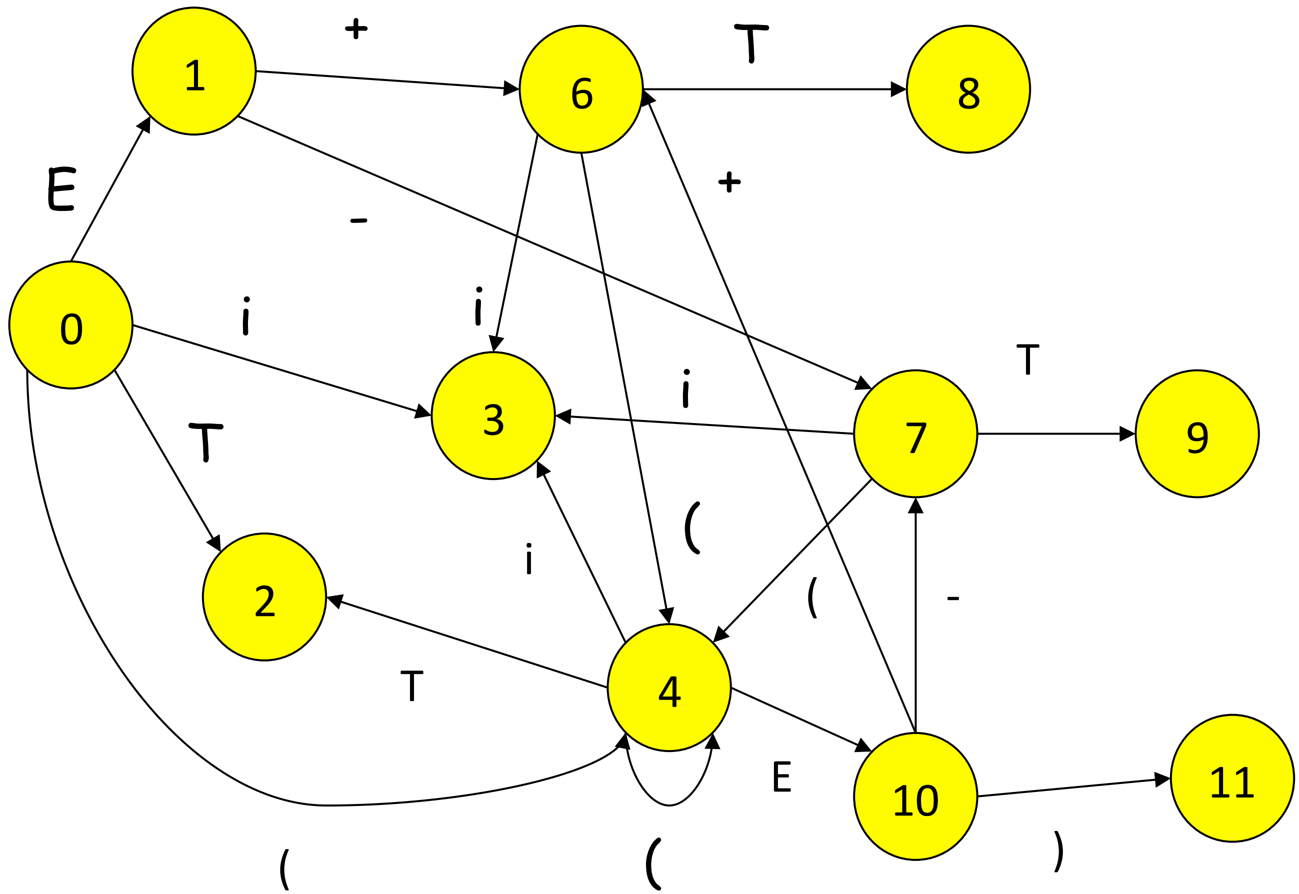
Stack	Input
0	ba;a;e$
0b1	a;a;e$
0b1a5	;a;e$
0b1a5;6	a;e$
0b1a5;6a5	;e$
0b1a5;6a5;6	e$
0b1a5;6a5;6S10	e$
0b1a5;6S10	e$
0b1S2	e$
0b1S2e3	$
accept!	

Parse Table

state	b	e	a	;	$	P	S
0	s1						
1	s4	r4	s5				2
2		s3					
3					accept		
4	s4	r4	s5				7
5				s6			
6	s4	r4	s5				10
7		s8					
8				s9			
9	s4	r4	s5				11
10		r2					
11		r3					

DFA for Parser

S -> E
E -> T | E + T | E - T
T -> I | (E)



Reduce States:

3: T -> i
2: E -> T
8: E -> E + T
9: E -> E - T
11: T -> (E)
1: (on $) S -> E

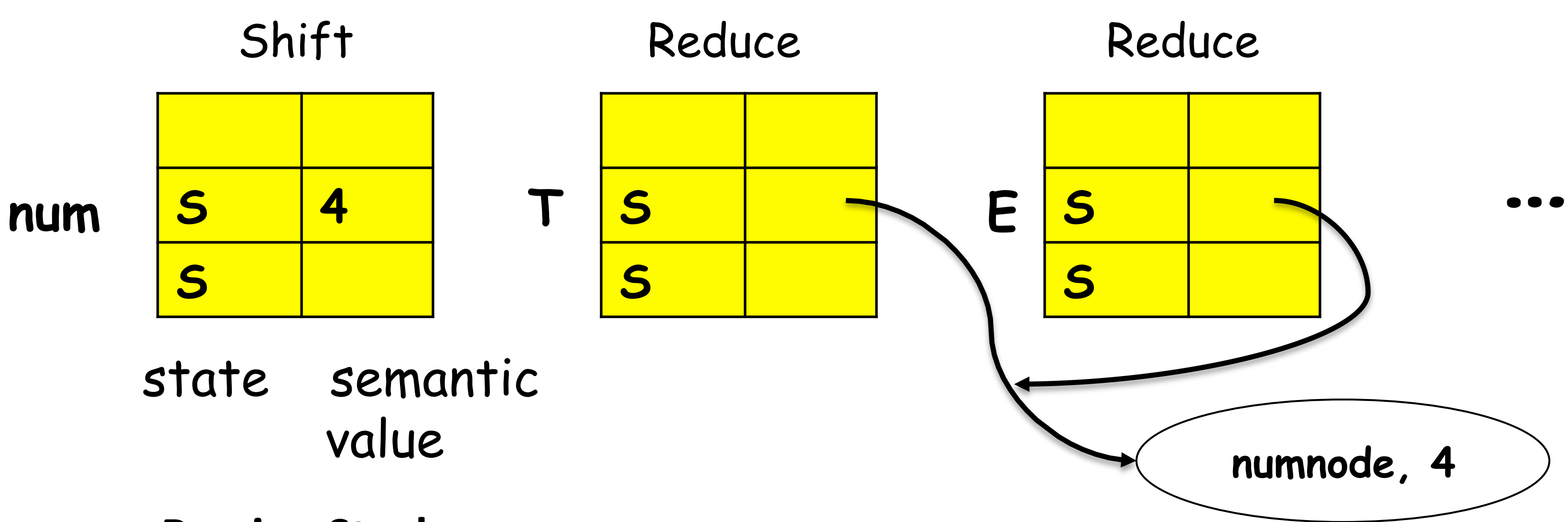
stack	input
0	i-(i+i)$
0i3	-(i+i)$
0T2	-(i+i)$
...	

Semantic Actions during Parsing

BISON (Parser generator) example definition:

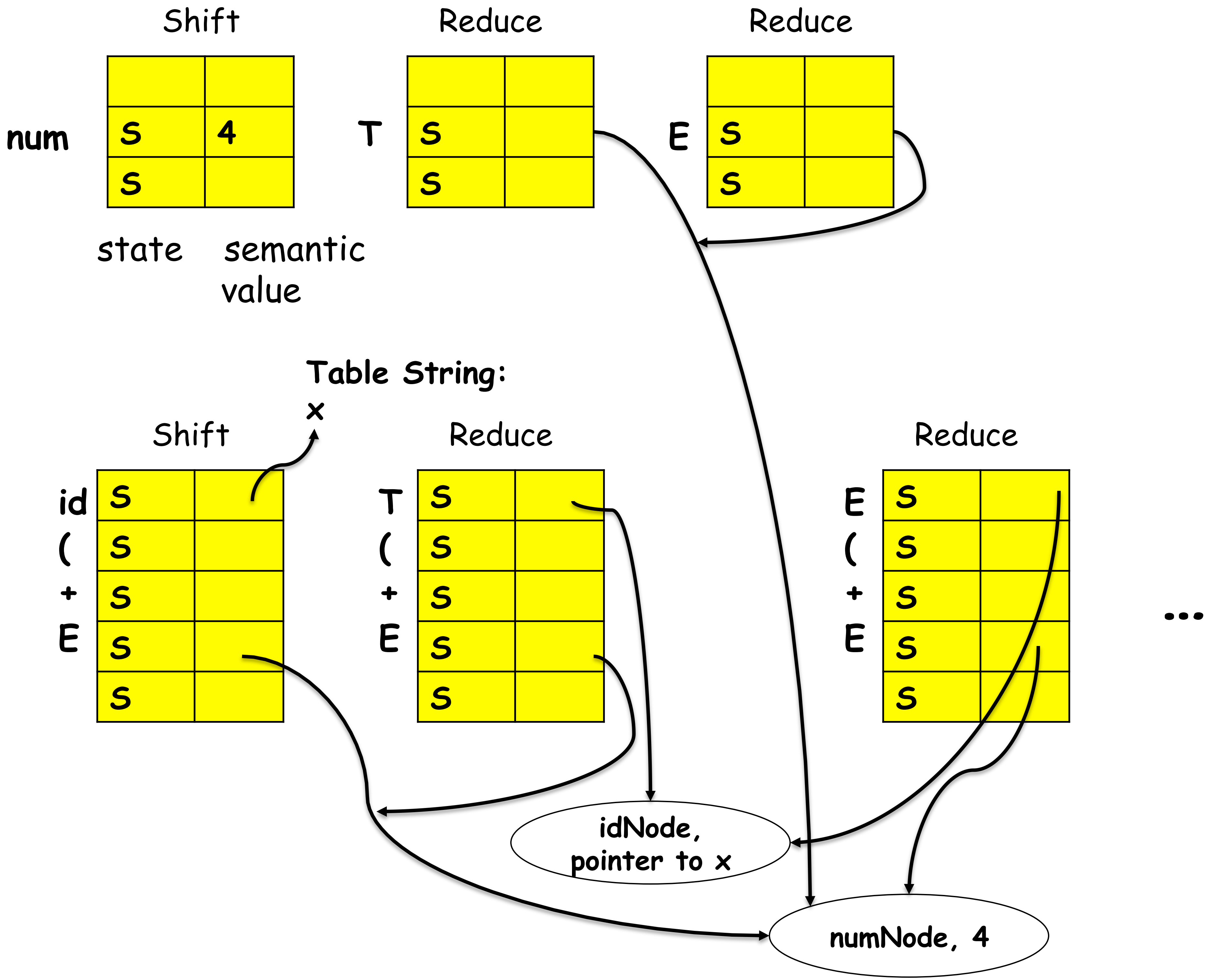
```
S -> E        { $$ = $1;  root = $$; }
E -> E + T    { $$ = makenode( '+' , $1, $3);}  // E is $1, - is $2, T is $3
E -> E - T    { $$ = makenode( '-' , $1, $3);}
E -> T        { $$ = $1;}                        // $$ is top of stack
T -> ( E )    { $$ = $2;}
T -> id       { $$ = makeleaf( 'idnode' , $1);}
T -> num          { $$ = makeleaf( 'numnode' , $1);}
```

Now Consider parsing: **4 + (x - y)**



Parsing Stack

Consider parsing: **4 + (x - y)**



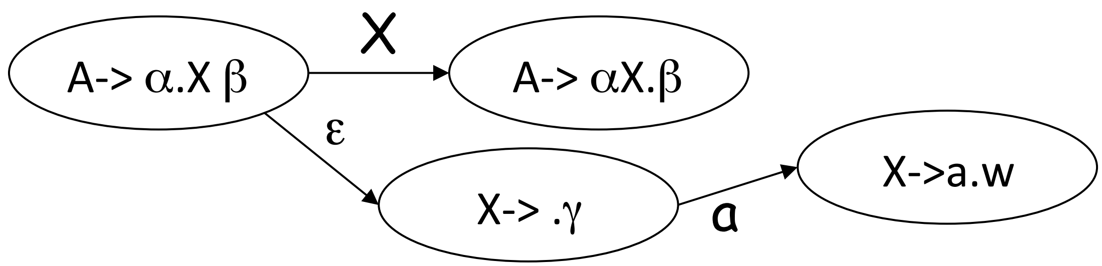
Building LR(0) and SLR(1) Parse Tables

1. Augment grammar
   - Add a production S' -> S, where S is original start state
   - Causes one ACCEPT table entry when reduce S' ->S on $.

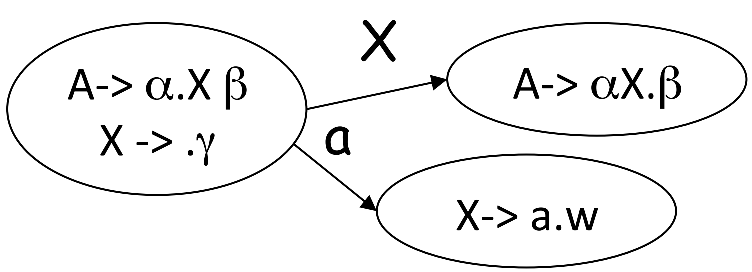
2. Create DFA from grammar

   item A -> α . β
   - just seen a string derivable from α
   - expect to see a string derivable from β

NFA : Each state represents a set of recognized viable prefixes (kernel set of items)



DFA: Subset construction to go from NFA to DFA = closure(kernel)



Items and States

LR(0) item - of a grammar G is a production of G with a dot at some position of the body

For example: A-> XbZ

All possible items are:

A -> . XbZ
A -> X.bZ
A -> Xb.Z
A -> XbZ.

Closure(item set I)

Given a set of kernel items I for a DFA state,

Closure(I) = { kernel items I

if A-> α.Bβ in I and B-> γ
then add B-> . γ to I }

Intuitively, we expect to see strings derivable from all nonterminals immediately to the right of the dot in any item in I.

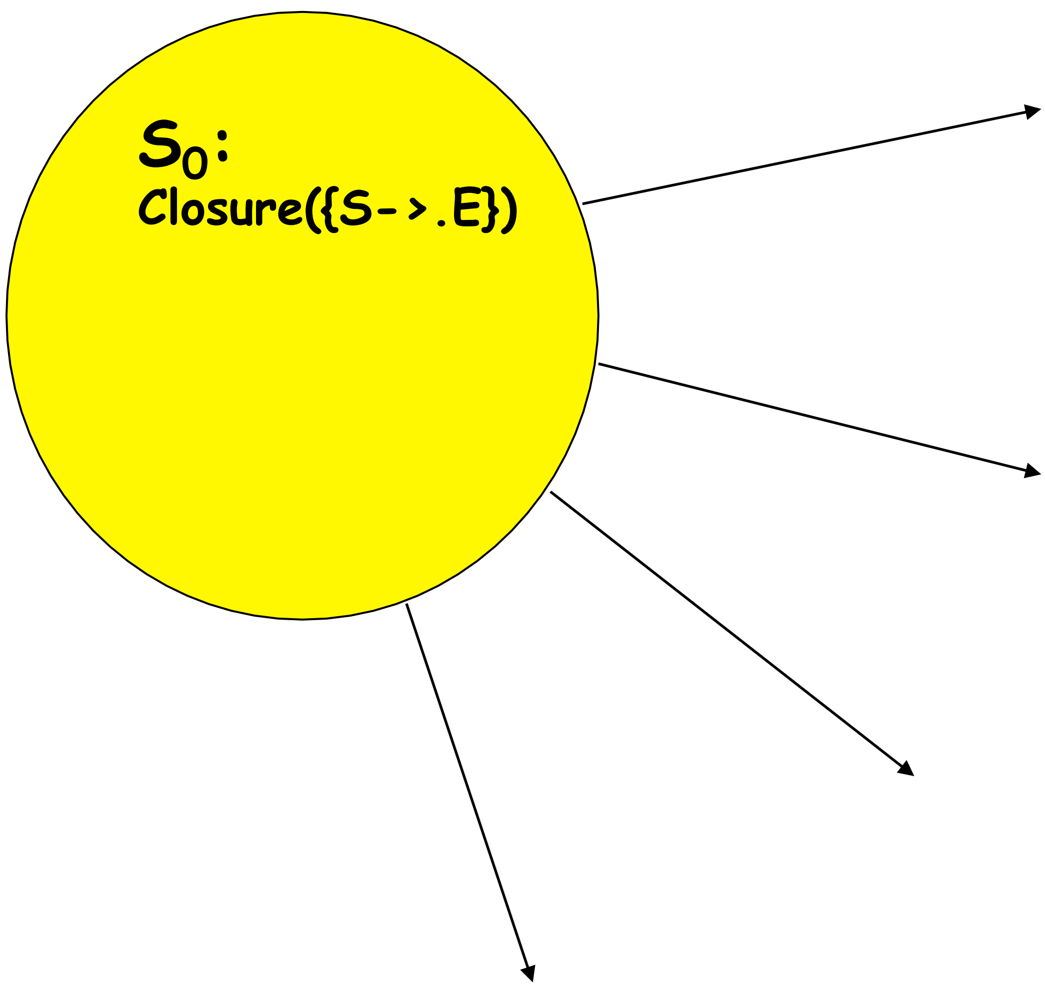
Example: S-> E
E -> T | E + T | E - T
T -> i | (E)

Let I = {S-> .E}
Closure(I)=

Let I = {E-> E+.T}
Closure(I) =

Example of DFA Construction

Grammar: S-> E
E -> T | E + T | E - T
T -> i | (E)



DFA Construction Algorithm

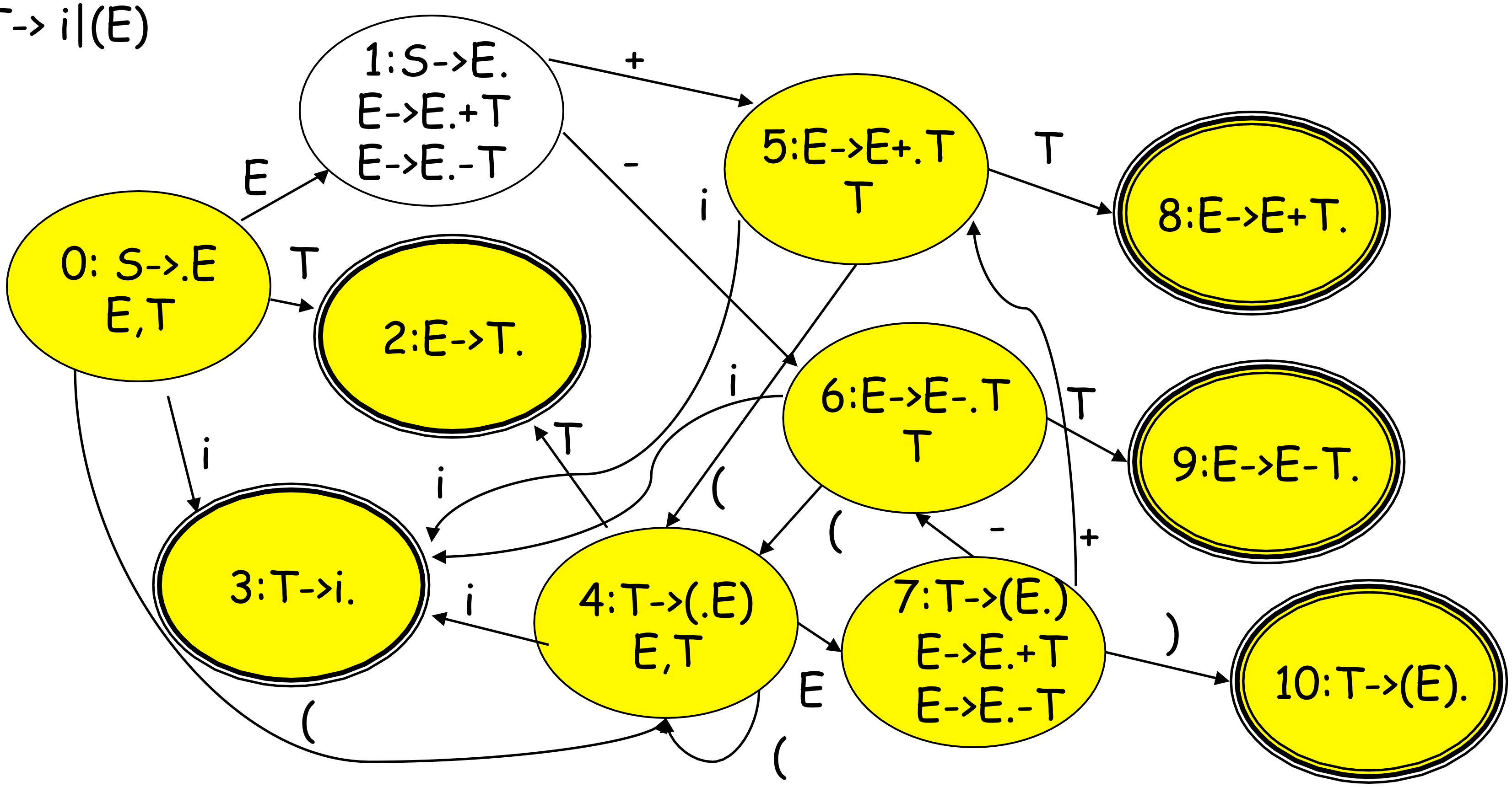
S_0 = Closure({S' -> .S});

Todo = {S_0};

WHILE Todo not empty DO
    Remove an item set (ie, state) Si from Todo;
    FOR each grammar symbol X DO FOR each A ->
        α.Xβ in Si DO
            S_{new} = Closure(A -> αX .β);
            If S_{new} is unique thus far,
                then Add S_{new} to DFA Add S_{new} to
                    Todo;
            Add edge Si -> S_{new} labeled by X

S-> E
E-> T|E+T|E-T
T-> i|(E)

Final DFA for Example

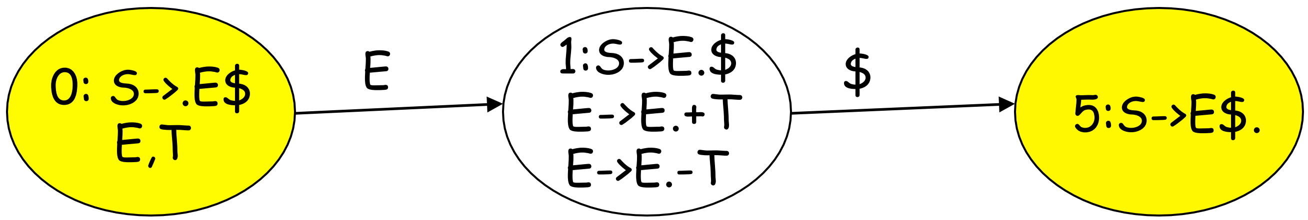


LR(0) grammar = DFA with no inadequate states,
where inadequate state has shift/reduce or reduce/reduce conflict
(e.g., state 1 is inadequate above)

SLR(1) grammar = Can resolve any inadequate states by FOLLOW info:
$A \to \alpha.$ and $B \to \beta.X\delta$ in same state, but FOLLOW(A) $\cap$ {X} is empty.
$A \to \alpha.$ and $B \to \beta.$ in same state, but FOLLOW(A) $\cap$ FOLLOW(B) =0

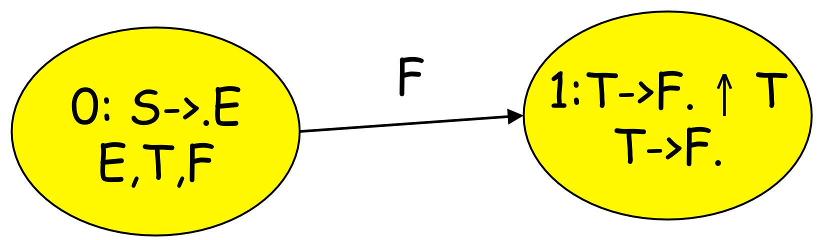
Examples showing LR(0) versus SLR(1)

To convert previous grammar to LR(0): Replace S-> E by S-> E$



Now consider Grammar:

S-> E
E-> E-T | T
T-> F↑T | F
F-> (E) | i

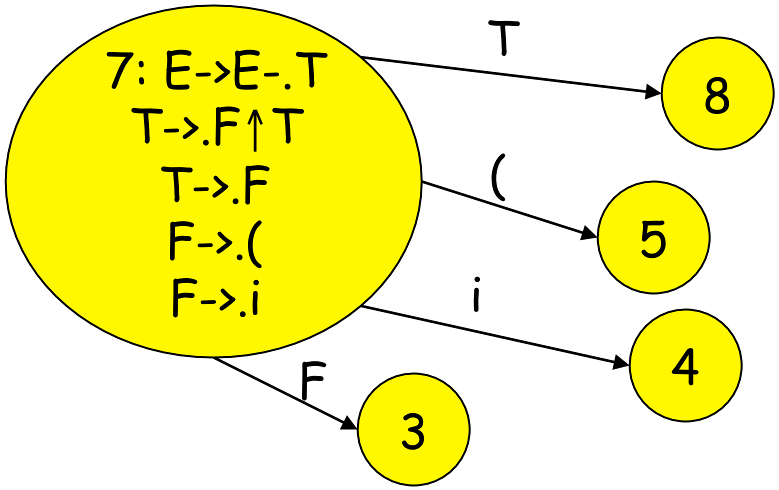


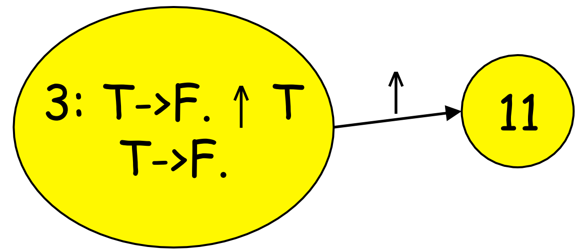
State 1 is inadequate, so **not LR(0)**

Is it SLR(1)?

- FOLLOW(T) = {-,),$}
- FOLLOW(T) ∩ {↑} is empty, so it is **SLR(1) Grammar**

From DFA to SLR(1) Parse Table





	FOLLOW
S	$
E	i,),$
T	-,),$
F	↑, -,),$

state	i	-	↑	(	)	$	S	E	T	F
3										
7										

Is the grammar LR(0),SLR(1)?

LR(0):

- construct parse table with no lookahead/FOLLOW info
  If there are no multidefined entries, then LR(0)

- construct DFA.If there are no inadequate states, then LR(0).

SLR(1):

- construct parse table with FOLLOW info
  If there are no multidefined entries, then SLR(1)

- construct DFA.If there are no inadequate states, or
  for each inadequate state of the form:

$A \rightarrow \alpha.$
$B \rightarrow \beta.$

FOLLOW(A) $\cap$ FOLLOW(B) is empty,

AND

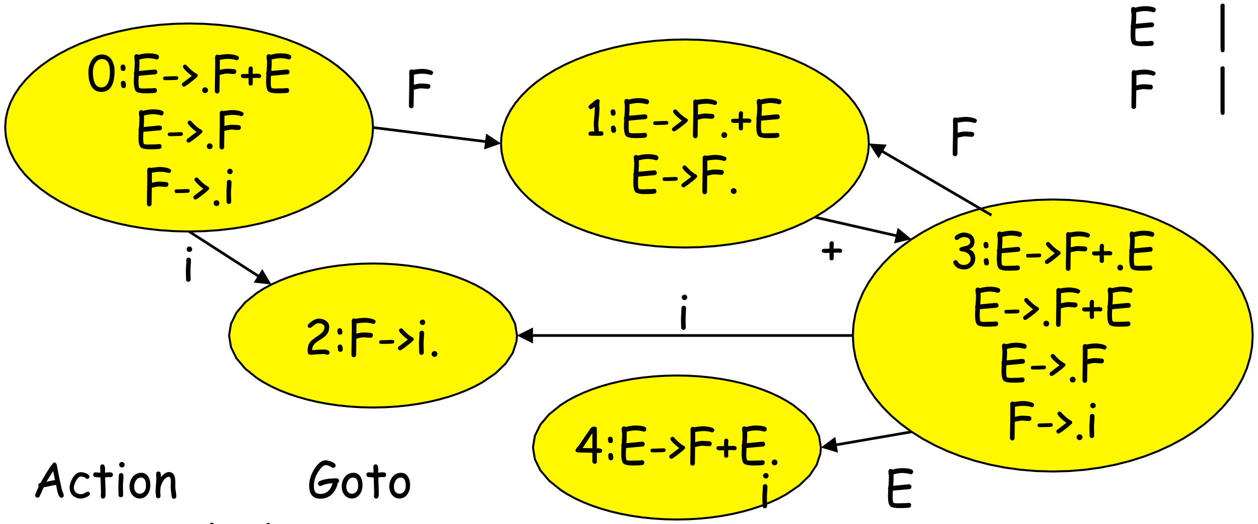
$A \rightarrow \alpha.$
$B \rightarrow \beta.a\gamma$

FOLLOW(A) $\cap$ {a} is empty THEN SLR(1)

Why augment the grammar?

Consider E-> F + E | F
F -> i

	FOLLOW
E	$
F	$, +



	Action			Goto	
	+	i	$	E	F
0		s2			1
1	s3		?		
2	r3		r3		
3		s2		4	1
4			?		

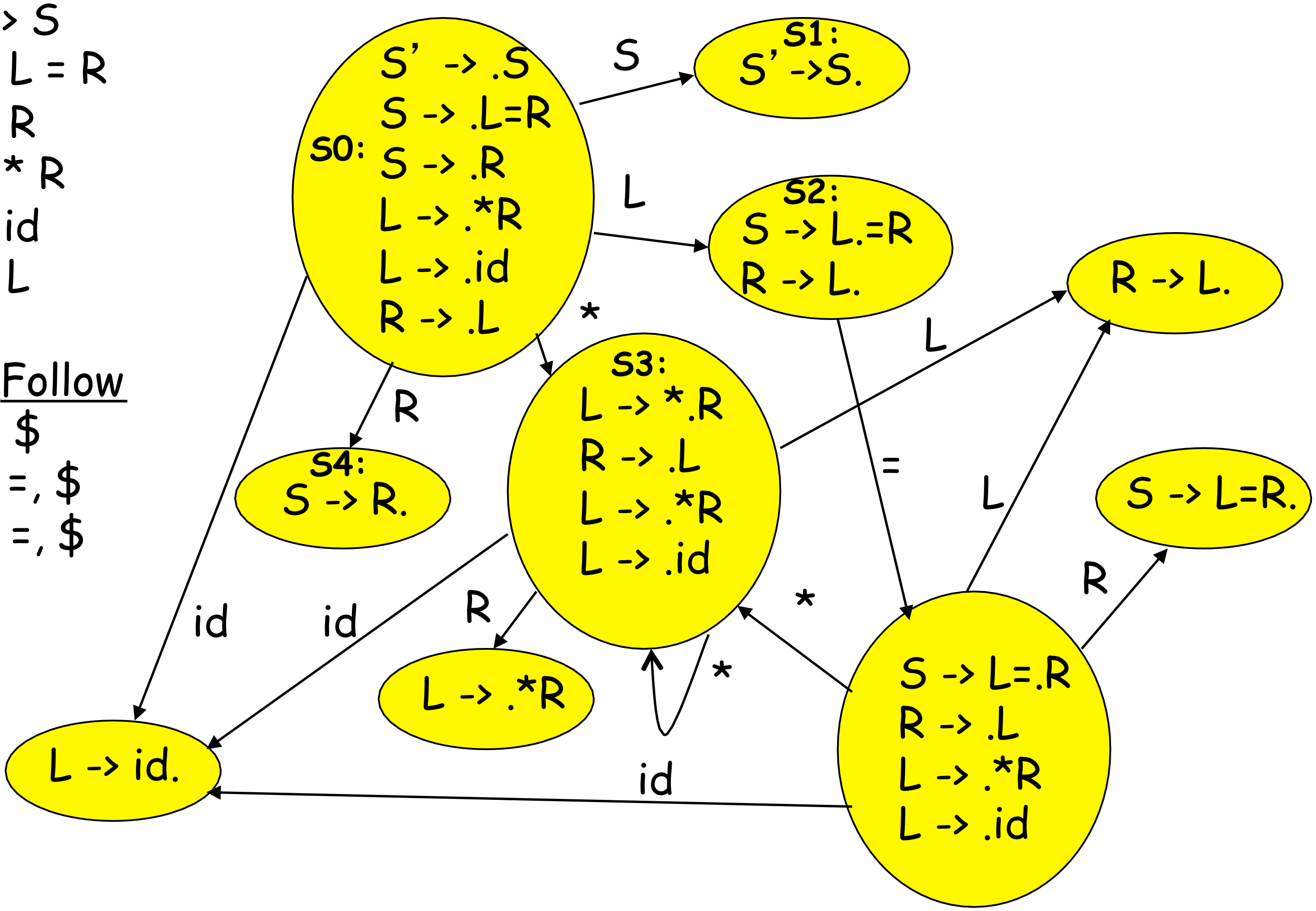
? Reduce E->F or Accept

? Reduce E-> F + E or Accept

Example – not SLR(1)

S' -> S
S -> L = R
S -> R
L -> * R
L -> id
R -> L

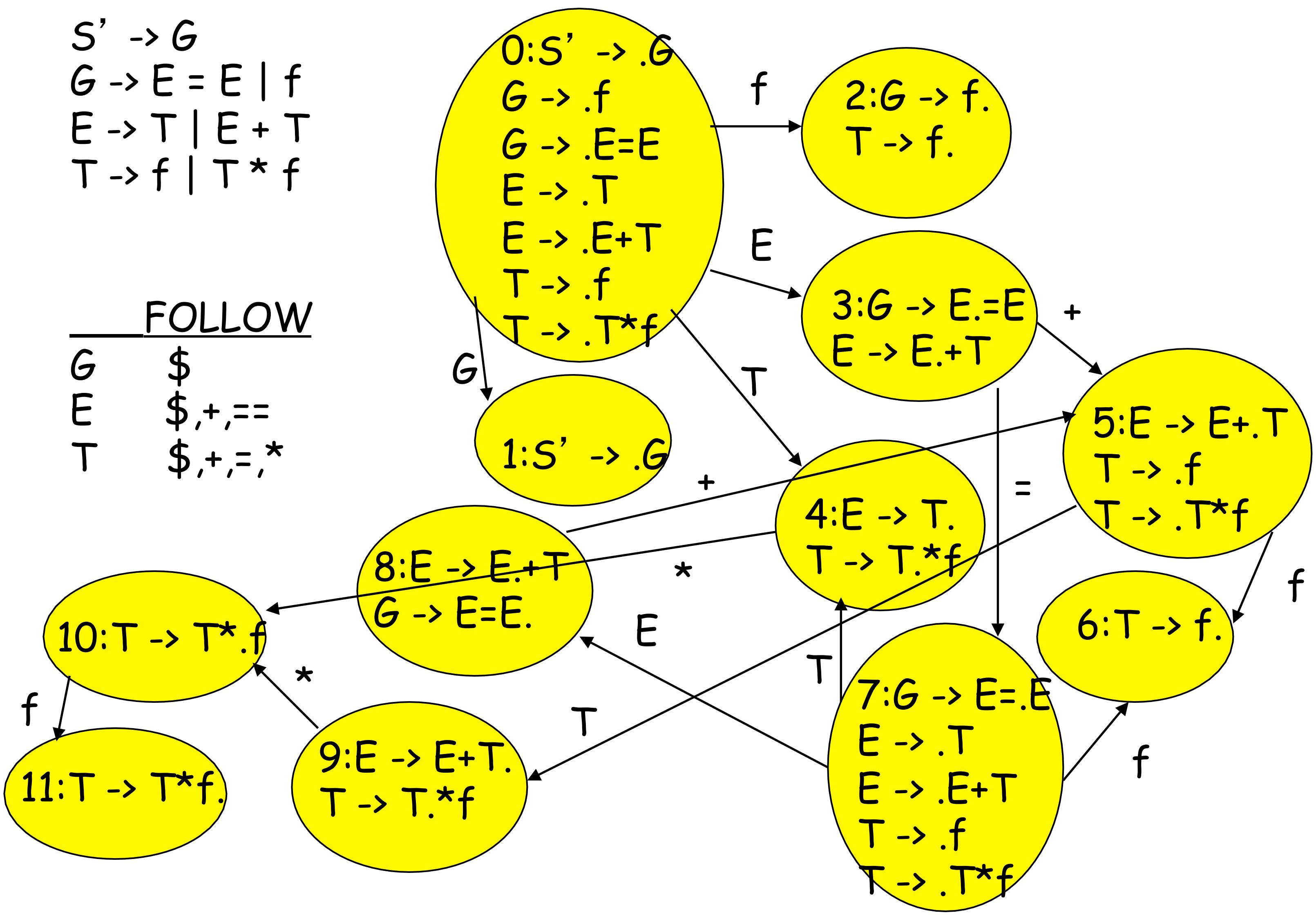
	Follow
S	$
L	=, $
R	=, $



Another Example – not SLR(1)

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

FOLLOW	
G	$
E	$,+,==
T	$,+,=,*



LR(1) Parser (for same grammar)