Parsing (2)

15-411/15-611 Compiler Design

Ben L. Titzer and Seth Copen Goldstein

Feb 13, 2025

Today

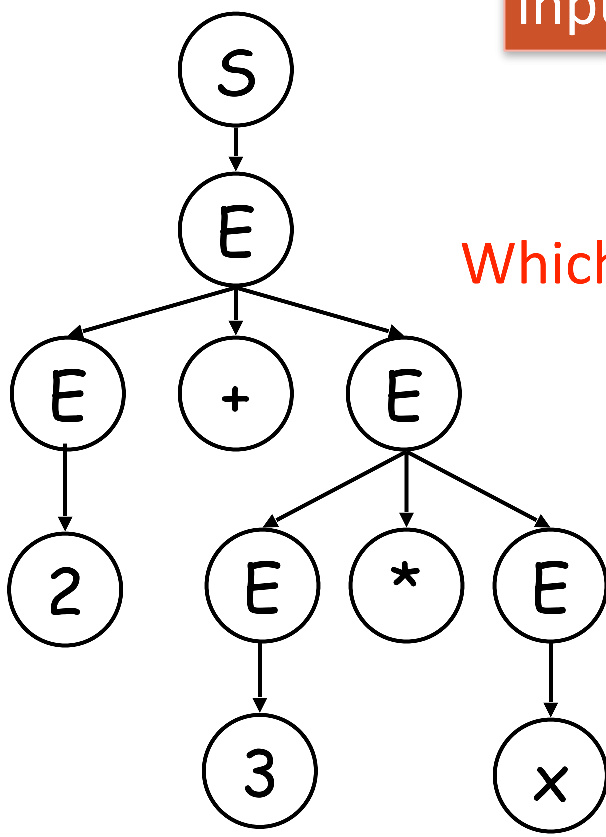
Parsing

- Bottom-up parsers
  - constructing state machine
  - LR0
  - SLR
  - LR(k) & LALR
  - Handling Ambiguity
- Typechecking

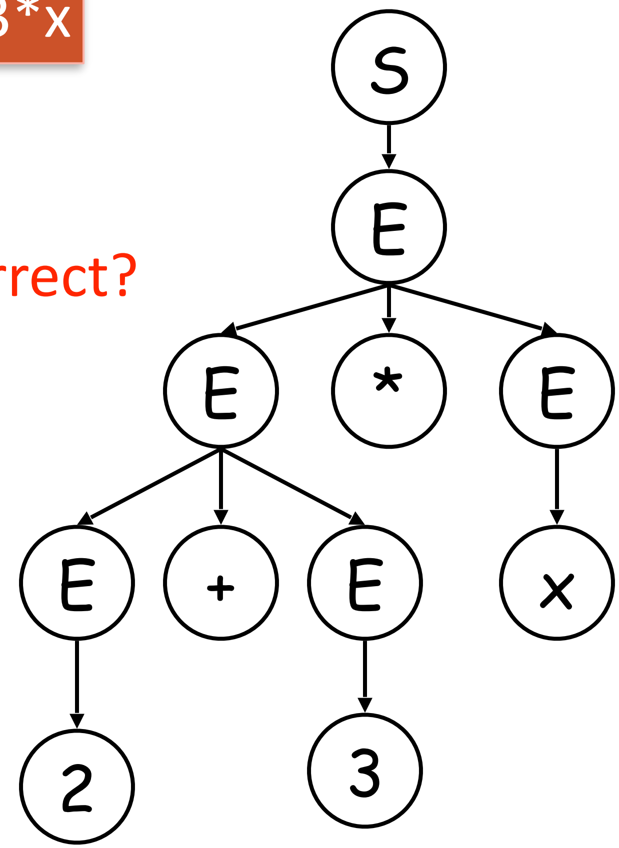
Ambiguity in Grammars

- Some grammars have more than one way to parse a given input, i.e. are ambiguous

input: 2+3*x



Which is correct?



Resolving Ambiguity

- Ambiguity is a problem with the grammar
- One possible fix: Add precedence with more non-terminals
- In this example, one for each level of precedence:
  - (+, -) exp
  - (*, /) term
  - (**id**, **int**) factor
  - Make sure parse derives sentences that respect the precedence
  - Make sure that extra levels of precedence can be bypassed, i.e., “x” is still legal

A Better Exp Grammar

1	S	:= Exp
2	Exp	:= Exp + Term
3	Exp	:= Exp - Term
4	Exp	:= Term
5	Term	:= Term * Factor
6	Term	:= Term / Factor
7	Term	:= Factor
8	Factor	:= **id**
9	Factor	:= **int**

S

by 1 ⇒ Exp

by 2 ⇒ Exp + Term

by 4 ⇒ Term + Term

by 7 ⇒ Factor + Term

by 9 ⇒ $\mathtt{int}_2$ + Term

by 5 ⇒ $\mathtt{int}_2$ + Term * Factor

by 7 ⇒ $\mathtt{int}_2$ + Factor * Factor

by 9 ⇒ $\mathtt{int}_2 + \mathtt{int}_3$ * Factor

by 8 ⇒ $\mathtt{int}_2 + \mathtt{int}_3 * \mathtt{id}_x$

What is the parse tree?

Parsing a Grammar

- Top-Down
  - start at root of parse-tree
  - pick a production and expand to match input
  - may require backtracking
  - if no backtracking required, predictive
- Bottom-up
  - start at leaves of tree
  - recognize valid prefixes of productions
  - consume input and change state to match
  - use stack to track state

Computing FIRST(α)

- Given X := A B C, FIRST(X) = FIRST(A B C)
- Can we ignore B or C?
- Consider:

```
A := a
   |
B := b
   | A
C := c
```

Computing FIRST(α)

- Given X := A B C, FIRST(X) = FIRST(A B C)
- Can we ignore B or C?
- Consider:

    A := a
    A :=
    B := b
    B := A
    C := c

- FIRST(X) must also include FIRST(C)
- IOW:
  - Must keep track of NTs that are nullable
  - For nullable NTs, determine FOLLOWS(NT)

nullable(A)

- nullable(A) is
  - true if A can derive the empty string
  - false otherwise
- For example:

B := X Y b

X := x

| Y Y

Y :=

In this case, nullable(X) = nullable(Y) = true

nullable(B) = false

FOLLOW(A)

- FOLLOW(A) is the set of terminals that can immediately follow A in a sentential form.
- I.e.,
  $a \in$ FOLLOW(A) iff $S \Rightarrow^* \alpha A a \beta$ for some α and β

Bottom-up parsers

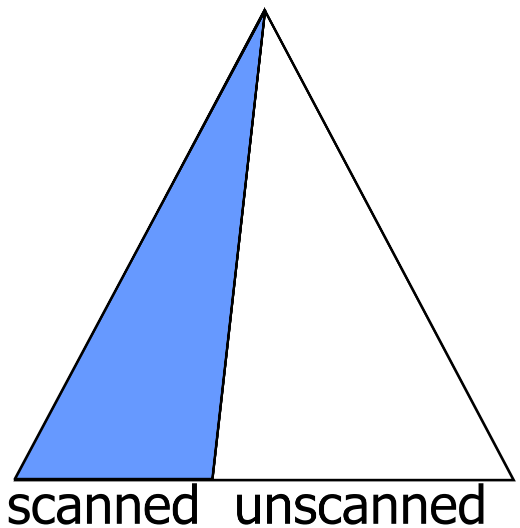
- What is the inherent restriction of top-down parsing, e.g., with LL(k) grammars?

Bottom-up parsers

- What is the inherent restriction of top-down parsing, e.g., with LL(k) grammars?
- Bottom-up parsers use the entire right-hand side of the production
- LR(k):
  - Left-to-right parse,
  - Rightmost derivation (in reverse),
  - k look ahead tokens

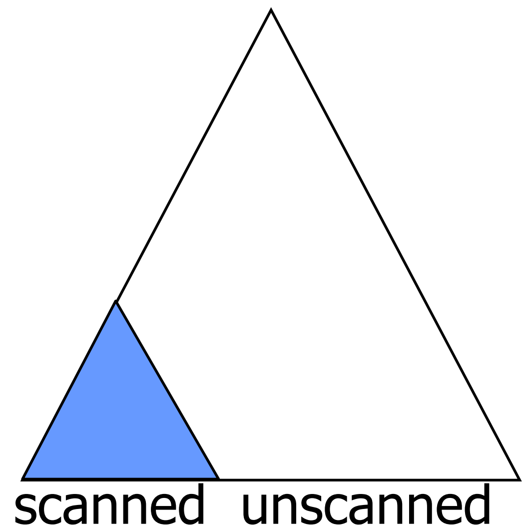
Top-down vs. Bottom-up

LL(k), recursive descent



Top-down

LR(k), shift-reduce



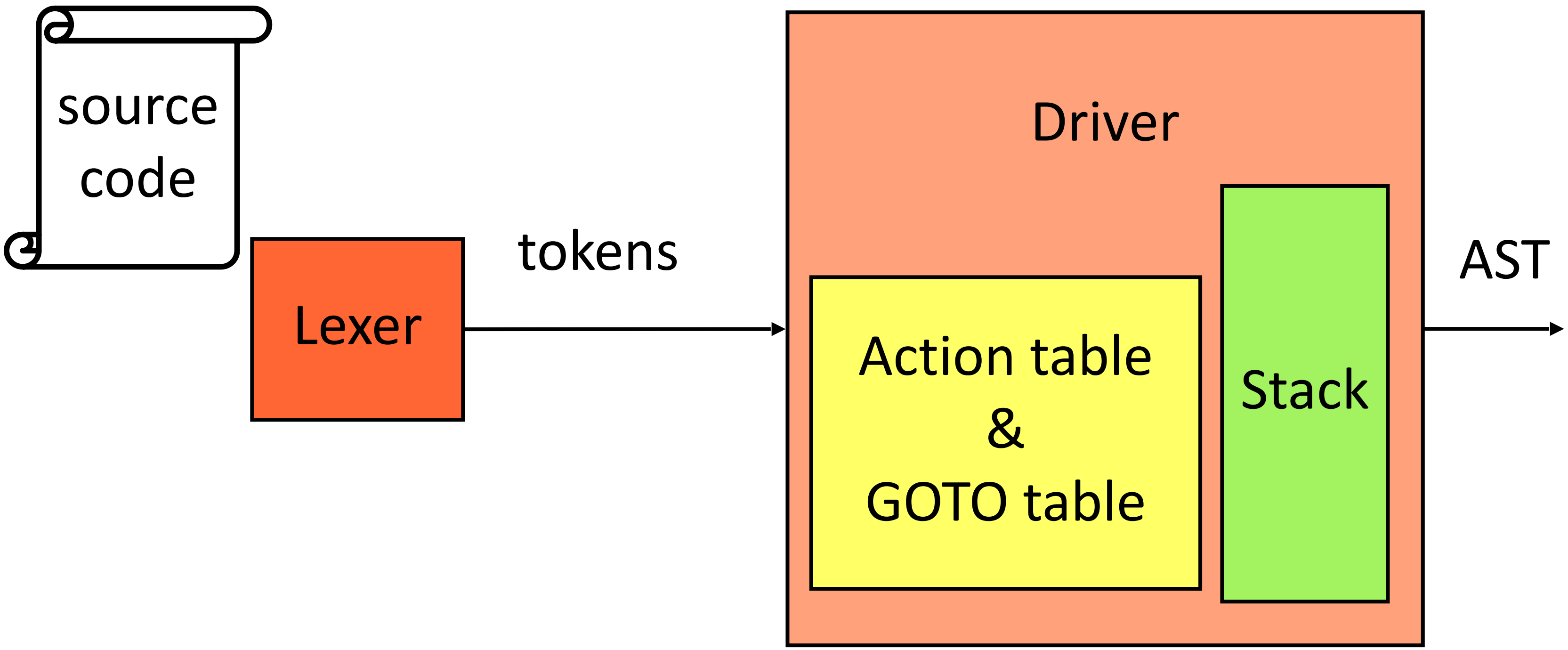
Bottom-up

A Shift-Reduce Parser

- Implement as a FSM with a stack
- Stack holds sequences of symbols
- Input stream holds remaining source
- Four actions:
  - shift: push token from input stream onto stack
  - reduce: right-end of a handle (β of $A \rightarrow \beta$) is at top of stack, pop handle (β), push A
  - accept: success
  - error: syntax error discovered

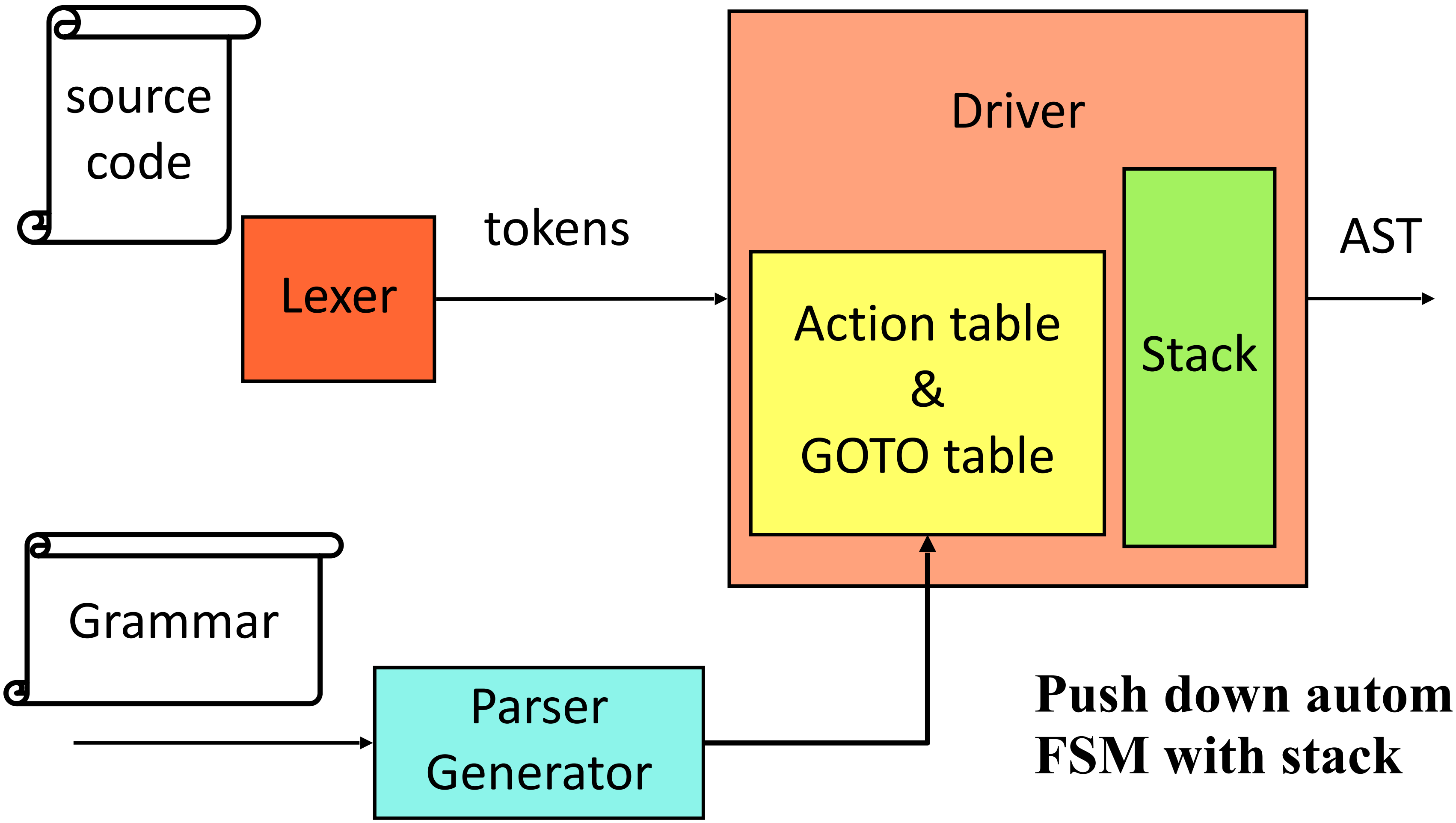
Key is recognizing handles efficiently

Table-driven LR(k) parsers



Push down automata: FSM with stack

Table-driven LR(k) parsers



Push down automata: FSM with stack

Parser Loop

Driver

- Same code regardless of grammar
  - only tables change
- (Very) General Algorithm:
  - Based on table contents, top of stack, and current input token either
    - **shift**: push token onto stack and read next token
    - **reduce**: replace part of stack with the correct rule (NT) that derived it
    - **accept**: successfully parsed entire input
    - **error**: input not in language

Stack

Stack

- Represents the input parsed so far
- Contents?
  - Symbols: terminals (and non-terminals)
  - Must also store previously seen *states*
    - the context of the current position
  - In fact, nonterminals unnecessary
    - include for readability

$x + y \bullet + z$

T

+

T

Parser Tables

Action table
&
GOTO table

Action table

- given state *s* and **terminal** *a* tells parser loop what action (shift, reduce, accept, reject) to perform

Goto table

- used when performing reduction; given a state *s* and **nonterminal** *X* says what state to transition to

Parser Tables

Action table
&
GOTO table

s*N* push state *N* onto stack
r*R* reduce by rule *R*
g*N* goto state *N*
a accept
error

state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4	r2		
3		r3	r3		
4	s3			g5	g2
5			r1		

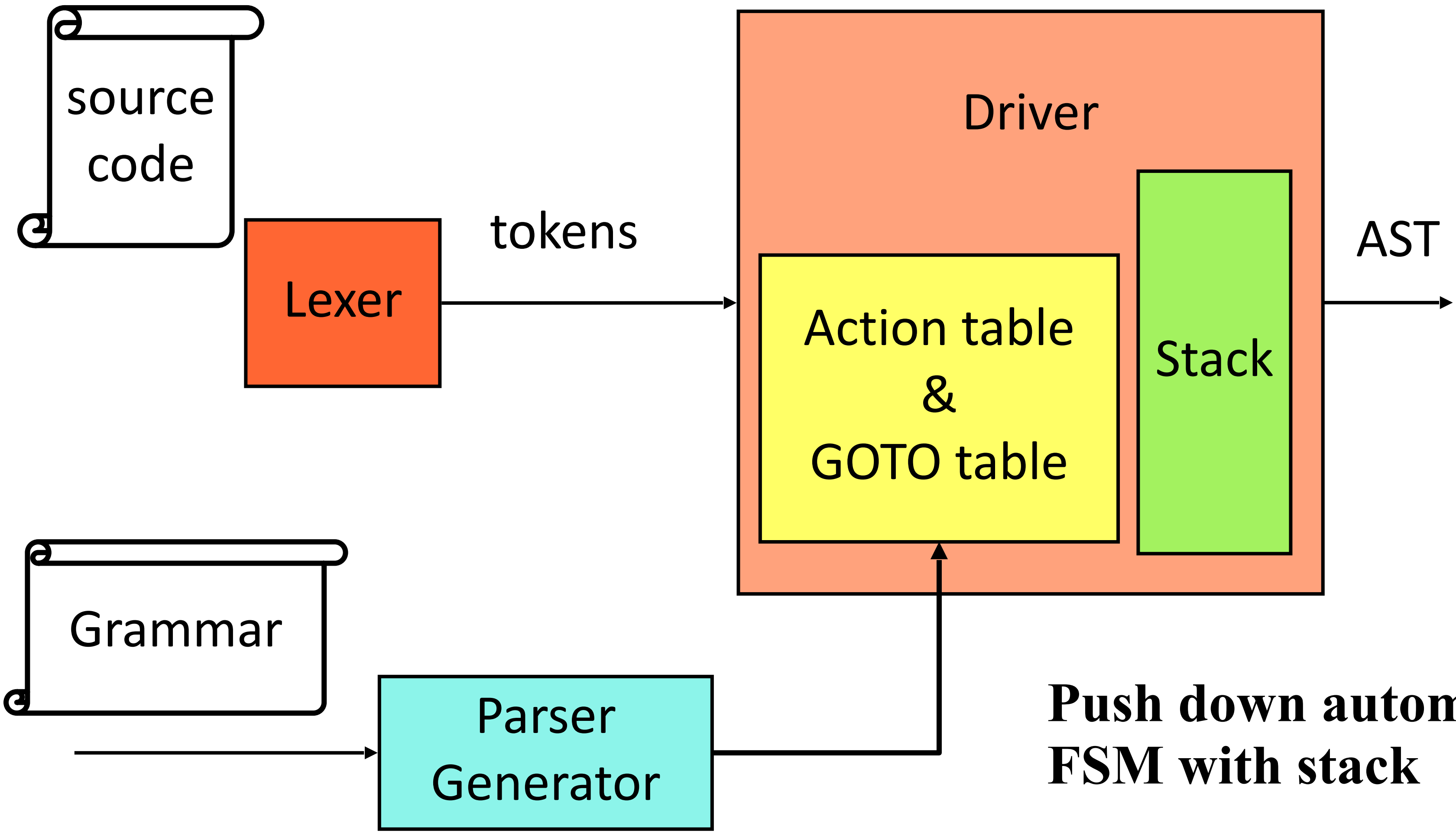
0 $S \longrightarrow E\$$
1 $E \longrightarrow T + E$
2 $E \longrightarrow T$
3 $T \longrightarrow \mathit{identifier}$

Parser Loop Revisited

Driver

```
while(true)
   s = state on top of stack
   a = current input token
   if(action[s][a] == sN)                          shift
      push N
      read next input token
   else if(action[s][a] == rR)                     reduce
      pop rhs of rule R from stack
      X = lhs of rule R
      N = state on top of stack
      push goto[N][X]
   else if(action[s][a] == a)                      accept
      return success
   else                                  error
      return failure
```

Table-driven LR(k) parsers



Push down automata: FSM with stack

The parser generator

Parser Generator

- Creates the action and GOTO tables.
- Creates the states
  - Each state indicates how much of a handle we have seen
  - each state is a set of *items*

Items

- Items are used to identify handles.
- LR(k) items have the form:

  [production-with-dot, lookahead]

- For example, $A \rightarrow a\ X\ b$ has 4 LR(0) items
  - $[A \rightarrow \bullet a\ X\ b]$
  - $[A \rightarrow a \bullet X\ b]$
  - $[A \rightarrow a\ X \bullet b]$
  - $[A \rightarrow a\ X\ b \bullet]$

The ● indicates how much of the handle we have recognized.

What LR(0) Items Mean

- [$X \rightarrow \bullet\ \alpha\ \beta\ \gamma$]
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$
- [$X \rightarrow \alpha \bullet \beta\ \gamma$]
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and we have already recognized α
- [$X \rightarrow \alpha\ \beta \bullet \gamma$]
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and we have already recognized $\alpha\ \beta$
- [$X \rightarrow \alpha\ \beta\ \gamma \bullet$]
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and we can reduce to X

Generating the States

- Start with start production.
- In this case, "S → E$"

$\boxed{S \rightarrow \bullet E\$}$

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

- Each state is consistent with what we have already shifted from the input and what is possible to reduce. So, what other items should be in this state?

Completing a state

- For each item in a state, add in all other consistent items.

$S \rightarrow \bullet E\$$

$E \rightarrow \bullet T + E$

$E \rightarrow \bullet T$

$T \rightarrow \bullet \mathit{identifier}$

0 $S \rightarrow E\$$

1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow \mathit{identifier}$

- This is called, taking the closure of the state.

Closure*

```
closure(state)
   repeat
      foreach item A → a•Xb in state
         foreach production X → w
            state.add(X → •w)
   until state does not change
   return state
```

Intuitively:

Given a set of items, add all production rules that could produce the nonterminal(s) at the current position in each item

*: for LR(0) items

What about the other states?

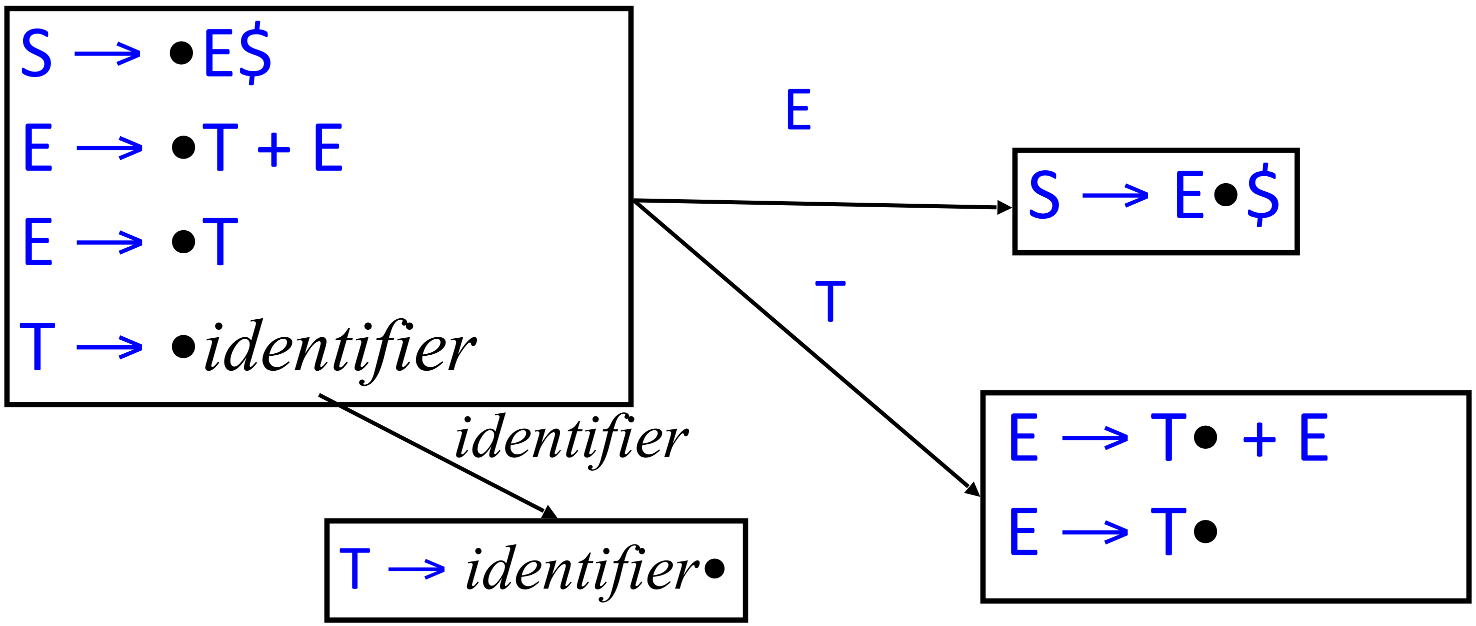
- How do we decide what the other states are?
- How do we decide what the transitions between states are?

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*



Next(state, sym)

- Next function determines what state to goto based on current state and symbol being recognized.
- For Non-terminal, this is used to determine the GOTO table.
- For terminal, this is used to determine the shift action.

Constructing states

```
initial_state = closure({start production})
state_set.add(initial_state)
state_queue.push(initial_state)

while(!state_queue.empty())
   s = state_queue.pop()
   foreach item A → a•Xb in s
      n = closure(next(s, X))
      if(!state_set.contains(n))
         state_set.add(n)
         state_queue.push(n)
```

A state is a set of LR(0) items

get "next" state

Closure*

closure({$S \longrightarrow \bullet E\$$}) =

$S \longrightarrow \bullet E\$$

0 $S \longrightarrow E\$$
1 $E \longrightarrow T + E$
2 $E \longrightarrow T$
3 $T \longrightarrow \textit{identifier}$

*: for LR(0) items

Closure*

closure({$S \rightarrow \bullet E\$$}) =

$S \rightarrow \bullet E\$$

$E \rightarrow \bullet T + E$

$E \rightarrow \bullet T$

$T \rightarrow \bullet \mathit{identifier}$

0 $S \rightarrow E\$$

1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow \mathit{identifier}$

*: for LR(0) items

Next

```
next(state, X)
    ret = empty
    foreach item A → a•Xb in state
        ret.add(A → aX•b)
    return ret
```

0 $S \rightarrow E\$$

1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow$ *identifier*

initial:

$S \rightarrow \bullet E\$$

$E \rightarrow \bullet T + E$

$E \rightarrow \bullet T$

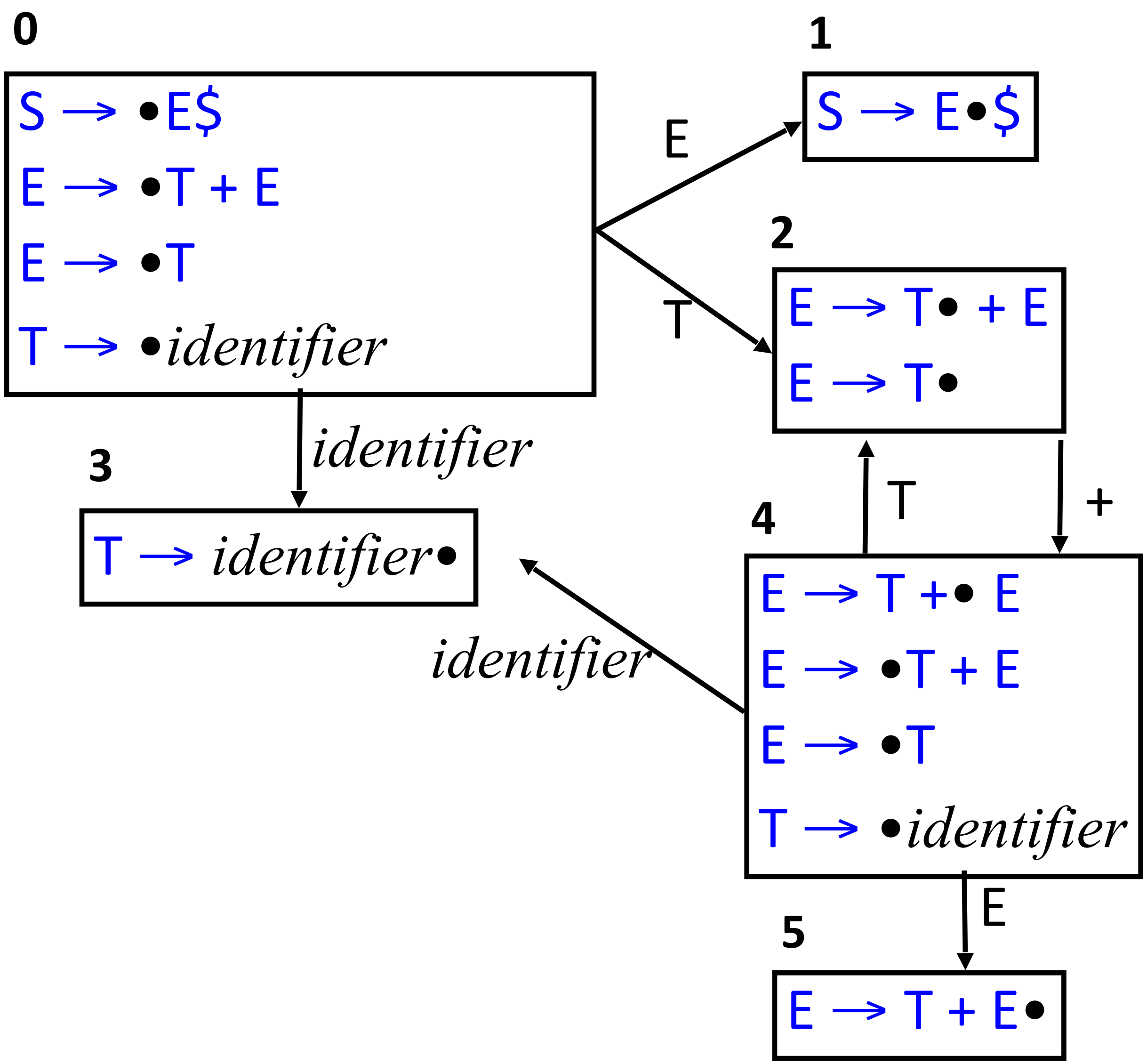
$T \rightarrow \bullet$*identifier*

next(initial, E)

next(initial, T)

next(initial, *identifier)*

Example



0 $S \rightarrow E\$$

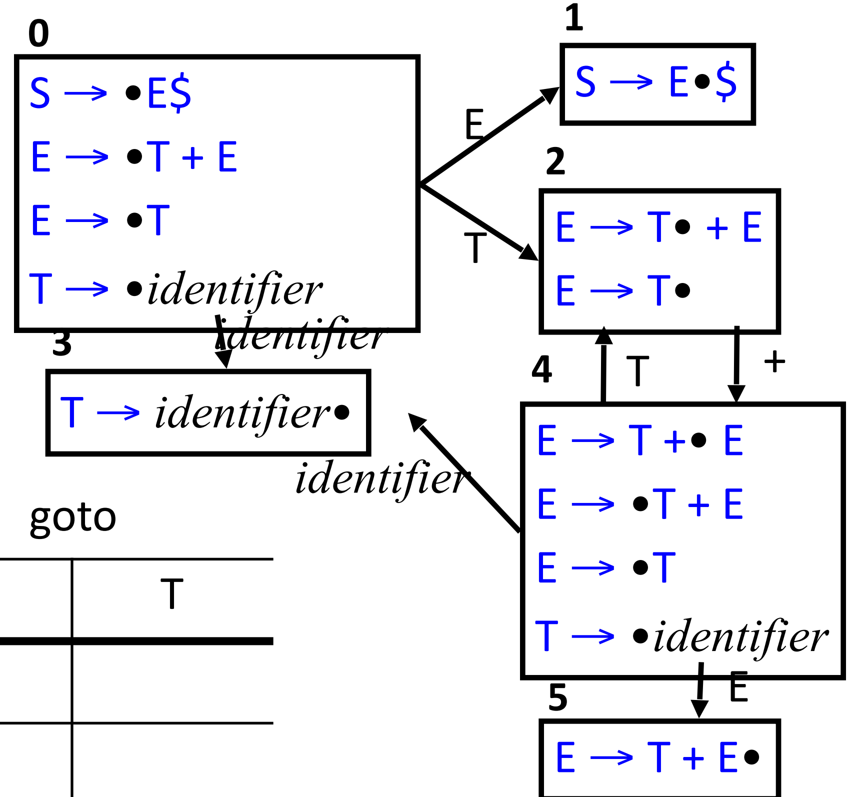
1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow \textit{identifier}$

Parse Tables for LR(0) parser

What can we fill out?



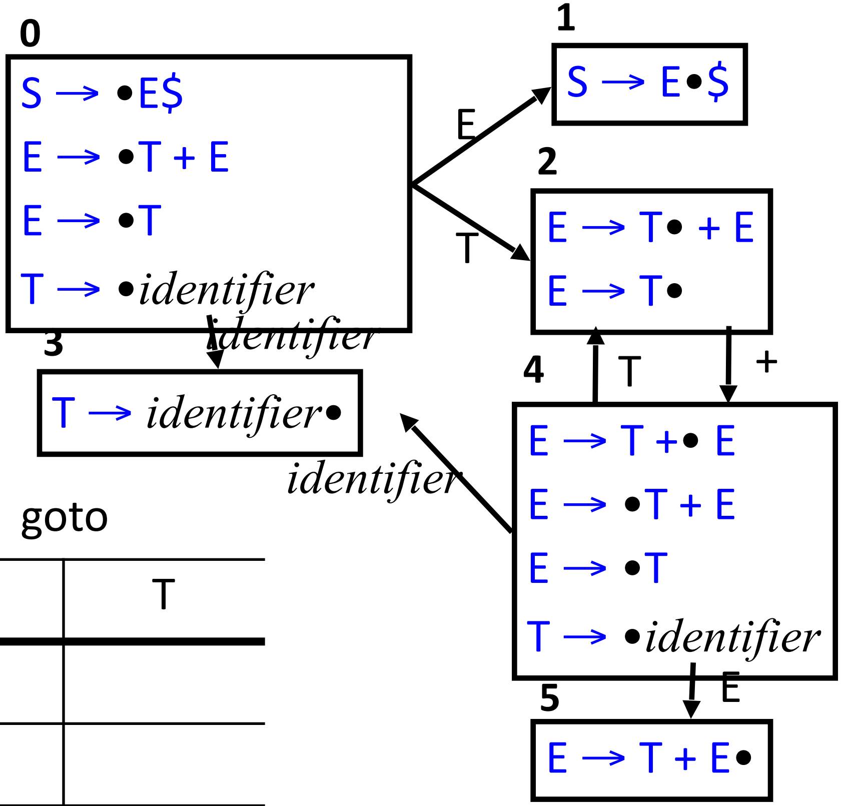
state	action			goto	
	ident	+	$	E	T
0					
1					
2					
3					
4					
5					

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Parse Tables for LR(0) parser

shift

transition on terminal



state	action			goto	
	ident	+	$	E	T
0	s3				
1					
2		s4			
3					
4	s3				
5					

0 S → E$

1 E → T + E

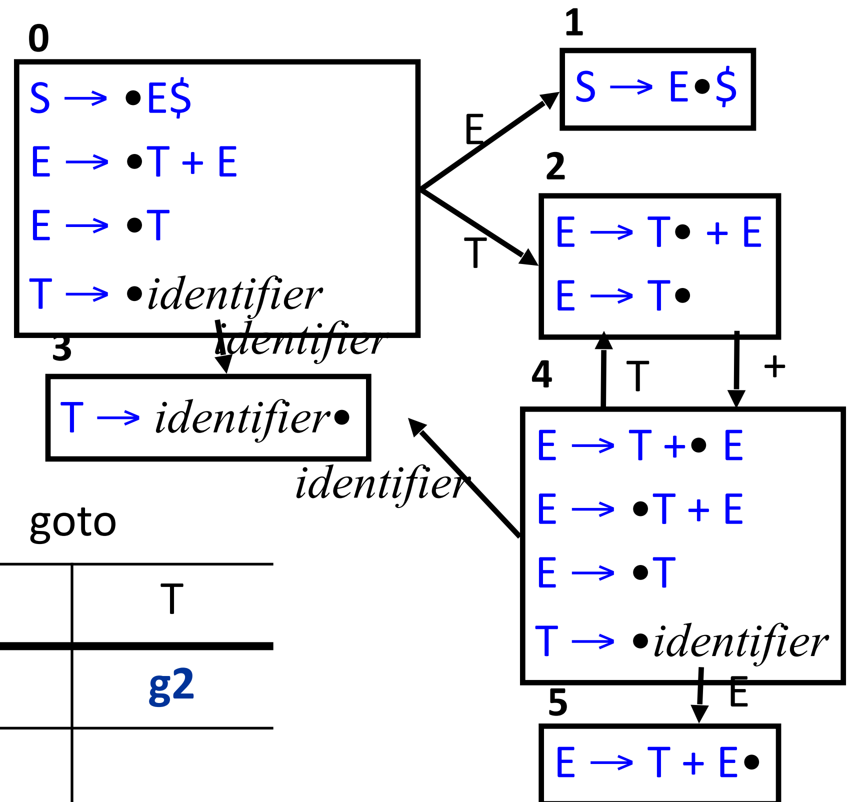
2 E → T

3 T → *identifier*

Parse Tables for LR(0) parser

goto

transition on nonterminal



state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1					
2		s4			
3					
4	s3			g5	g2
5					

0 S → E$

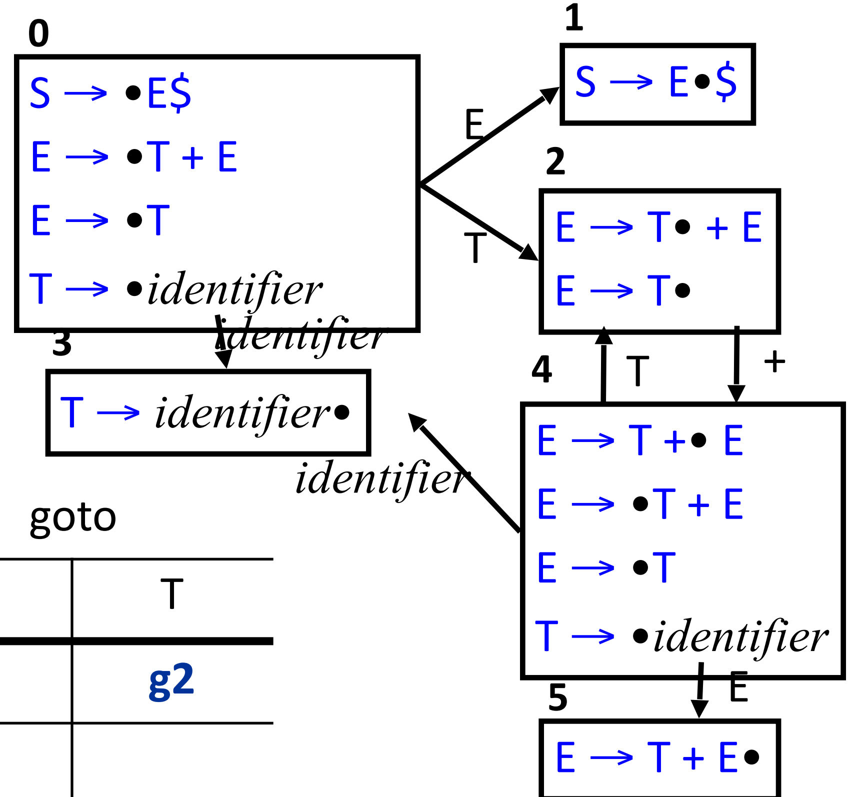
1 E → T + E

2 E → T

3 T → *identifier*

Parse Tables for LR(0) parser

accept
about to shift $



state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4			
3					
4	s3			g5	g2
5					

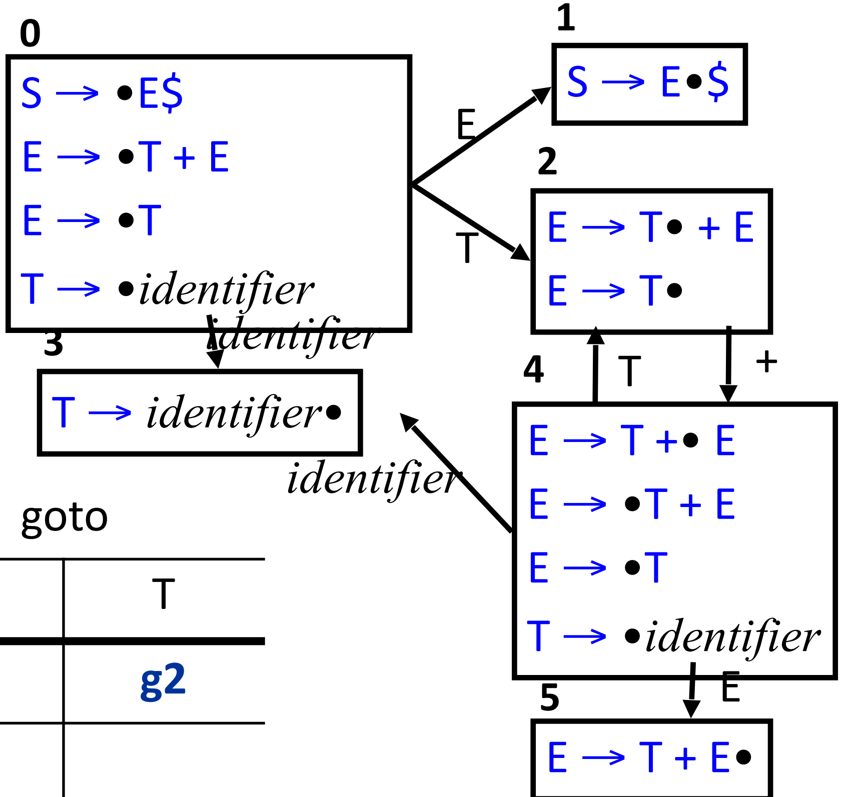
0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

Parse Tables for LR(0) parser

reduce

item has dot at end

$A \rightarrow w\bullet$



state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2		s4			
3					
4	s3			g5	g2
5					

0 $S \rightarrow E\$$

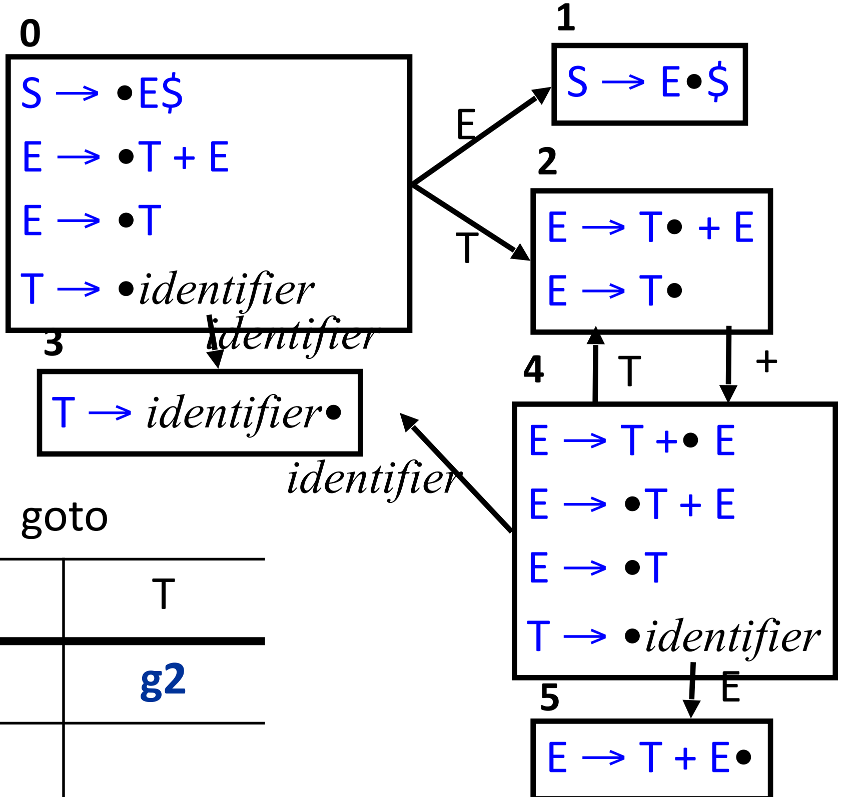
1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow$ *identifier*

LR(0)

No lookahead
reduce state for *all* nonterminals

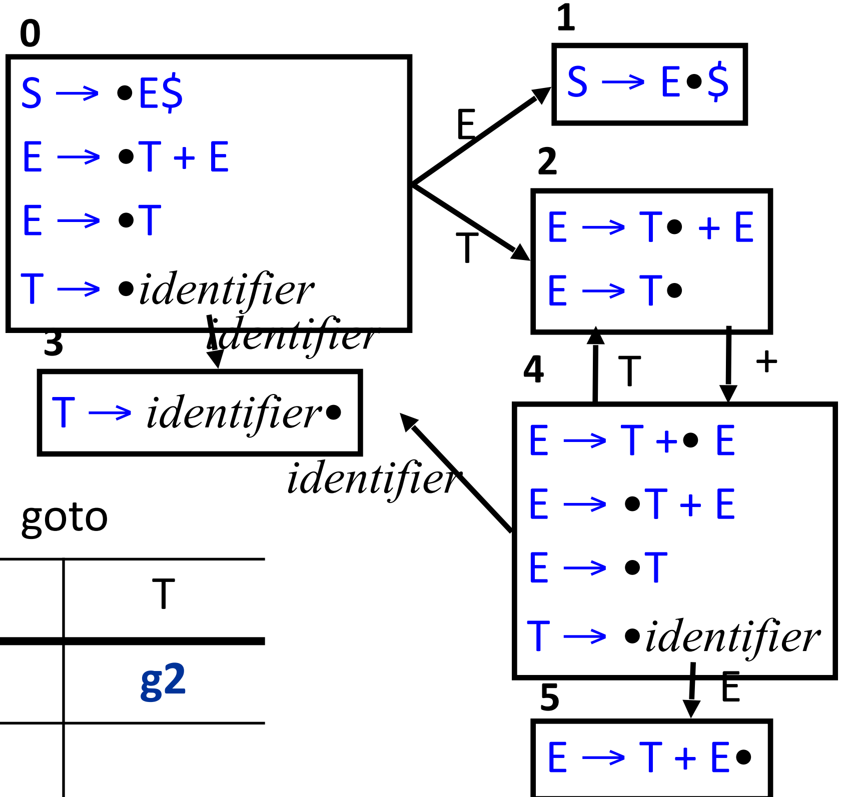


state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2	r2	r2/s4	r2		
3	r3	r3	r3		
4	s3			g5	g2
5	r1	r1	r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

LR(0)

shift/reduce conflict
need to be pickier about
when we reduce

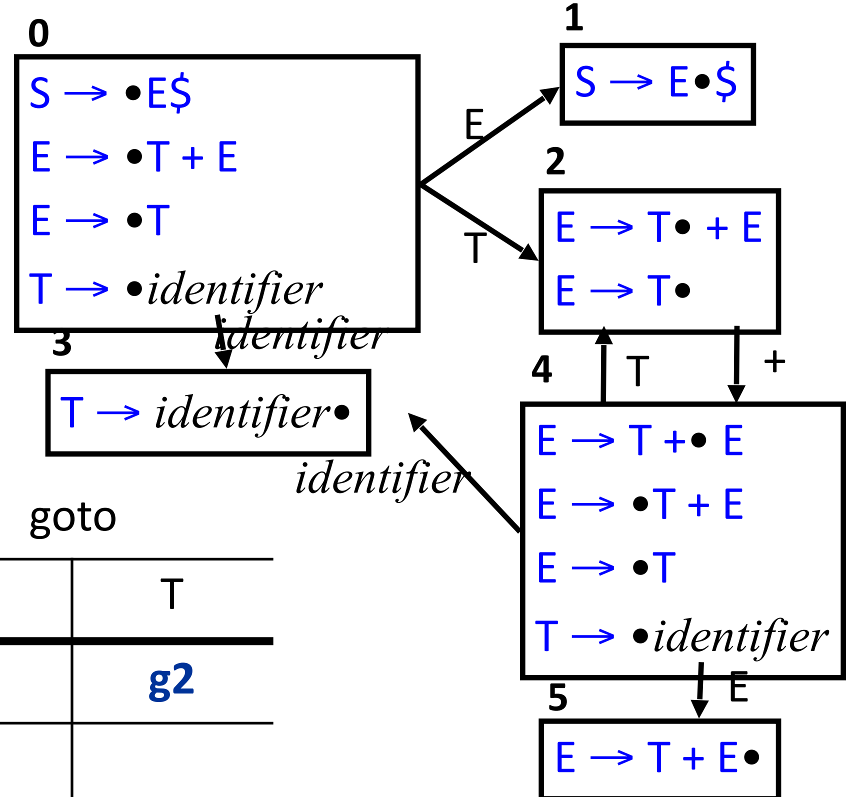


state	action			goto	
	ident	+	$	E	T
0	s3			g1	g2
1			a		
2	r2	r2/s4	r2		
3	r3	r3	r3		
4	s3			g5	g2
5	r1	r1	r1		

0 S → E$
1 E → T + E
2 E → T
3 T → *identifier*

SLR - Simple LR

Only reduce in position (**s**,**a**) by rule R:A $\longrightarrow$ *w* if **a** is in the *follow set* of A



	action			goto	
state	*ident*	+	$	E	T
0	s3			g1	g2
1			a		
2		s4			
3					
4	s3			g5	g2
5					

0 S $\longrightarrow$ E$

1 E $\longrightarrow$ T + E

2 E $\longrightarrow$ T

3 T $\longrightarrow$ *identifier*

Reminder: Follow sets

FOLLOW(X)

set of terminals that can appear immediately after the nonterminal X in some sentential form

0 $S \rightarrow E\$$

1 $E \rightarrow T + E$

2 $E \rightarrow T$

3 $T \rightarrow \mathit{identifier}$

I.e., $t \in \mathrm{FOLLOW}(X)$ iff $S \Rightarrow^* \alpha X t \beta$ for some α and β

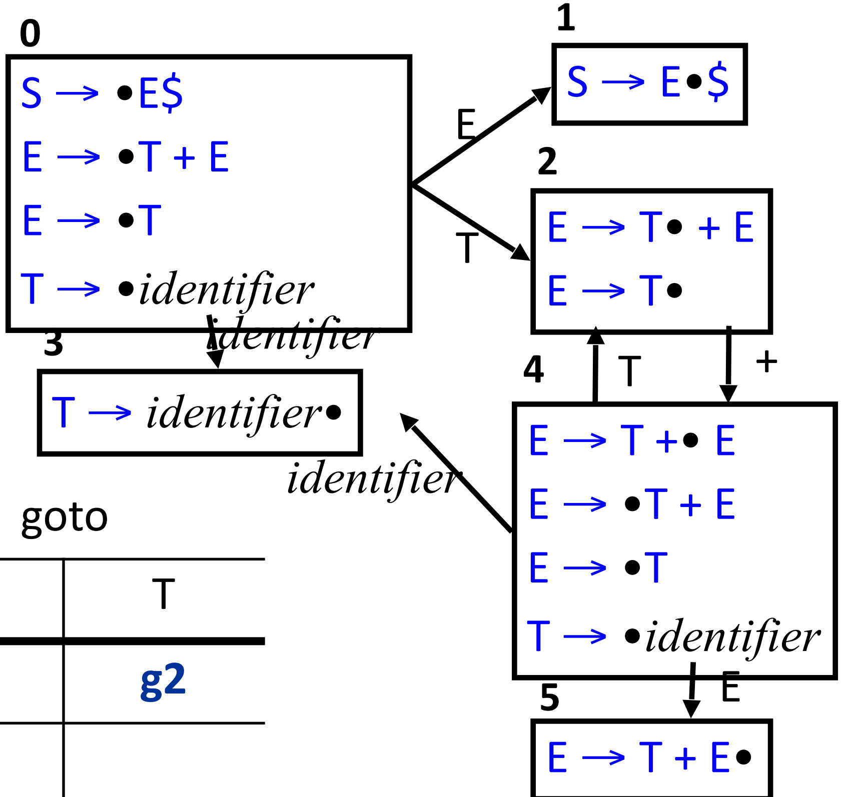
FOLLOW(E**) = {\$}**

FOLLOW(T**) = {+,\$}**

SLR - Reduce using follow sets

FOLLOW(E) **=** {$}

FOLLOW(T) **=** {+,$}



state	action			goto	
	ident	+	$	E	T
0	**s3**			**g1**	**g2**
1			**a**		
2		**s4**	**r2**		
3		**r3**	**r3**		
4	**s3**			**g5**	**g2**
5			**r1**		

0 S → E$

1 E → T + E

2 E → T

3 T → *identifier*

SLR Limitations

- SLR uses LR(0) item sets
- Can remove some (but not all) shift/reduce conflicts using follow set
- Doesn't fix all problems

Problem with SLR

- Reduce on ALL terminals in FOLLOW set

$$\boxed{\begin{array}{lll} S & \rightarrow & L = R \\ & | & R \\ L & \rightarrow & * \, R \\ & | & \mathbf{id} \\ R & \rightarrow & L \end{array}}$$

$$2 \quad \boxed{\begin{array}{l} S \rightarrow L \bullet \quad = R \\ R \rightarrow L \bullet \end{array}}$$

- FOLLOW(R) = FOLLOW(L)
- But, we should never reduce R → L on ‘=‘
  I.e., R=… is not a viable prefix for a right sentential form
- Thus, there should be no reduction in state 2
- How can we solve this?

LR(1) Items

- An LR(1) item is an LR(0) item combined with a single terminal (the *lookahead*)
- $[X \rightarrow \alpha \bullet \beta, a]$ Means
  - α is at top of stack
  - Input string is derivable from βa
- In other words, when we reduce $X \rightarrow \alpha\beta$, a must be the lookahead symbol.
- Or, Only put 'reduce by $X \rightarrow \alpha\beta$' in `action[s,a]`
- Can construct states as before, but have to modify closure

What LR(1) Items Mean

- $[X \rightarrow \bullet\ \alpha\ \beta\ \gamma,\ a]$
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$
- $[X \rightarrow \alpha\ \bullet\ \beta\ \gamma,\ a]$
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and we have already recognized α
- $[X \rightarrow \alpha\ \beta\ \bullet\ \gamma,\ a]$
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and we have already recognized $\alpha\ \beta$
- $[X \rightarrow \alpha\ \beta\ \gamma\ \bullet,\ a]$
  input is consistent with $X \rightarrow \alpha\ \beta\ \gamma$ and if lookahead symbol is a, then we can reduce to X

LR(1) Closure

```
closure(state)
   repeat
      foreach item [A → α•Xβ, t] in state
         foreach production X → w
           and each terminal t′ in FIRST(βt)
            state.add([X → •w, t′])
   until state does not change
   return state
```

Closure

0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

closure({S → •E$, ?}) =

S → •E$, ?

FIRST($) = {$}

Closure

$0\ S \rightarrow E\$$
$1\ E \rightarrow L = R$
$2\ E \rightarrow R$
$3\ L \rightarrow id$
$4\ L \rightarrow *R$
$5\ R \rightarrow L$

closure($\{S \rightarrow \bullet E\$,\ ?\}$) =

$S \rightarrow \bullet E\$,$	?
$E \rightarrow \bullet L = R,$	\$
$E \rightarrow \bullet R,$	\$

FIRST(\$)	{\$}
FIRST(=R\$)	{=}

Closure

closure({$S \rightarrow \bullet E\$$, ?}) =

$S \rightarrow \bullet E\$$,	?
$E \rightarrow \bullet L = R$,	\$
$E \rightarrow \bullet R$,	\$
$L \rightarrow \bullet id$,	=
$L \rightarrow \bullet *R$,	=

0 $S \rightarrow E\$$
1 $E \rightarrow L = R$
2 $E \rightarrow R$
3 $L \rightarrow id$
4 $L \rightarrow *R$
5 $R \rightarrow L$

FIRST(\$)	{\$}
FIRST(=R\$)	{=}

Closure

closure({$S \rightarrow \bullet E\$, ?$}) =

$S \rightarrow \bullet E\$,$	?
$E \rightarrow \bullet L = R,$	\$
$E \rightarrow \bullet R,$	\$
$L \rightarrow \bullet id,$	=
$L \rightarrow \bullet {*}R,$	=
$R \rightarrow \bullet L,$	\$

0 $S \rightarrow E\$$
1 $E \rightarrow L = R$
2 $E \rightarrow R$
3 $L \rightarrow id$
4 $L \rightarrow {*}R$
5 $R \rightarrow L$

FIRST(\$)	{\$}
FIRST(=R\$)	{=}

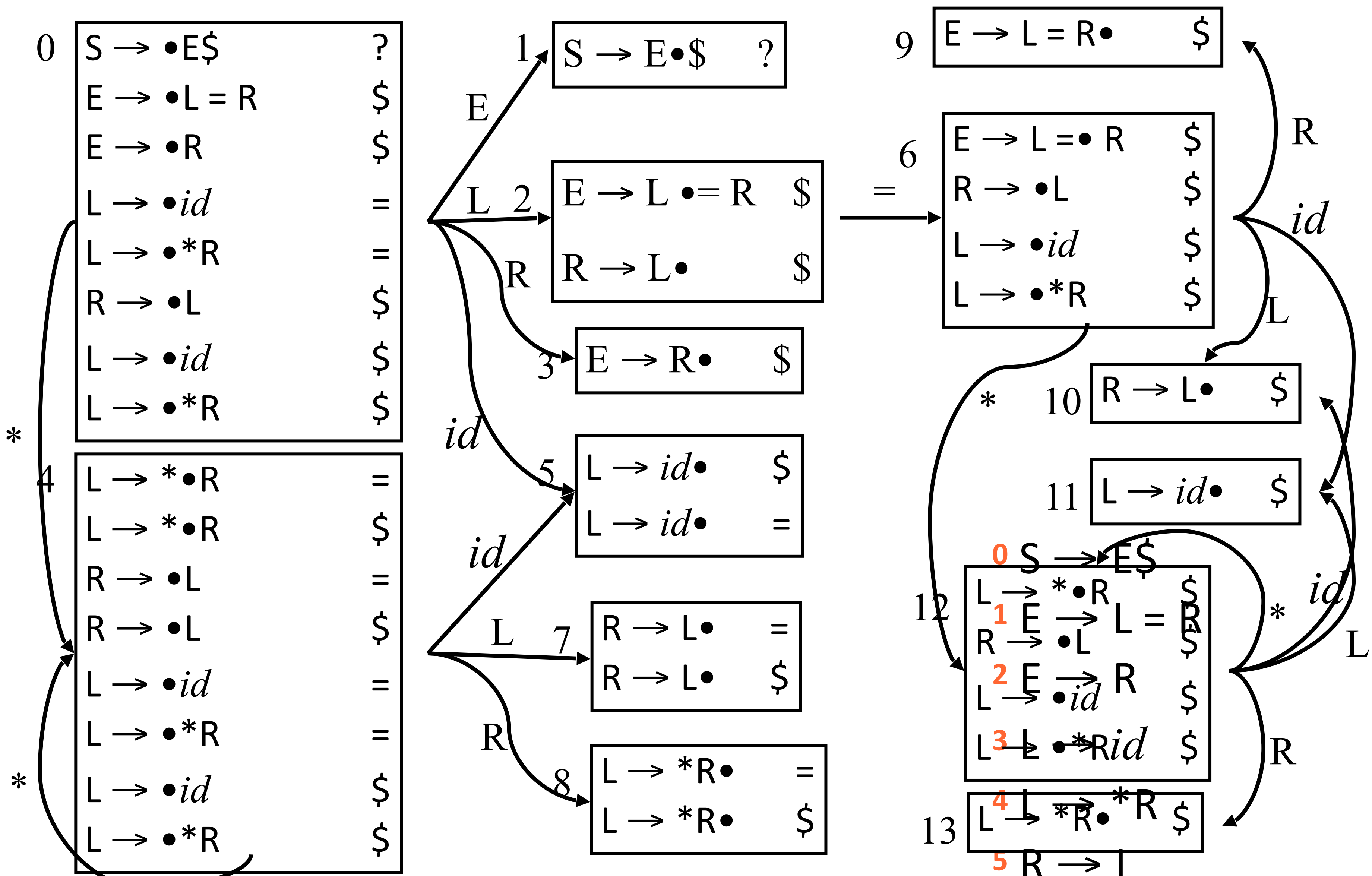
Closure

closure({S → •E$, ?}) =

S → •E$,	?
E → •L = R,	$
E → •R,	$
L → •*id*,	=
L → •*R,	=
R → •L,	$
L → •*id*,	$
L → •*R,	$

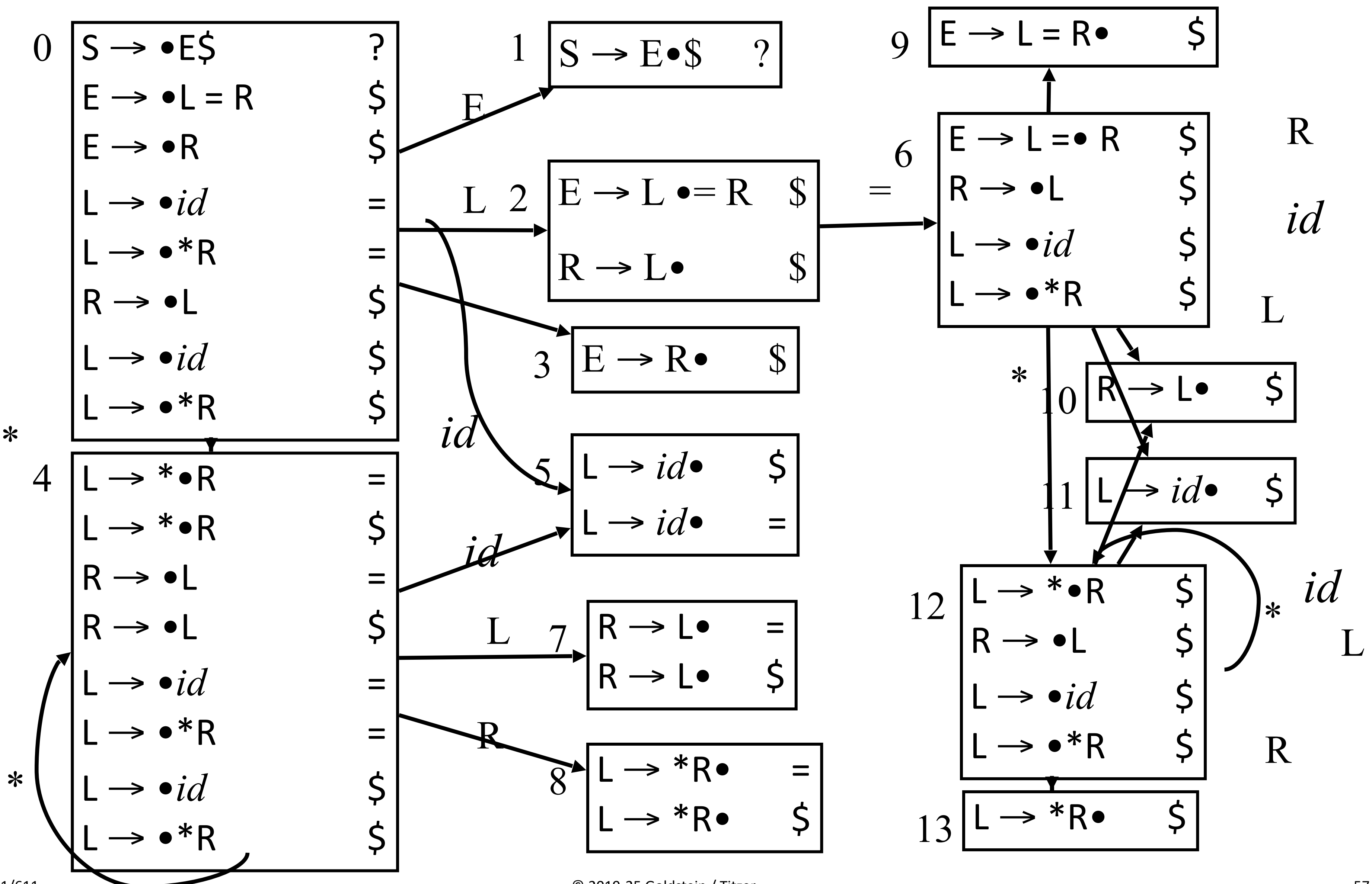
0 S → E$
1 E → L = R
2 E → R
3 L → *id*
4 L → *R
5 R → L

LR(1) Example



0
S → •E$?
E → •L = R $
E → •R $
L → •id =
L → •*R =
R → •L $
L → •id $
L → •*R $
1
S → E•$?
2
E → L •= R $
R → L• $
3
E → R• $
4
L → *•R =
L → *•R $
R → •L =
R → •L $
L → •id =
L → •*R =
L → •id $
L → •*R $
5
L → id• $
L → id• =
6
E → L =• R $
R → •L $
L → •id $
L → •*R $
7
R → L• =
R → L• $
8
L → *R• =
L → *R• $
9
E → L = R• $
10
R → L• $
11
L → id• $
12
L → *•R $
R → •L $
L → •id $
L → •*R $
13
L → *R• $
0 S → E$
1 E → L = R
2 E → R
3 L → id
4 L → *R
5 R → L
E
L
R
id
=
*

LR(1) Example


0
S → •E$?
E → •L = R $
E → •R $
L → •id =
L → •*R =
R → •L $
L → •id $
L → •*R $
1
S → E•$?
2
E → L •= R $
R → L• $
3
E → R• $
4
L → *•R =
L → *•R $
R → •L =
R → •L $
L → •id =
L → •*R =
L → •id $
L → •*R $
5
L → id• $
L → id• =
6
E → L =• R $
R → •L $
L → •id $
L → •*R $
7
R → L• =
R → L• $
8
L → *R• =
L → *R• $
9
E → L = R• $
10
R → L• $
11
L → id• $
12
L → *•R $
R → •L $
L → •id $
L → •*R $
13
L → *R• $
E
L
id
id
L
R
*
*
=
R
id
L
*
*
id
L
R

Parsing Table for LR(1)

- 14 states versus 10 LR(0) states
- In general, the number of states (and therefore size of the parsing table) is much larger with LR(1) items

LALR: Lookahead LR

- More powerful than SLR
- Given LR(1) states, merge states that are identical except for lookaheads
- End up with ~ same size table as SLR
- Can this introduce conflicts?

LALR

- Can generate parse table without constructing LR(1) item sets
  - construct LR(0) item sets
  - compute *lookahead* sets
    - more precise than follow sets
- LALR is used by most parser generators (e.g., bison,lalrpop,ocamlyacc,…)

Recap

- LR(0) not very useful
- SLR uses follow sets to reduce
- LALR uses lookahead sets
- LR(1) uses full lookahead context

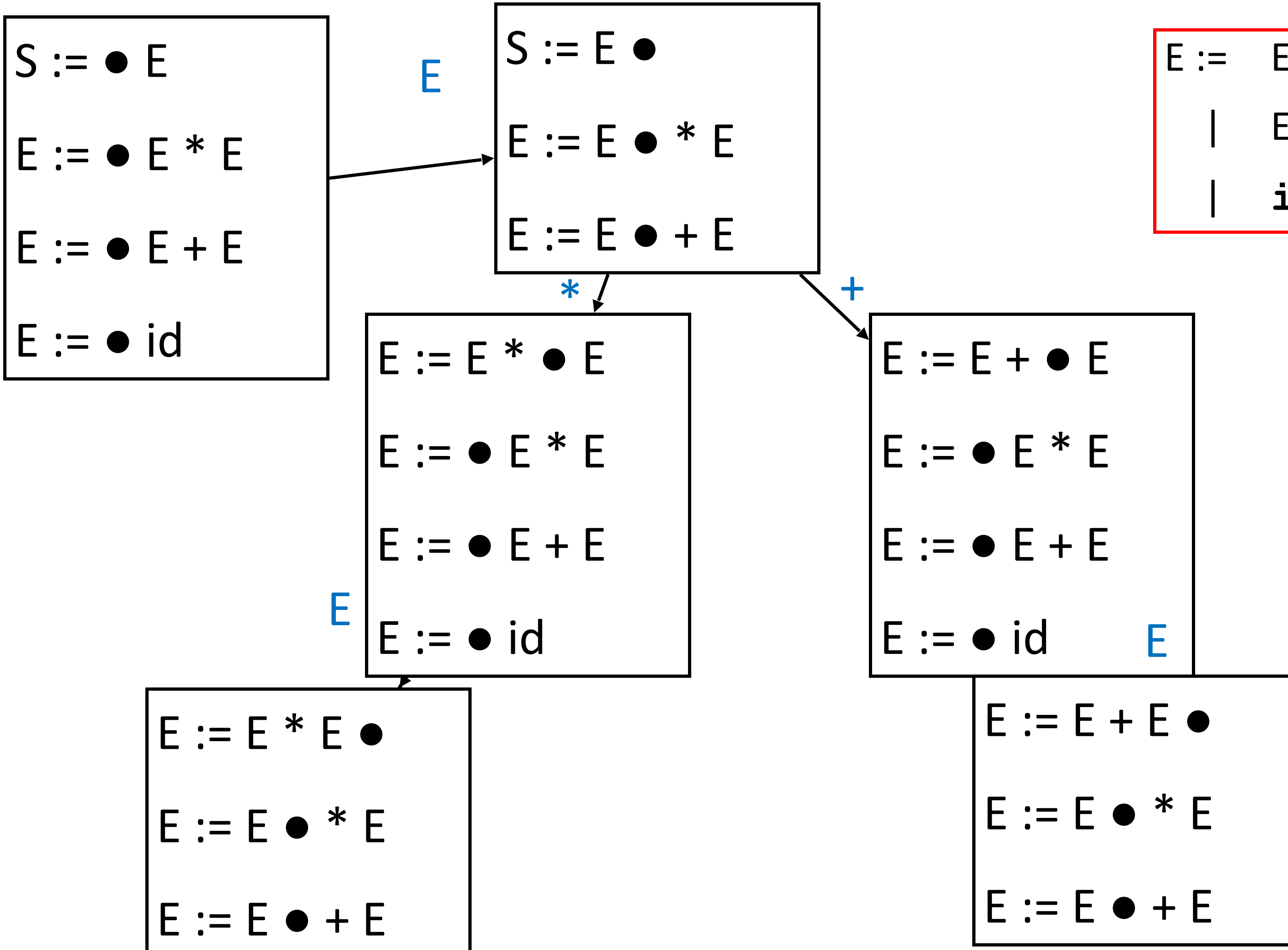
Power of shift-reduce parsers

- There are unambiguous grammars which cannot be parsed with shift-reduce parsers.
- Such grammars can have
  – shift/reduce conflicts
  – reduce/reduce conflicts
- There grammars are not LR(k)
- But, we can often choose shift or reduce to recognize what want.

Handling precedence and associativity

- Straightforwardly written grammars for infix will have shift/reduce conflicts
- Associativity: a + b •+ c
  - prefer reduce => left-associative (a + b) + c
  - prefer shift => right-associative a + (b + c)
- Precedence: a * b •+ c
  - prefer reduce => higher prec (a * b) + c
  - prefer shift => lower prec a * (b + c)

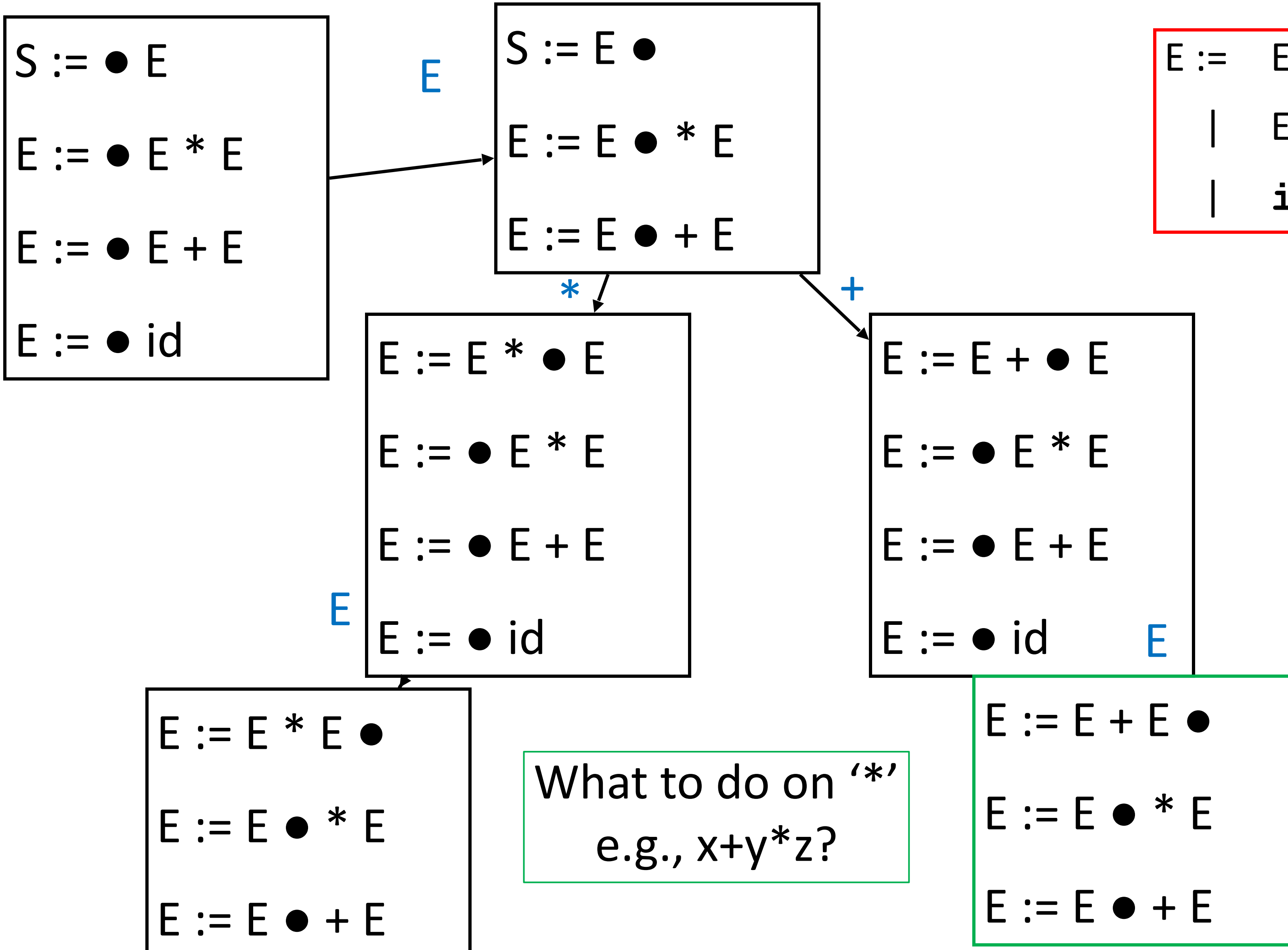
Expression Grammars & Precedence



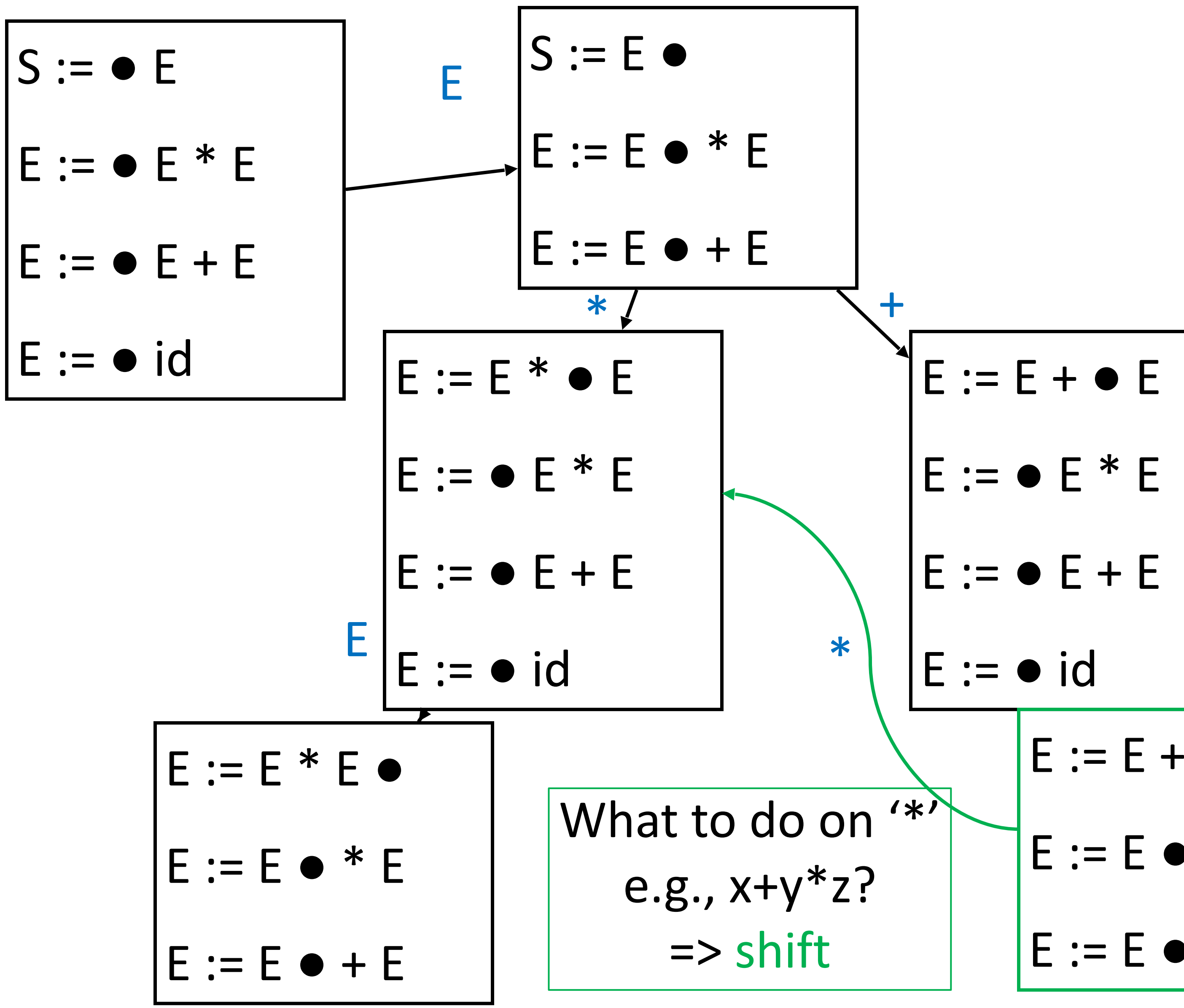

```
E :=  E*E
   |  E+E
   |  id
```

Expression Grammars & Precedence

E := E*E
| E+E
| **id**



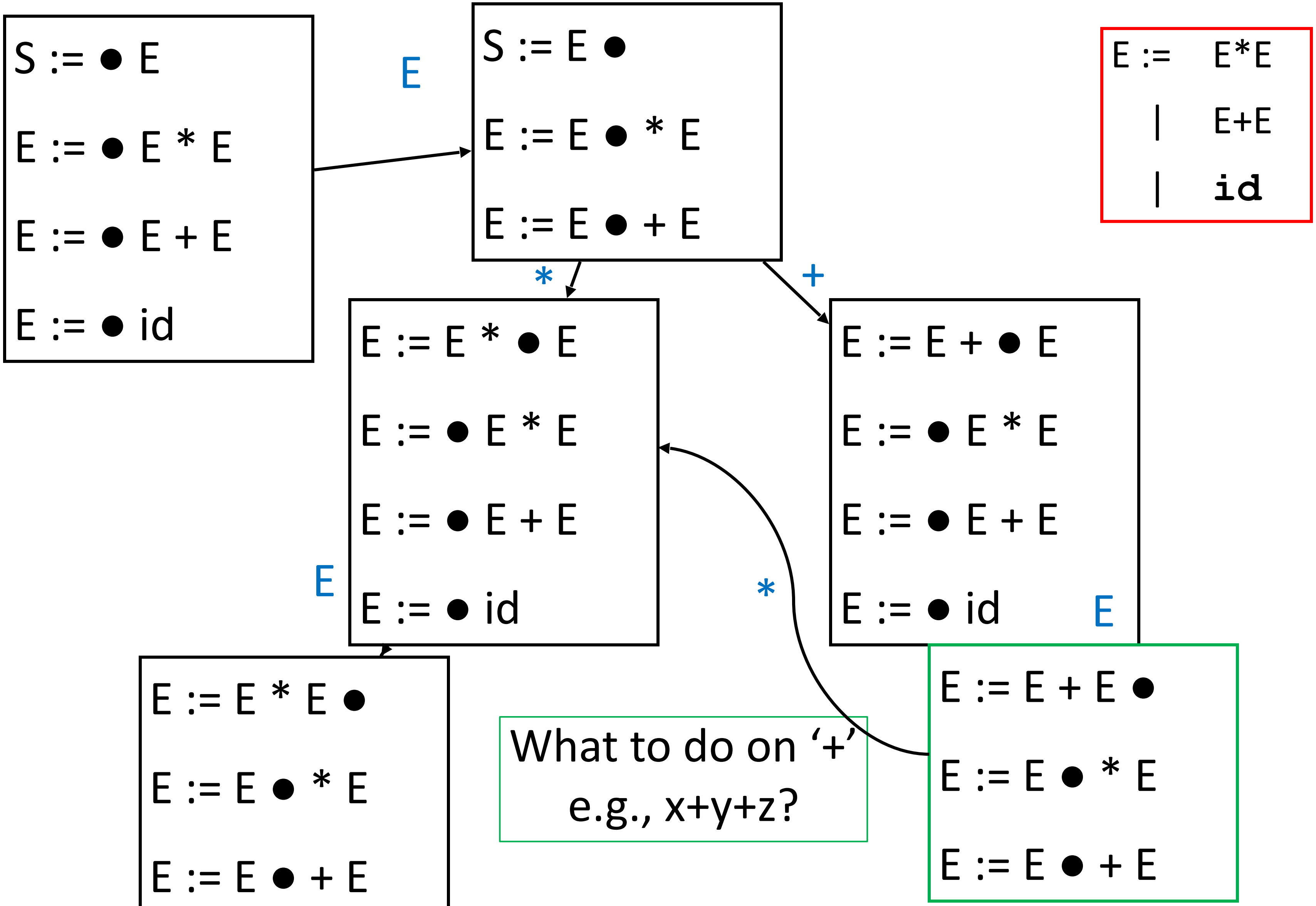
Expression Grammars & Precedence



E := E*E
| E+E
| **id**

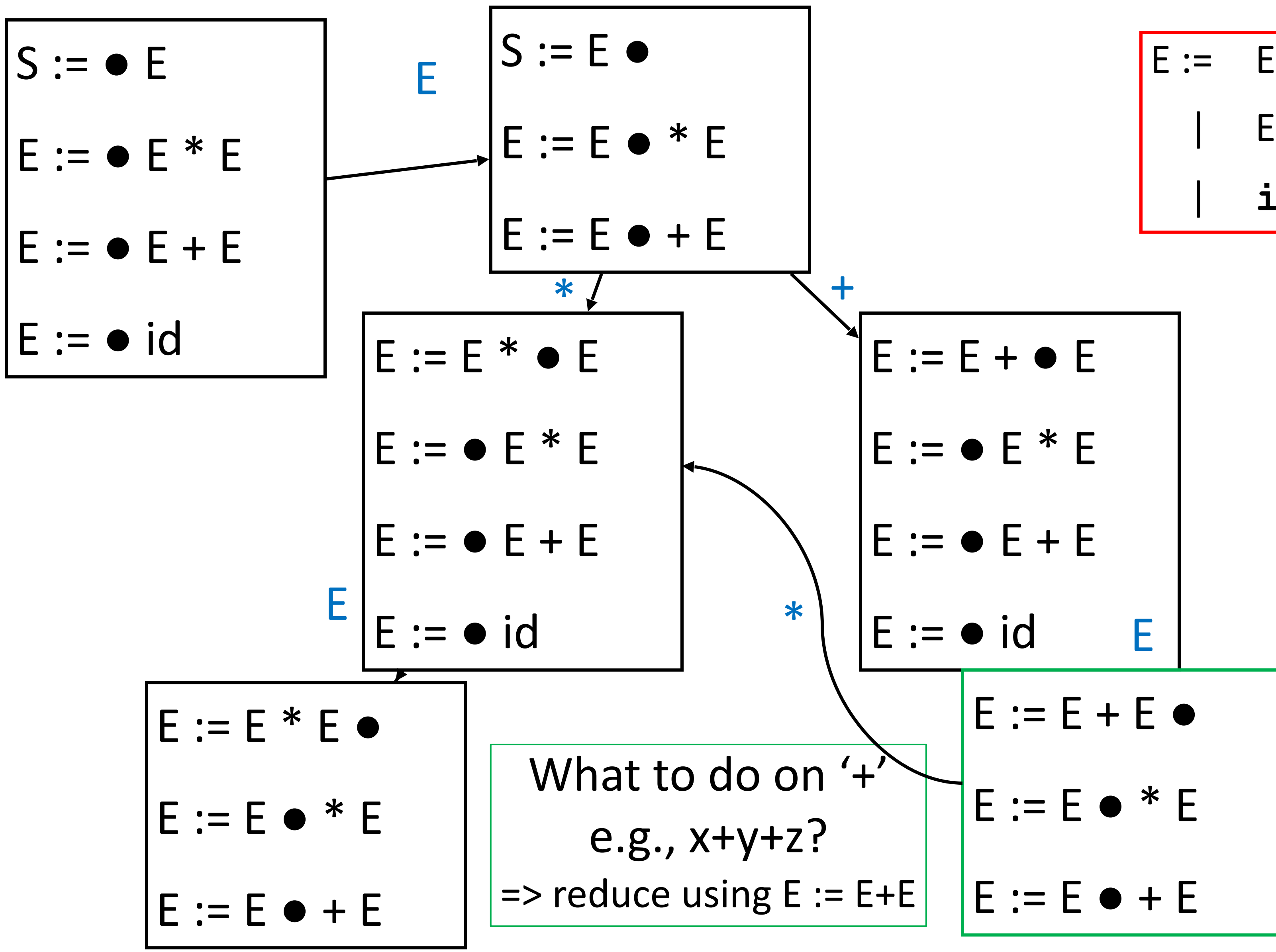
What to do on '*'
e.g., x+y*z?
=> shift

Expression Grammars & Precedence



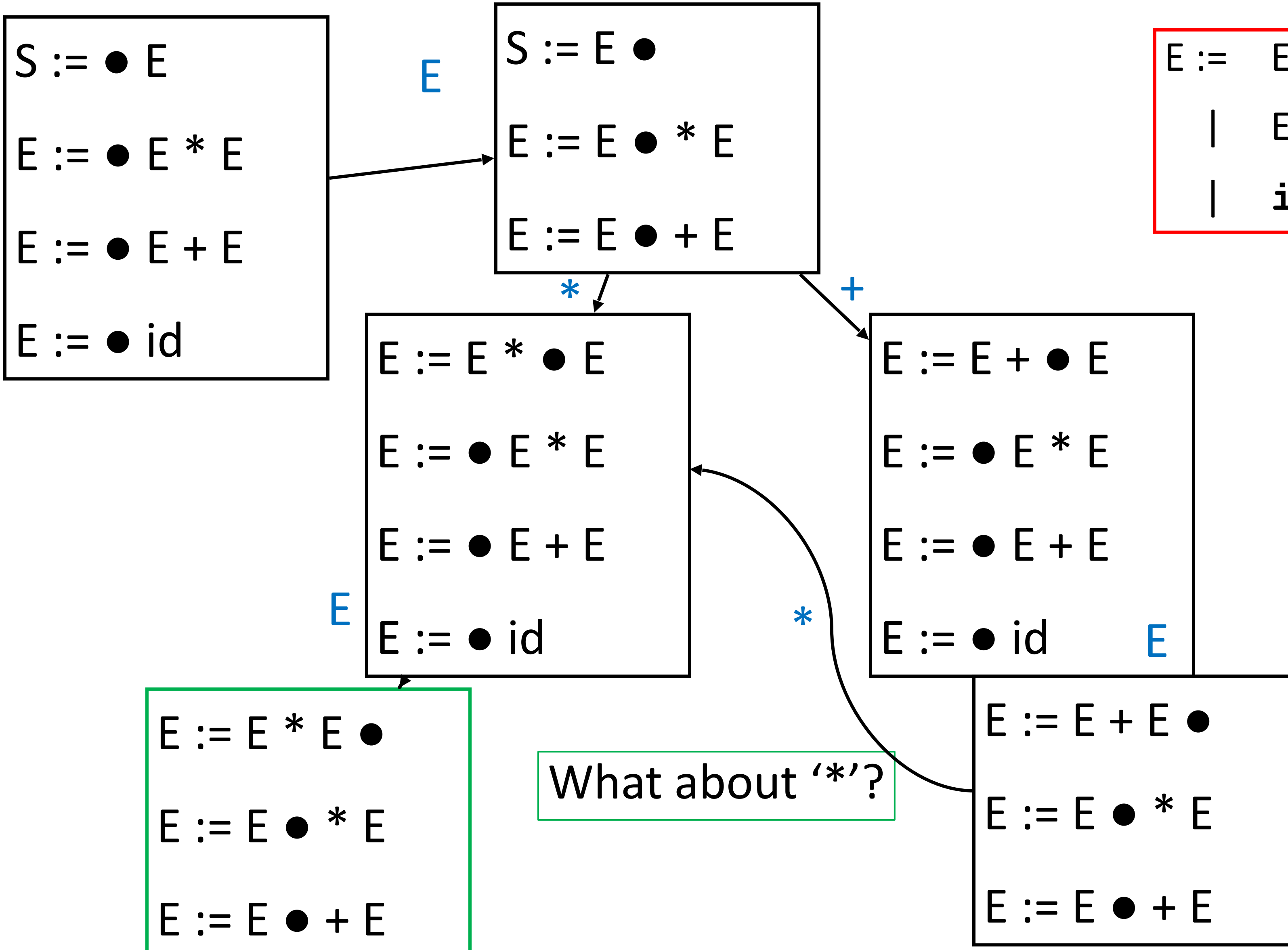
Expression Grammars & Precedence

E := E*E
  | E+E
  | **id**



What to do on '+'
e.g., x+y+z?
=> reduce using E := E+E

Expression Grammars & Precedence



E := E*E
| E+E
| **id**

Bison

- Precedence and Associativity declarations
- Precedence derived from order of directives: from lowest to highest
- Associativity from %left, %right, %nonassoc
- Can be attached to rules as well (This can solve the dangling if-else problem
- Use output of generator showing items and transitions to debug s/r and r/r errors

Dangling Else

S := **if** E **then** S

| **if** E **then** S **else** S

| **other**

- We can be in the following state:

… **if** E **then** S   **else** … **$**

- What do we do?
  - shift the **else** (hoping to reduce by second rule), or
  - reduce by first rule