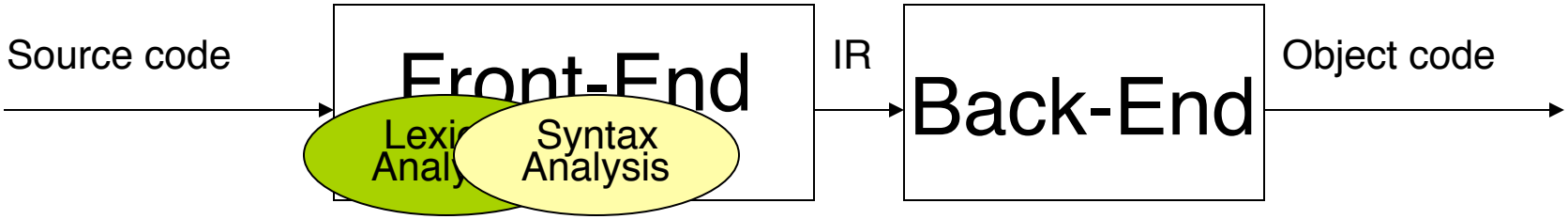
Compiler Design

Lecture 6:
Syntax Analysis
Bottom-Up Parsing (part V)

Dr. Momtazi
momtazi@aut.ac.ir

based on the slides of the course book

Parsing (Syntax Analysis)



- Syntax Analysis:
  - Derivation and parse trees ✓
  - Top-down parsing ✓
  - **Bottom-up parsing**

Outline

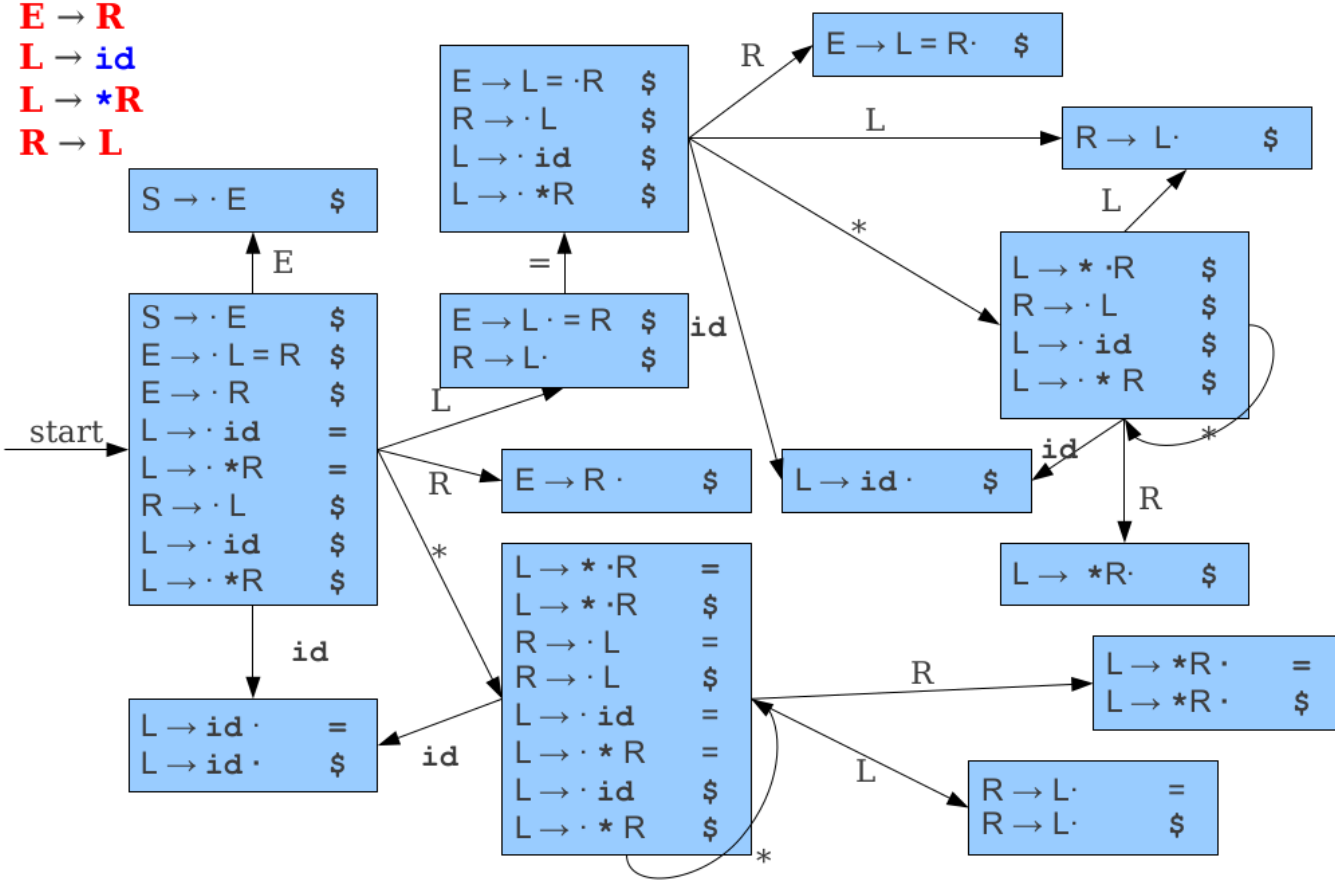
- Introduction
- Shift-reduce Parsing
- **LR Parsing**
  - LR(0)
  - LR(1)
  - SLR
  - **LALR**
- Ambiguity
- Error-handling

From LR(1) to LALR(1)

$S \rightarrow E$
$E \rightarrow L = R$
$E \rightarrow R$
$L \rightarrow \texttt{id}$
$L \rightarrow *R$
$R \rightarrow L$

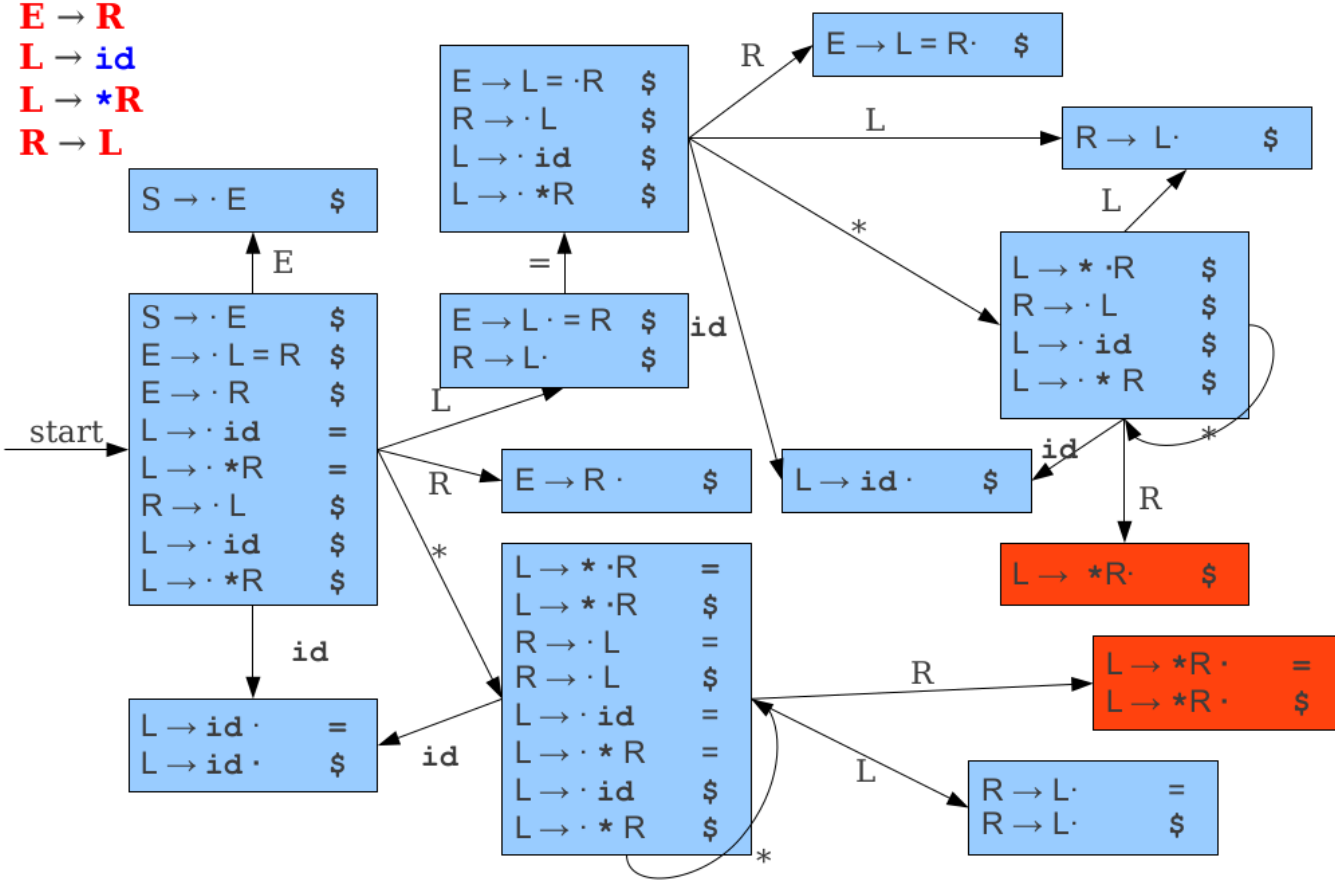
From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



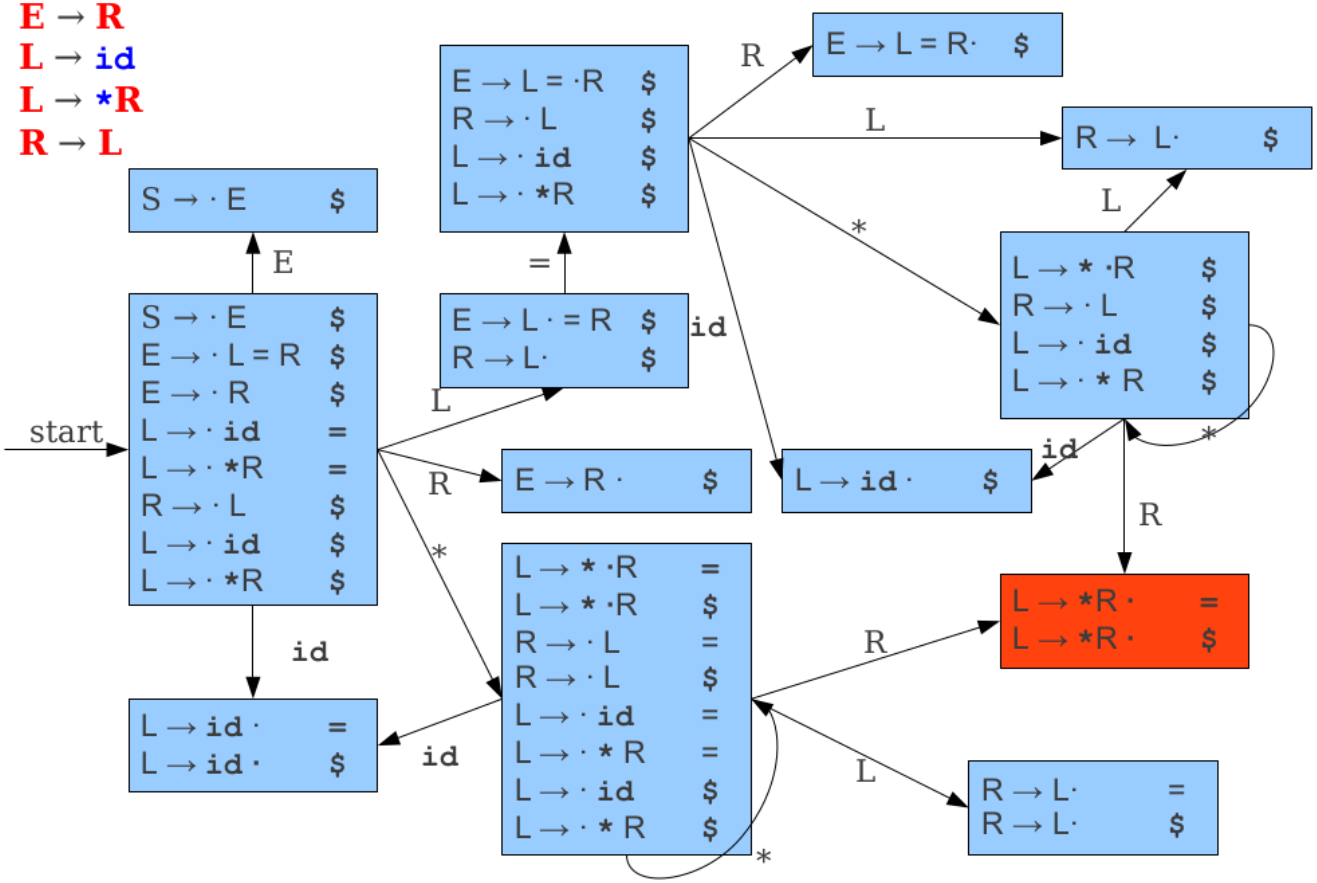
From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



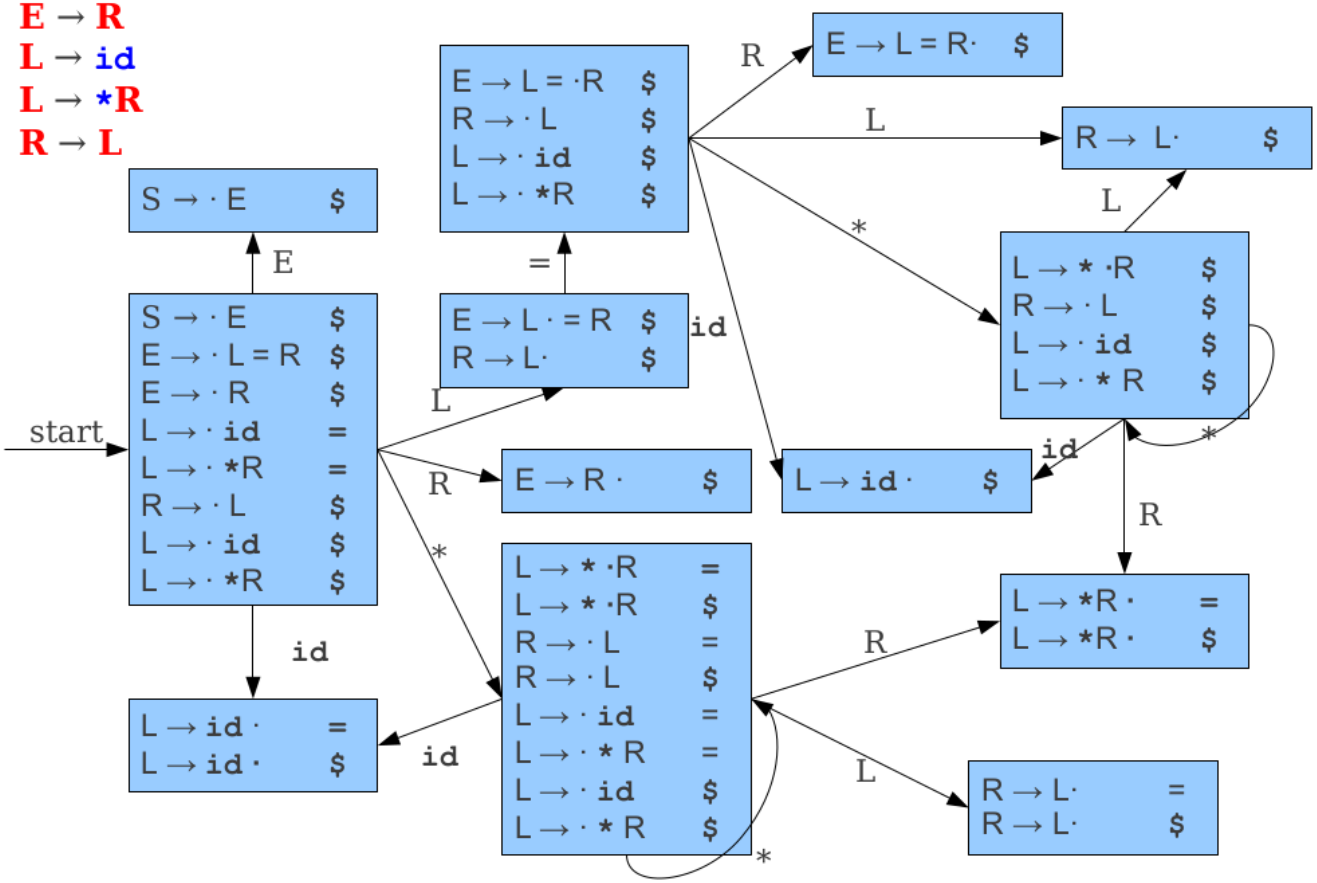
From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



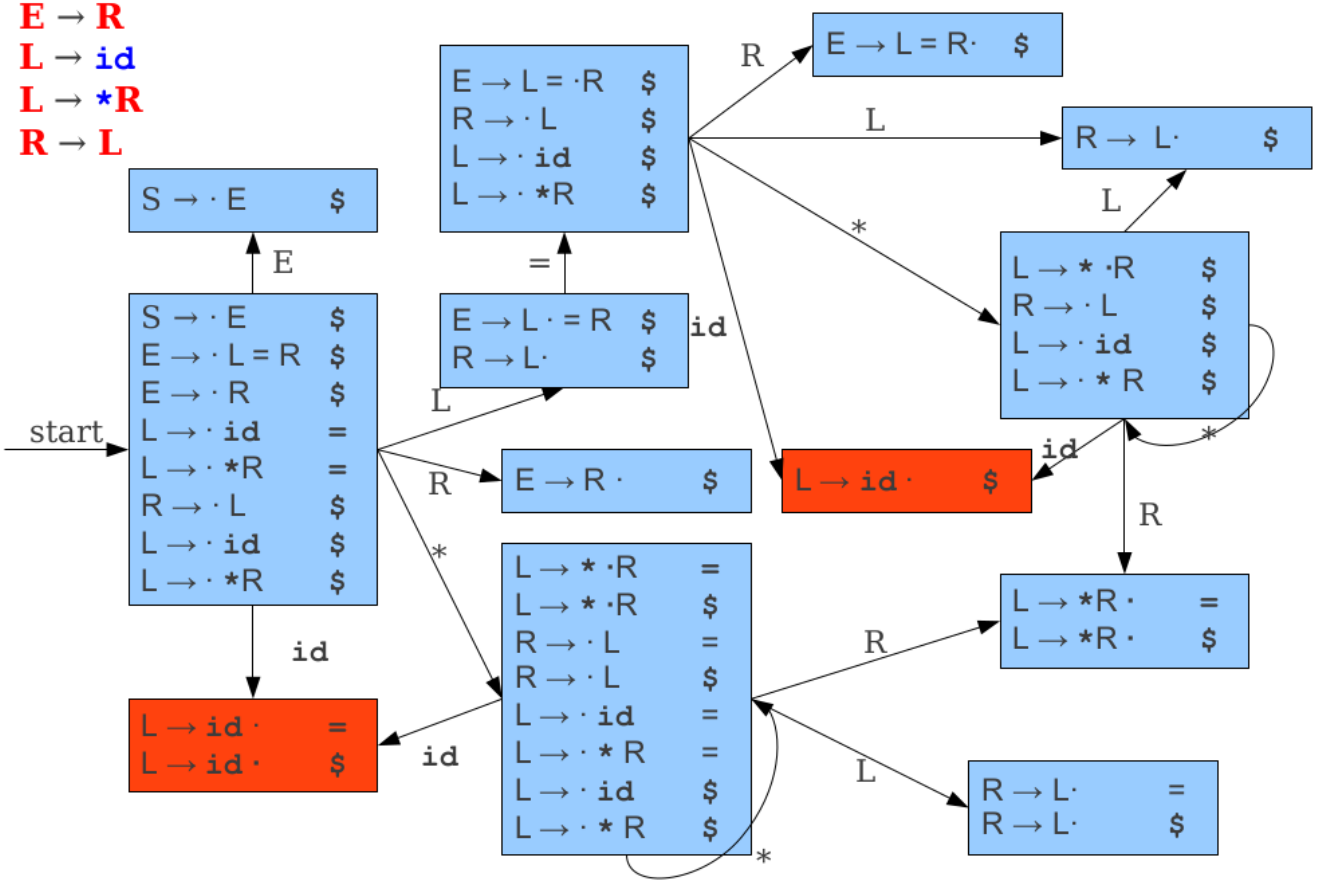
From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



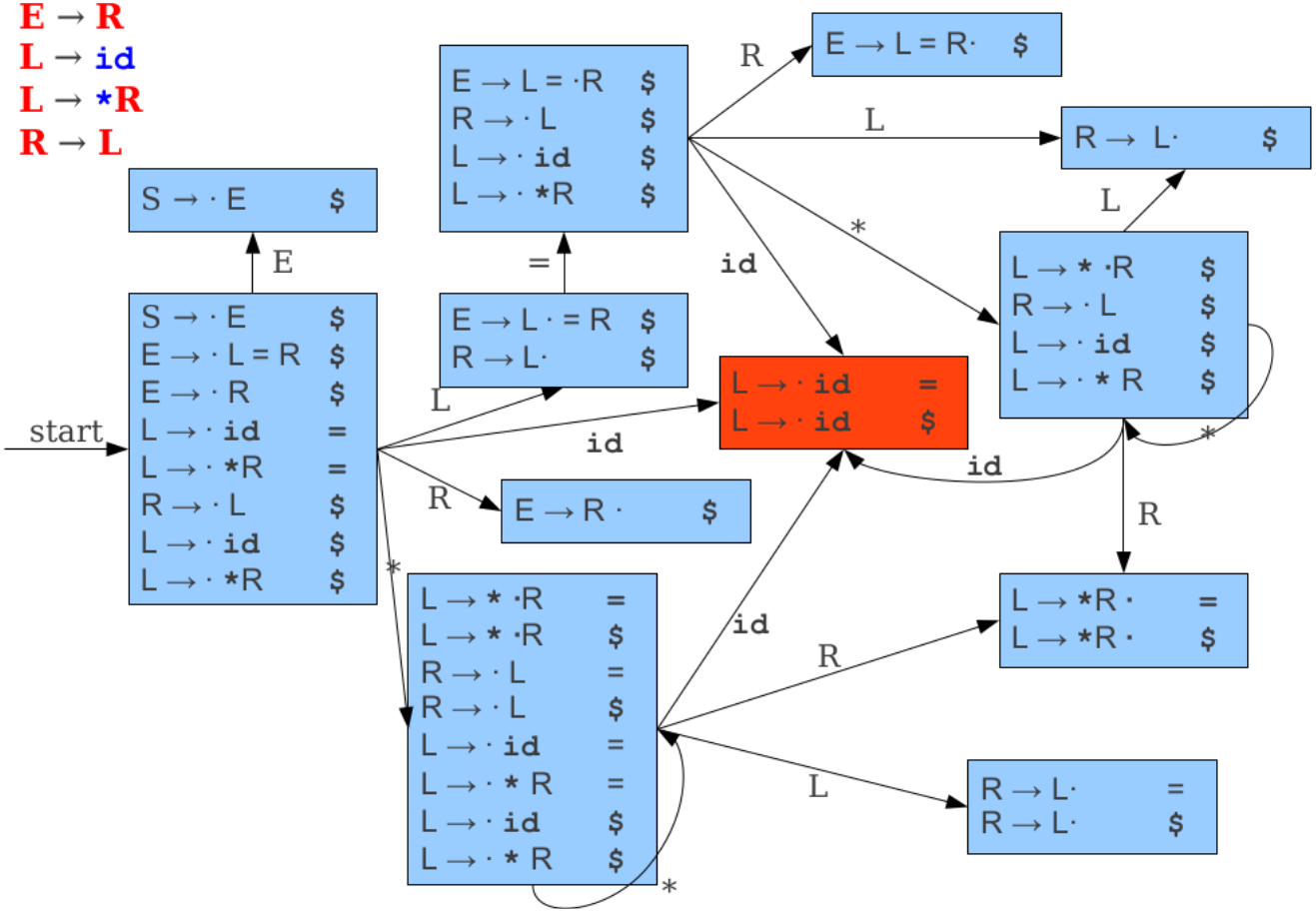
From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



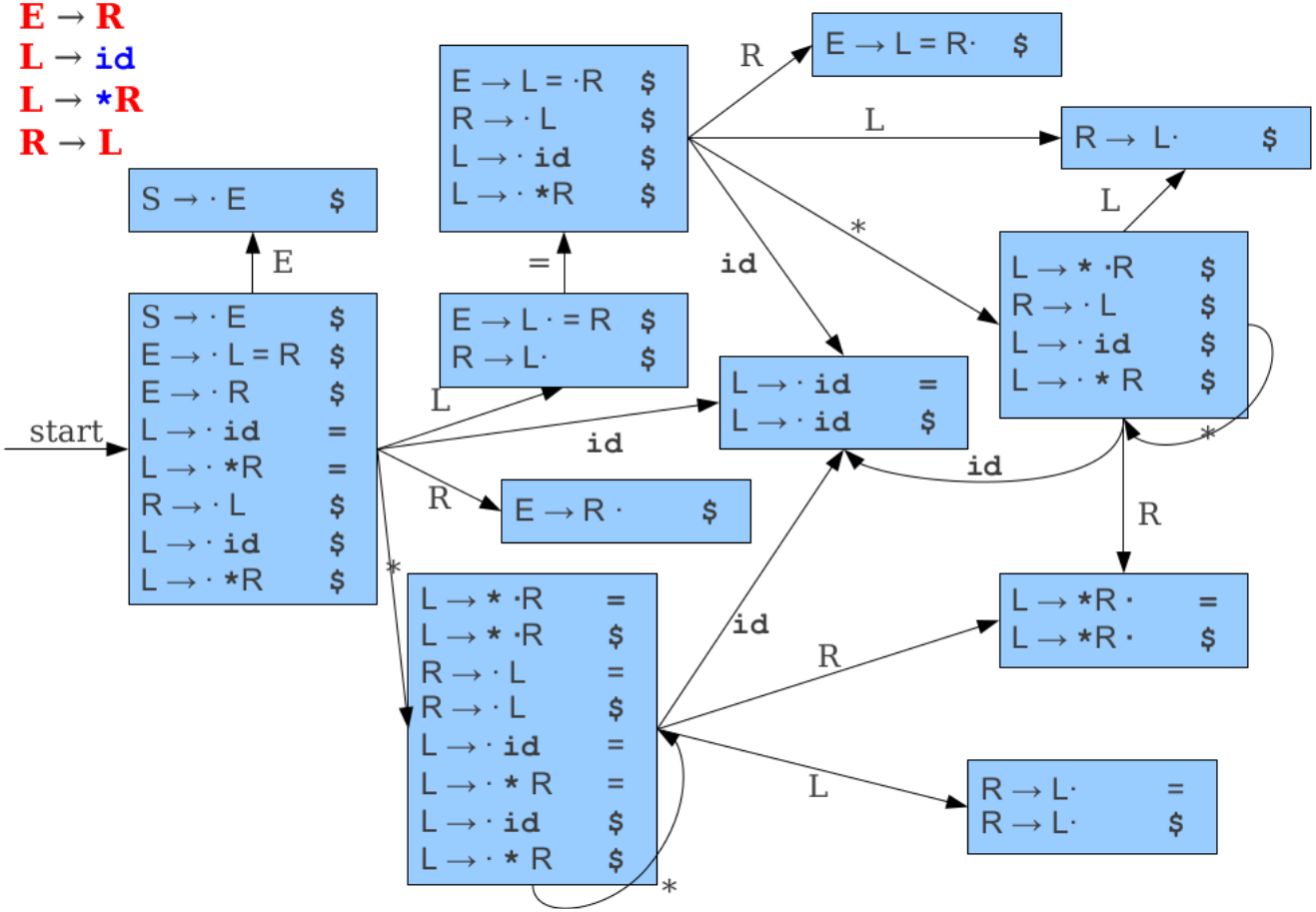
From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



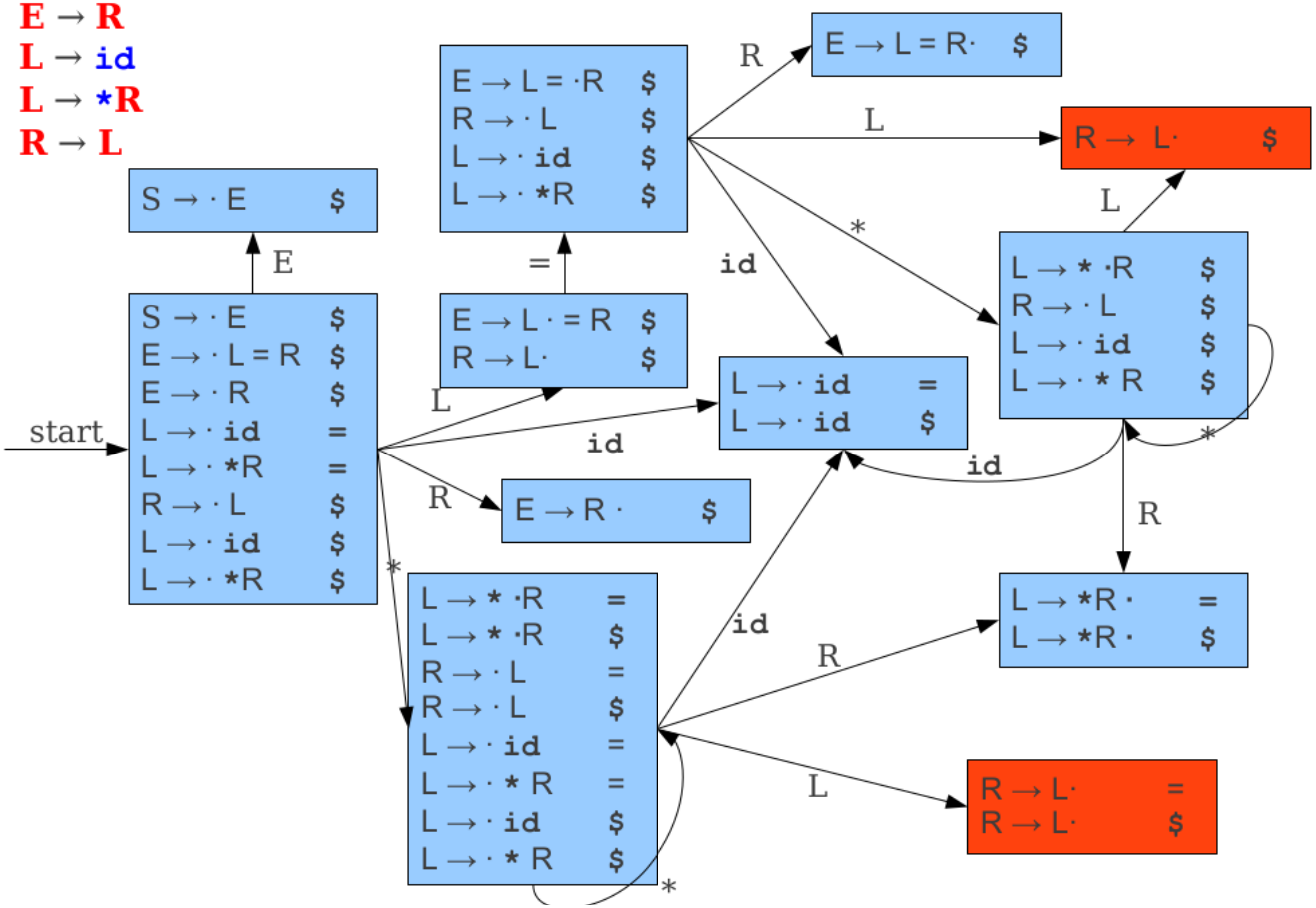
From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



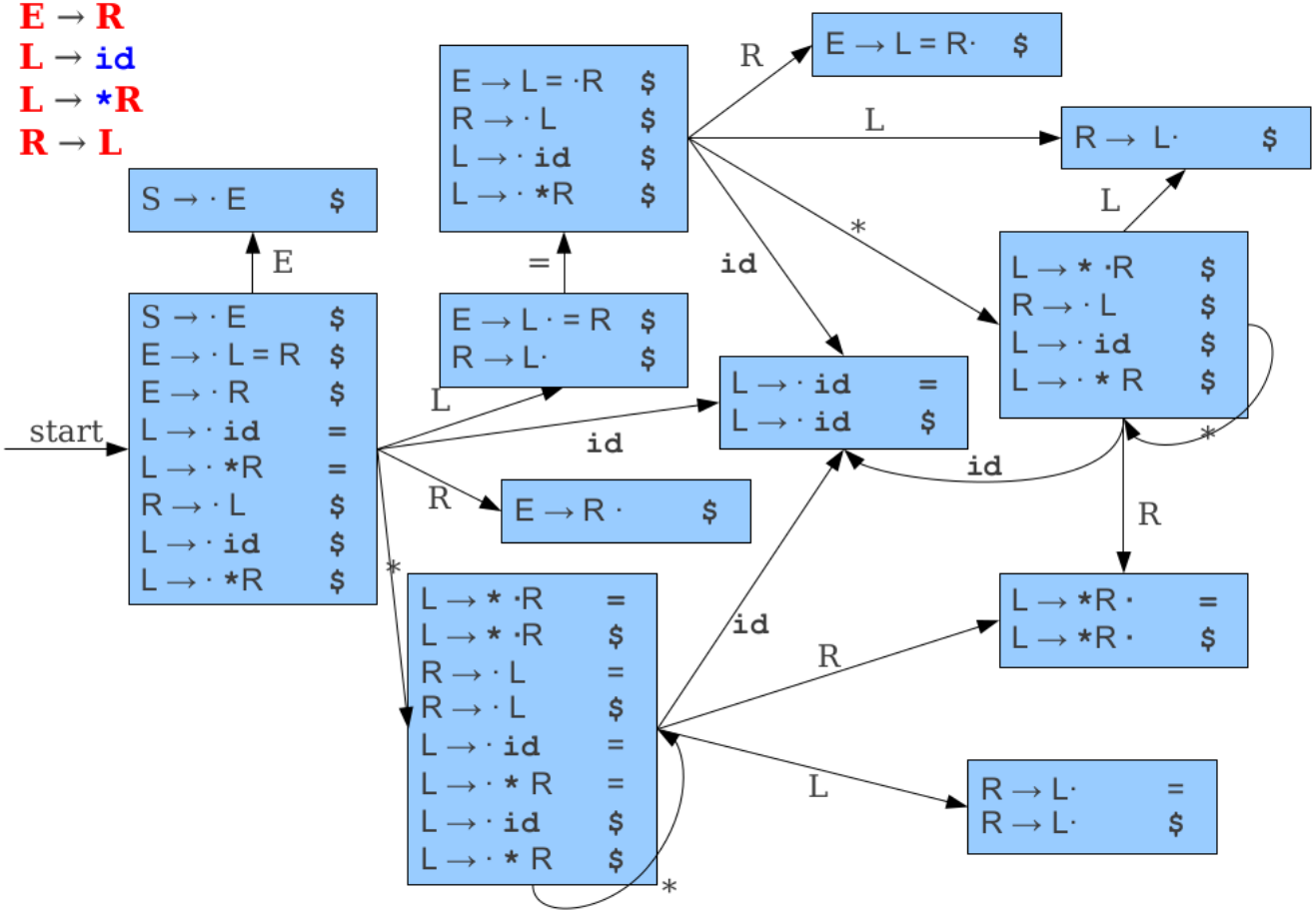
From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



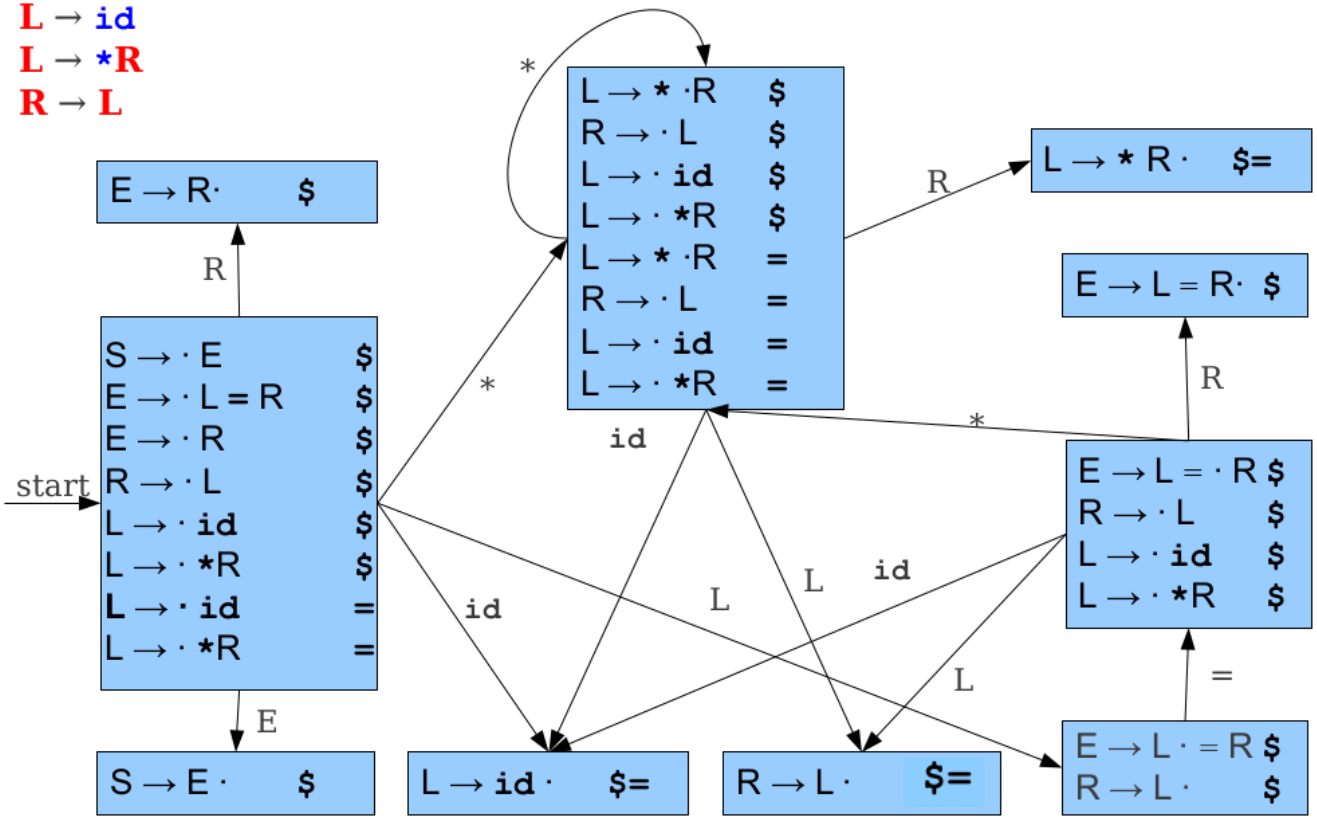
From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



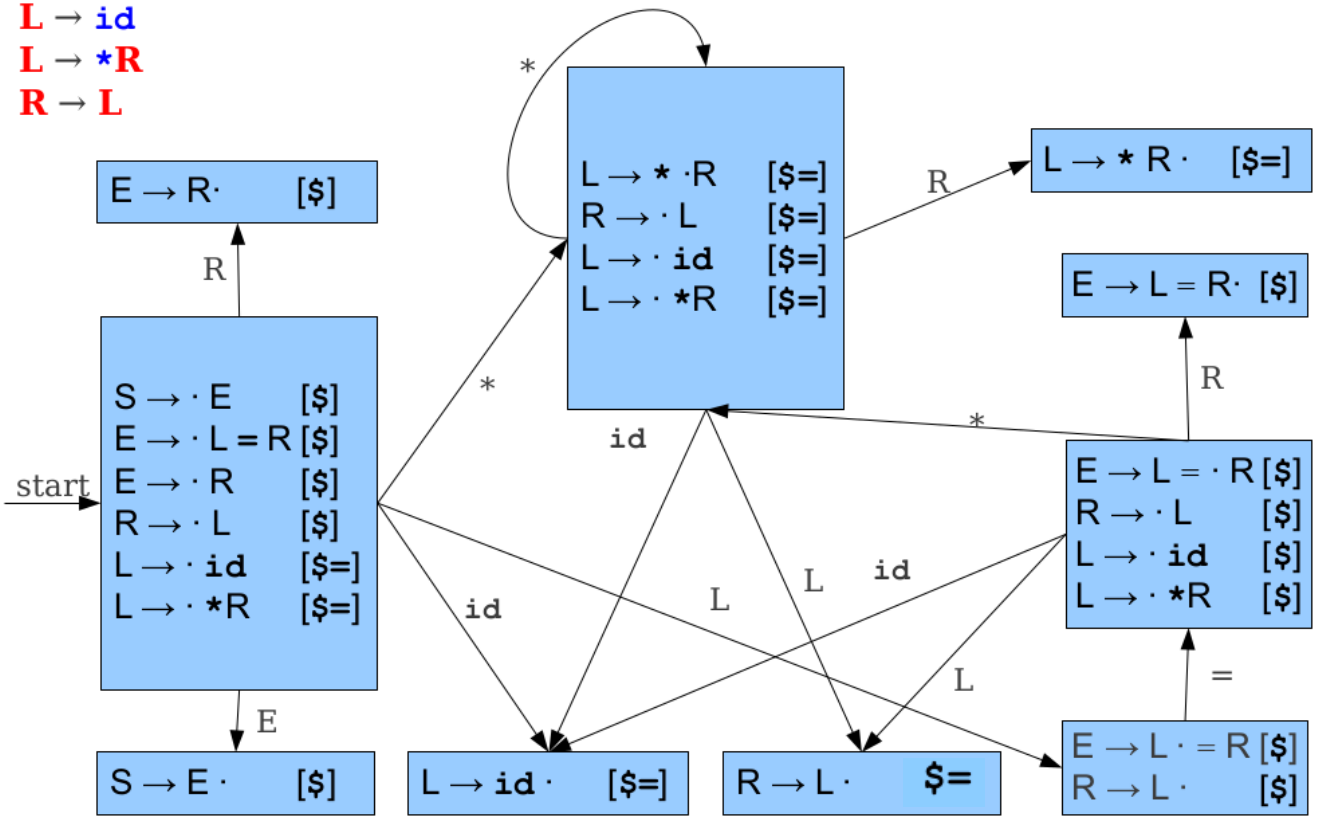
From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



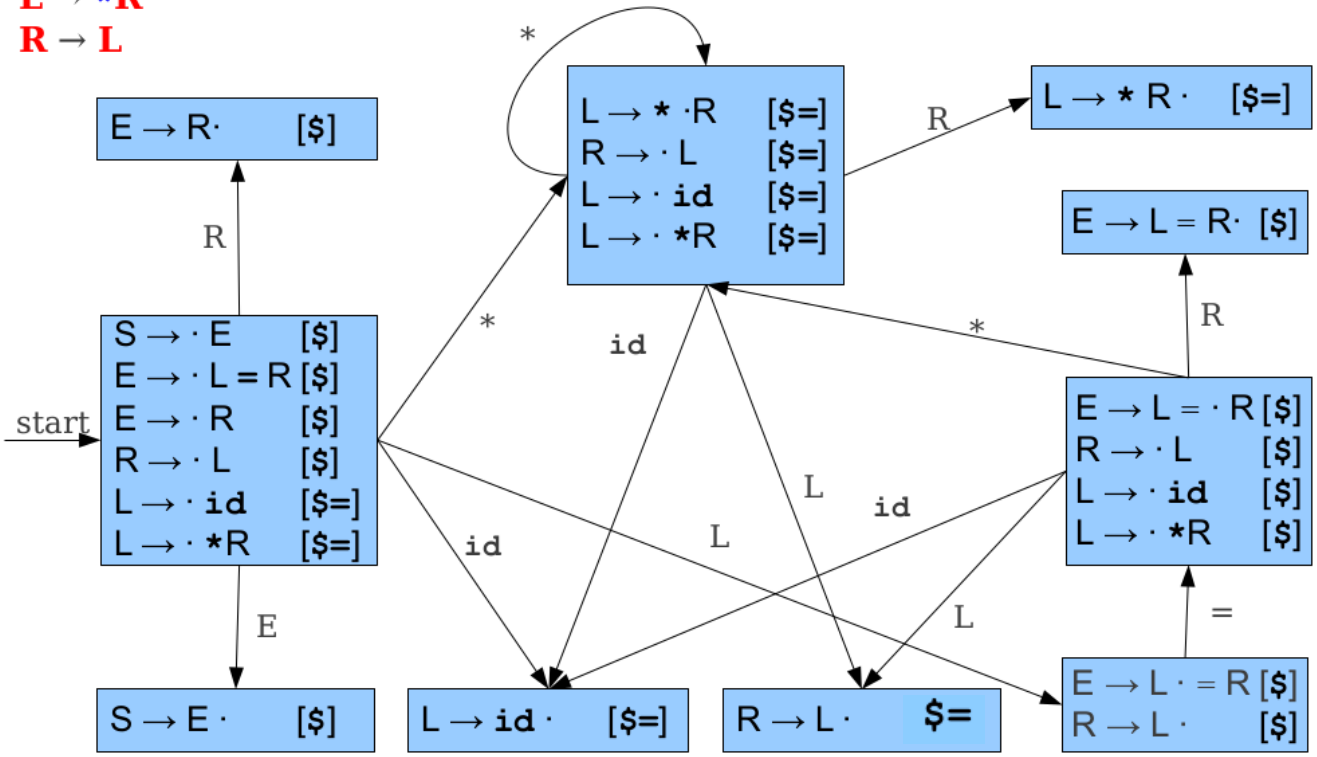
From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



From LR(1) to LALR(1)

S → E
E → L = R
E → R
L → id
L → *R
R → L



Advantages of LALR(1)

- Maintains context.
  - Lookup sets based on the fine-grained LR(1) automaton.
  - Each state's lookup relevant only for that state.

- Keeps automaton small.
  - Resulting automaton has same size as LR(0) automaton.

LALR(1) is Powerful

- Every LR(0) grammar is LALR(1).
- Every SLR(1) grammar is LALR(1)
- Most (but not all) LR(1) grammars are LALR(1).

LALR(1) isn't LR(1)

- Merging LR(1) states cannot introduce a shift/reduce conflict.

- Why?
  - Since the items have the same core, a shift/reduce conflict in a LALR(1) state would have to also exist in one of the LR(1) states it was merged from.
  - Merging LR(1) states can introduce a reduce/reduce conflict.
  - Often these conflicts appear without any good reason; this is one limitation of LALR(1).

Constructing LALR(1) Automata

- It's not a good idea to build LALR(1) automata from LR(1) automata.

- Why?
  - LR(1) automata are impractically large.
  - There are more efficient methods for LALR(1) automata construction

The “Lazy Merging” Technique

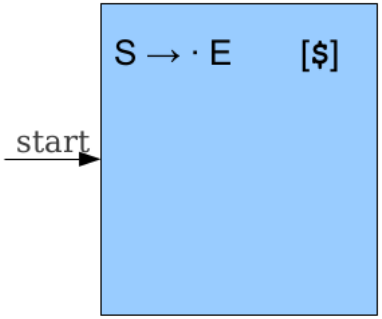
- Idea: Merge together LR(1) states as they're generated.
- Maintain a worklist of states to process; begin with the initial LR(1) state.
- When adding a new state, if it has the same core as an old state, update the old state and put it back in the worklist.

LALR(1) Construction

$S \rightarrow E$
$E \rightarrow L = R$
$E \rightarrow R$
$L \rightarrow \texttt{id}$
$L \rightarrow *R$
$R \rightarrow L$

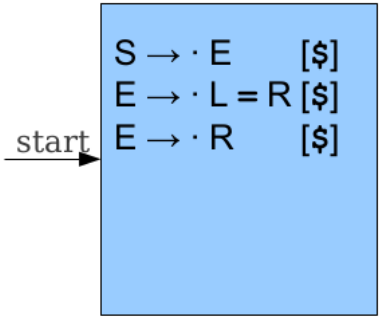
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



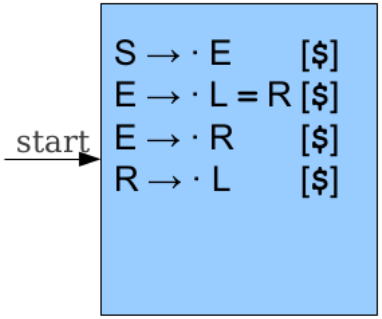
LALR(1) Construction

$S \rightarrow E$
$E \rightarrow L = R$
$E \rightarrow R$
$L \rightarrow \texttt{id}$
$L \rightarrow \texttt{*}R$
$R \rightarrow L$



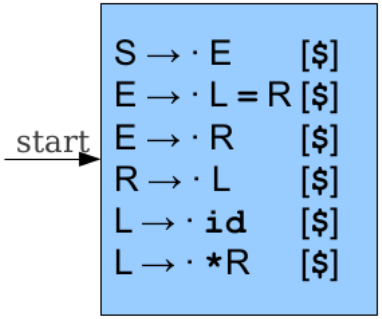
LALR(1) Construction

$S \rightarrow E$
$E \rightarrow L = R$
$E \rightarrow R$
$L \rightarrow \texttt{id}$
$L \rightarrow \texttt{*}R$
$R \rightarrow L$



LALR(1) Construction

$S \rightarrow E$
$E \rightarrow L = R$
$E \rightarrow R$
$L \rightarrow \texttt{id}$
$L \rightarrow \texttt{*}R$
$R \rightarrow L$



LALR(1) Construction

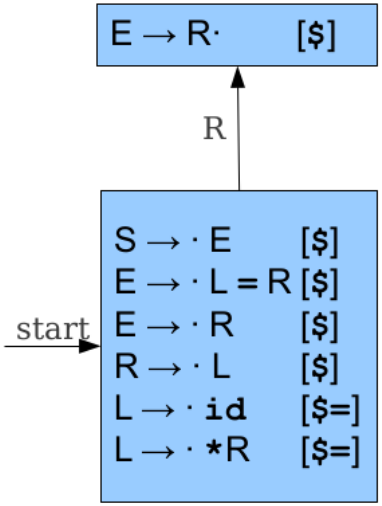
$S \rightarrow E$
$E \rightarrow L = R$
$E \rightarrow R$
$L \rightarrow \texttt{id}$
$L \rightarrow \texttt{*}R$
$R \rightarrow L$

start

$S \rightarrow \cdot E$	[$]
$E \rightarrow \cdot L = R$	[$]
$E \rightarrow \cdot R$	[$]
$R \rightarrow \cdot L$	[$]
$L \rightarrow \cdot \texttt{id}$	[$=]
$L \rightarrow \cdot \texttt{*}R$	[$=]

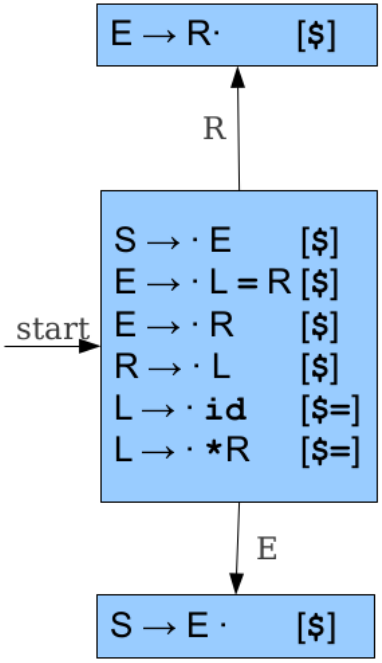
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



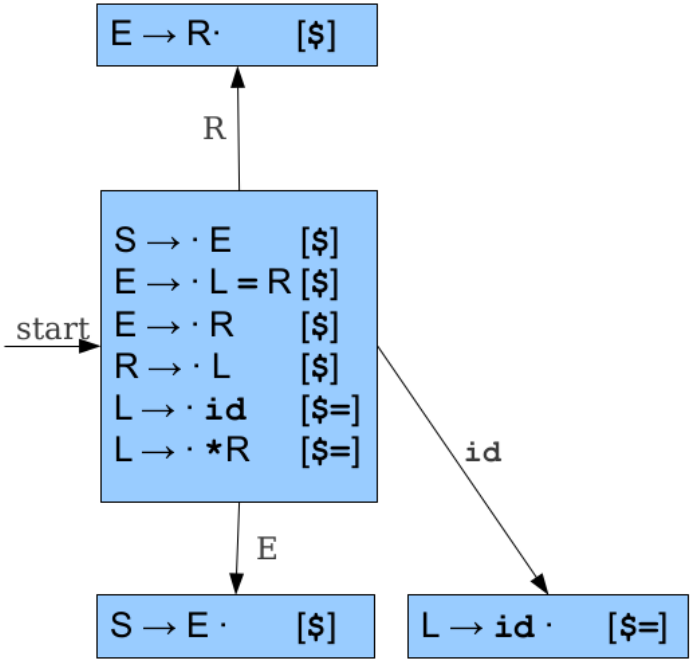
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



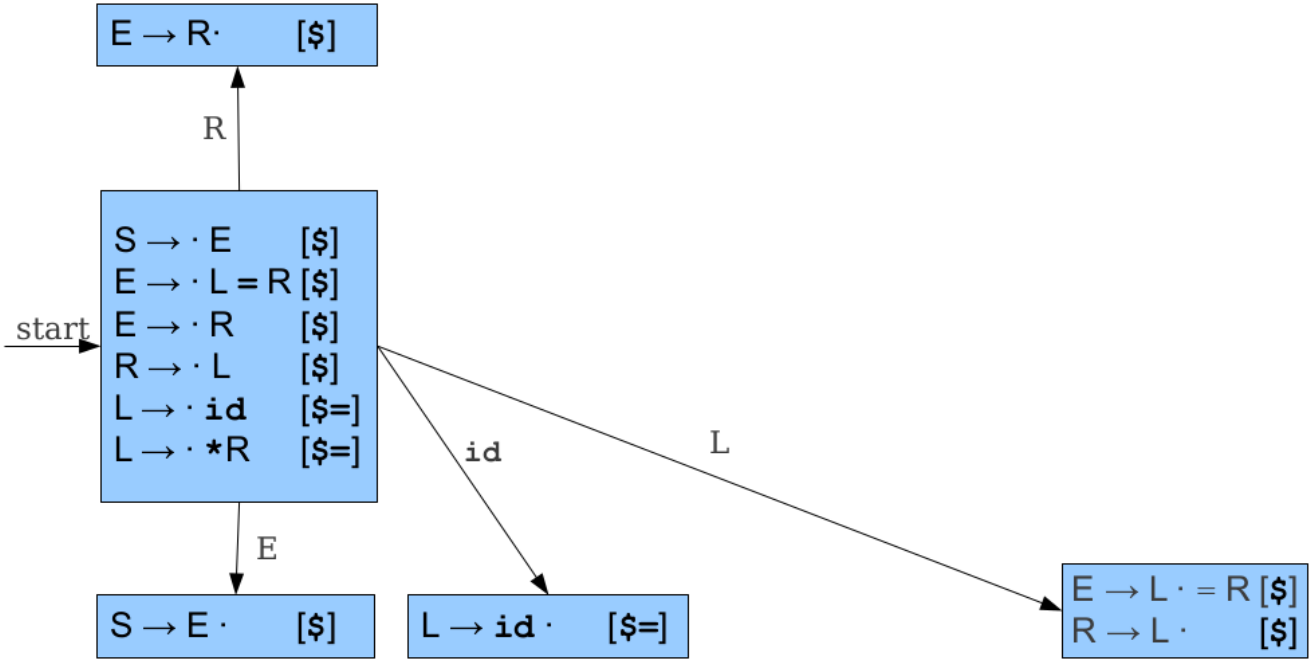
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



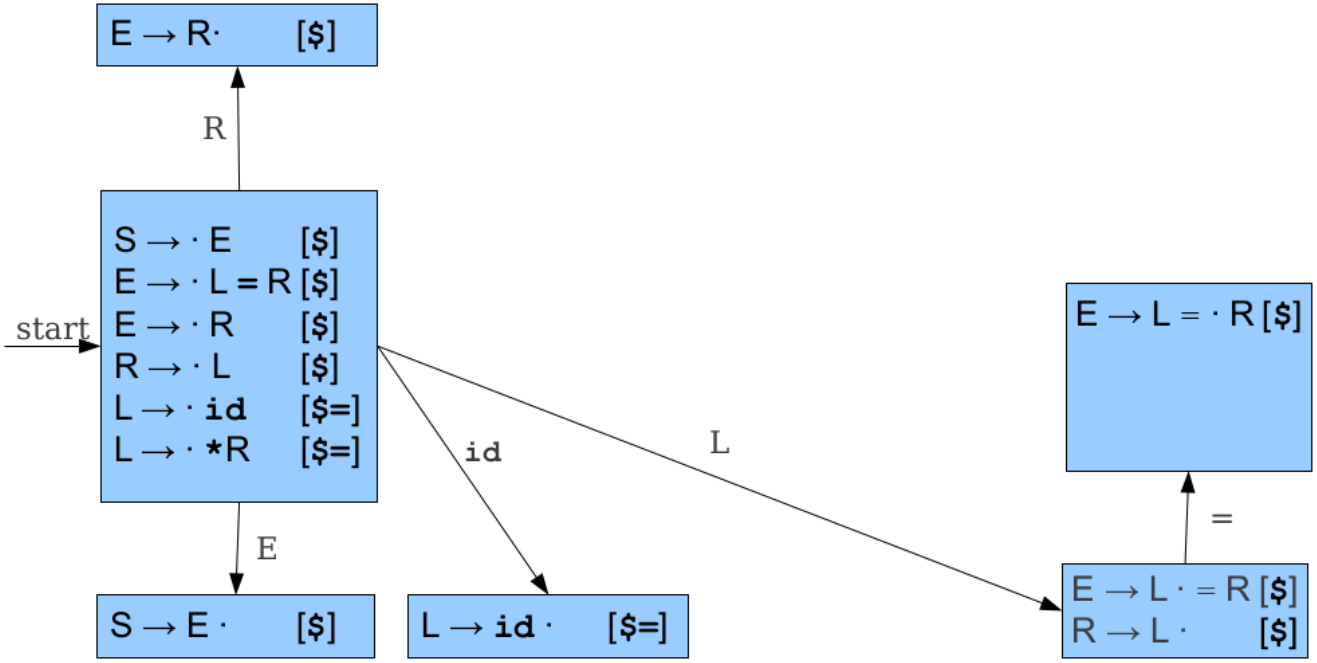
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



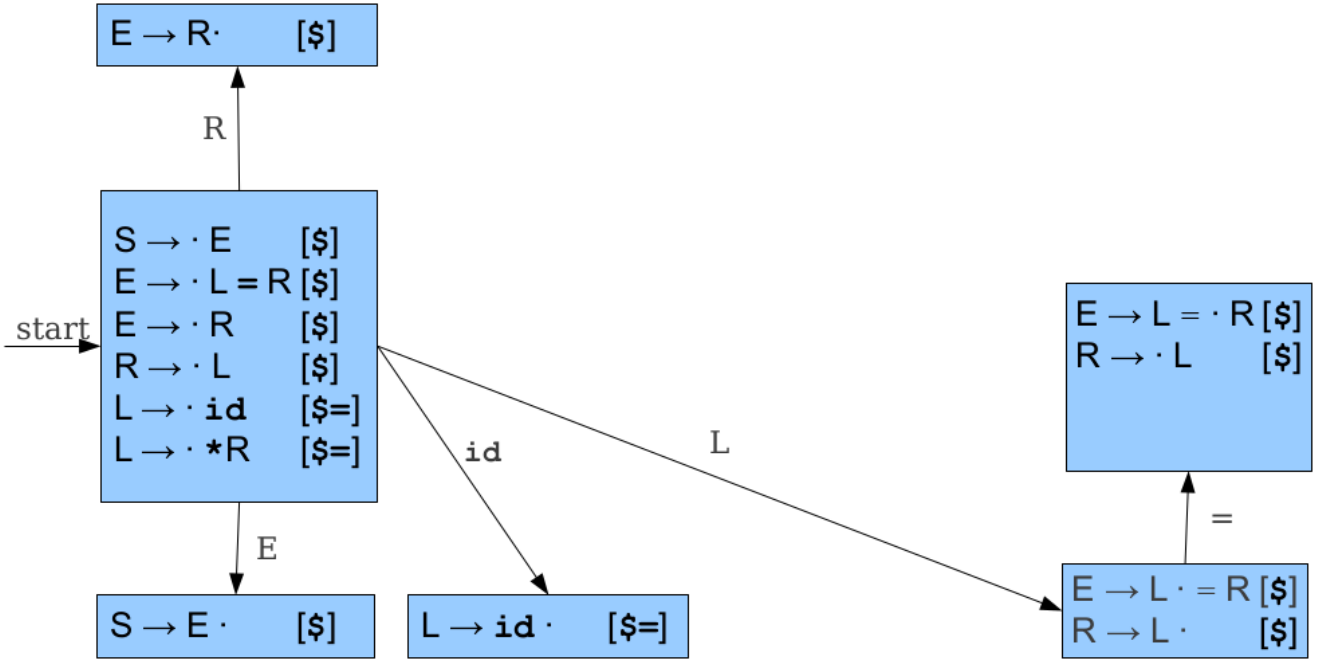
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



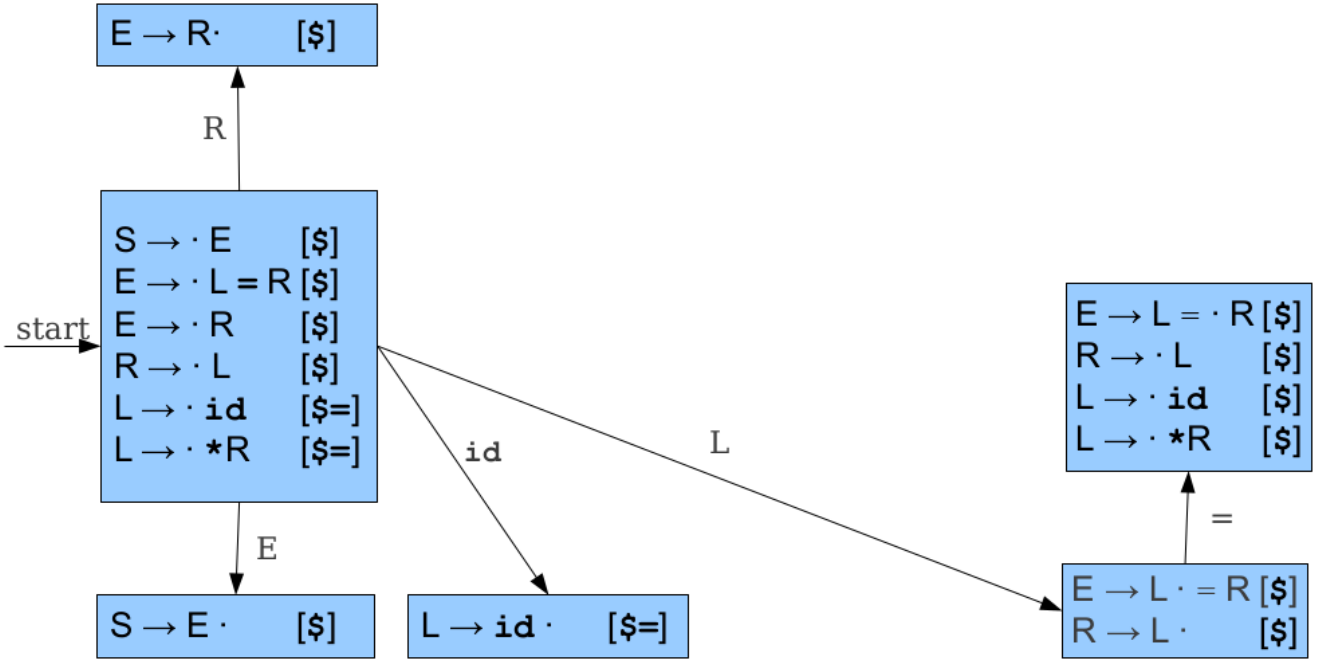
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



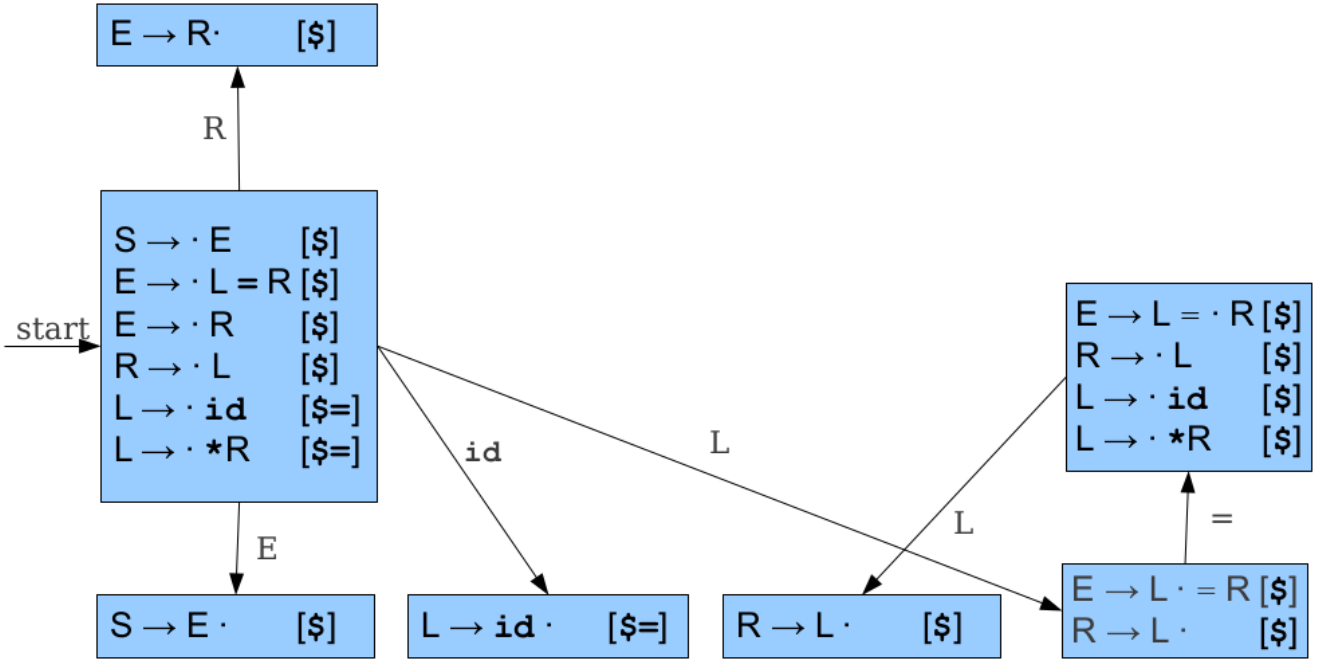
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



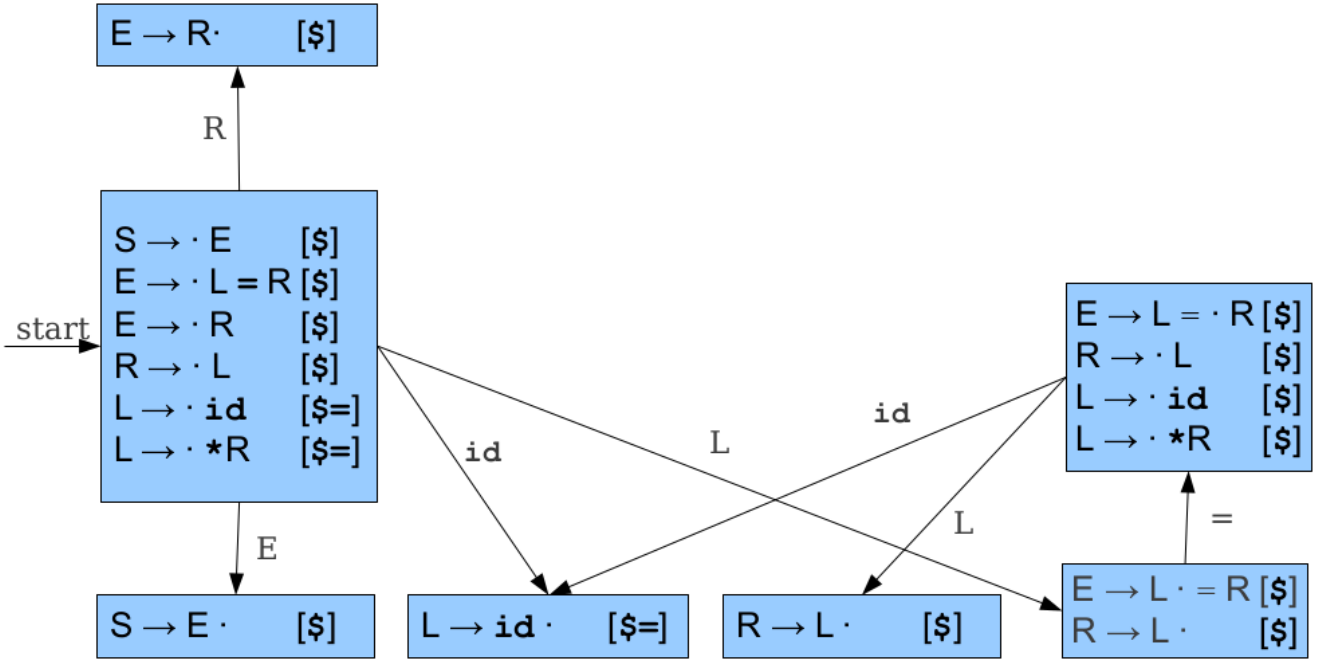
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



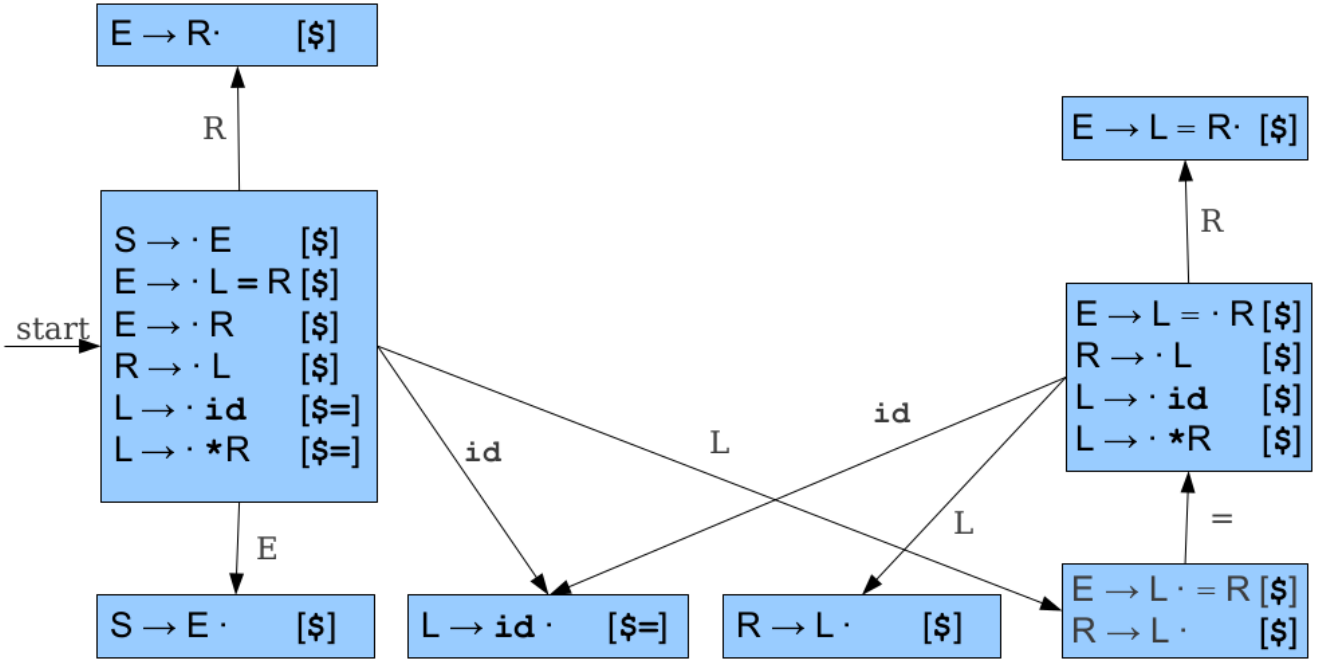
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



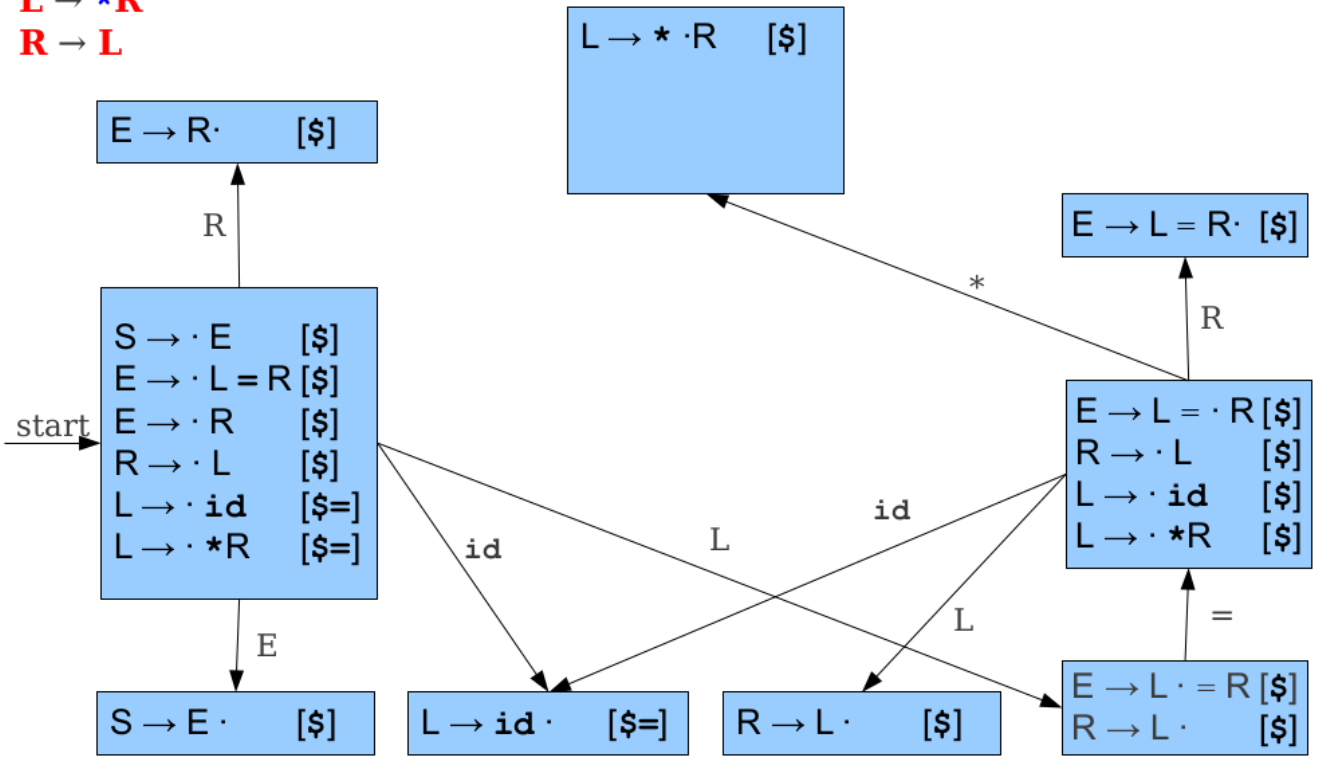
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



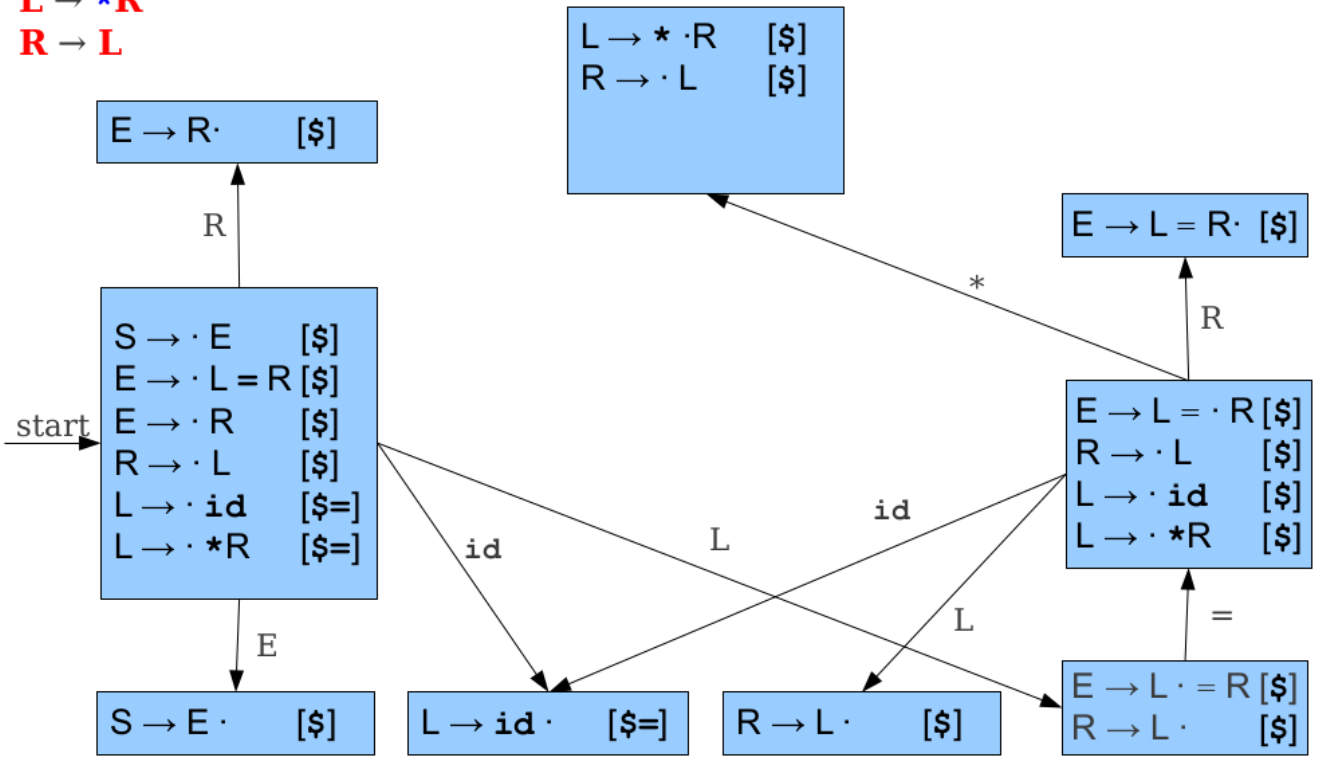
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



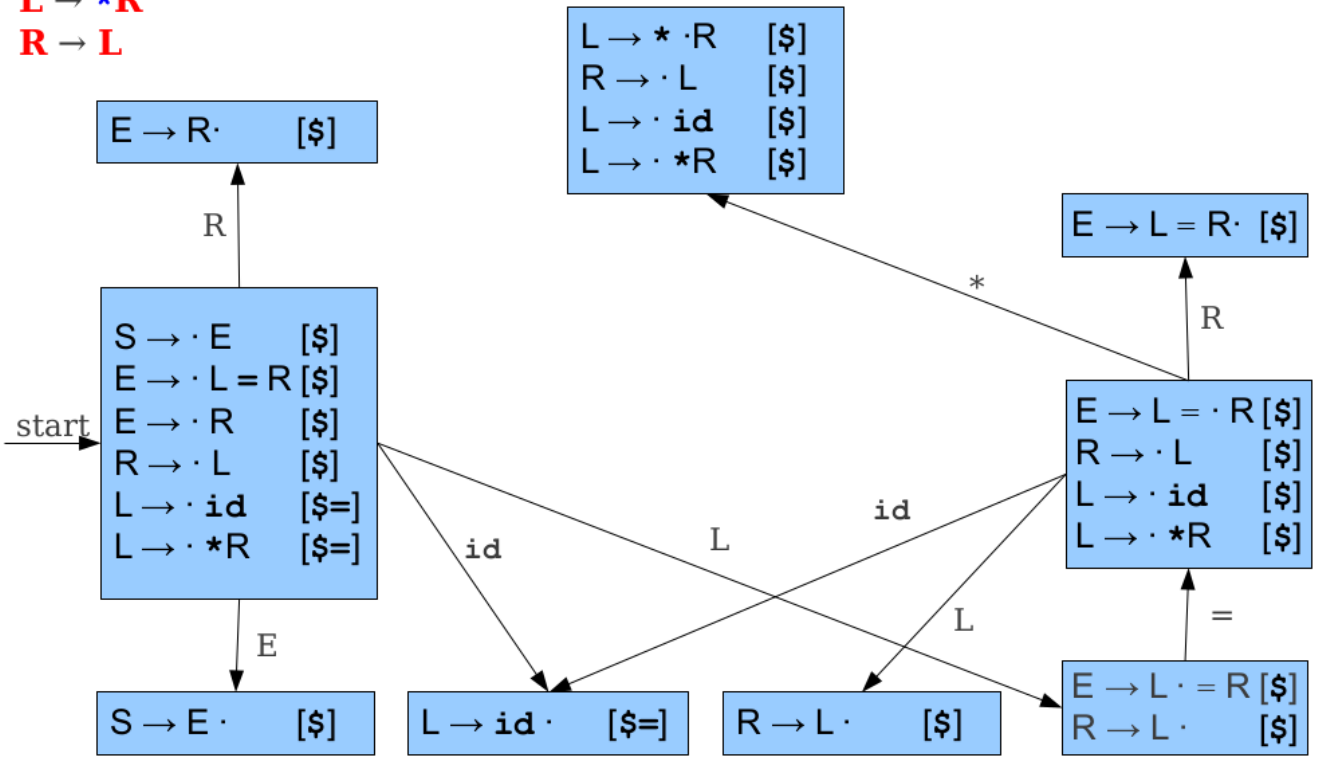
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



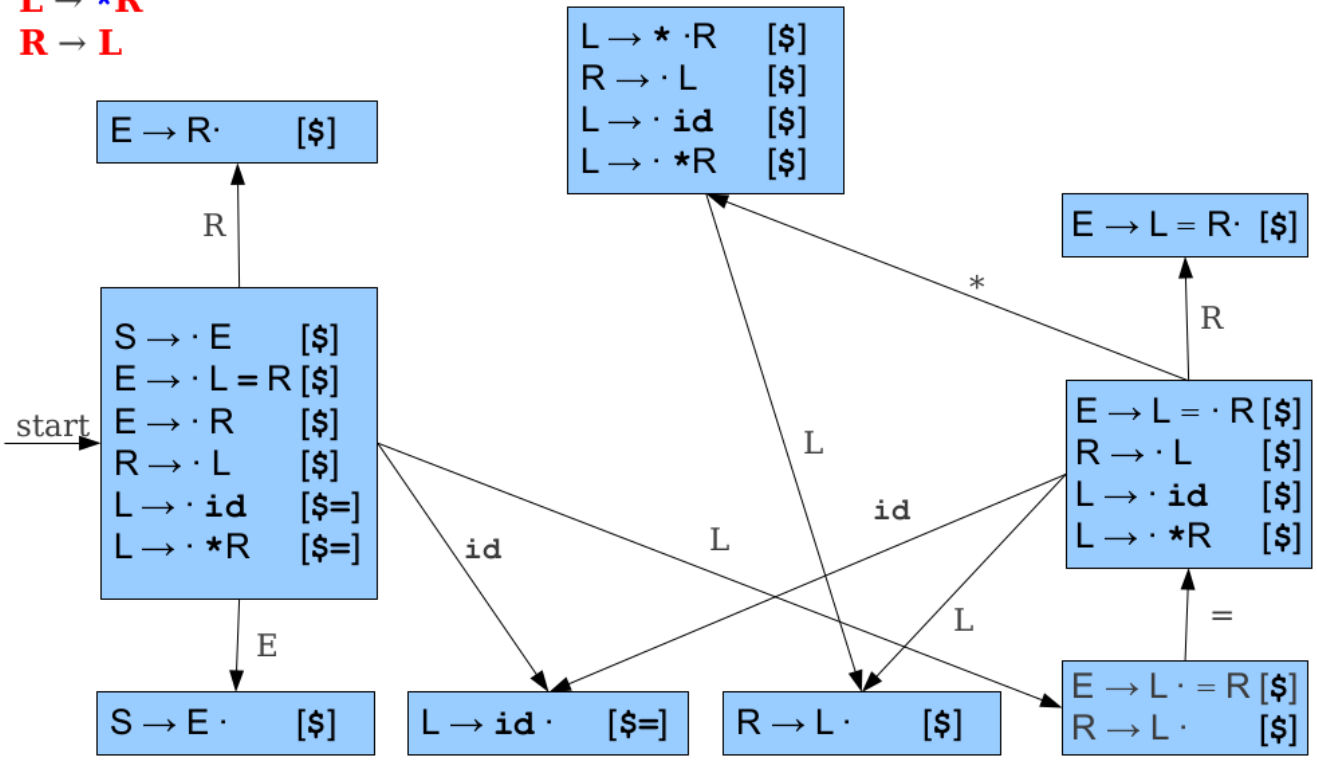
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



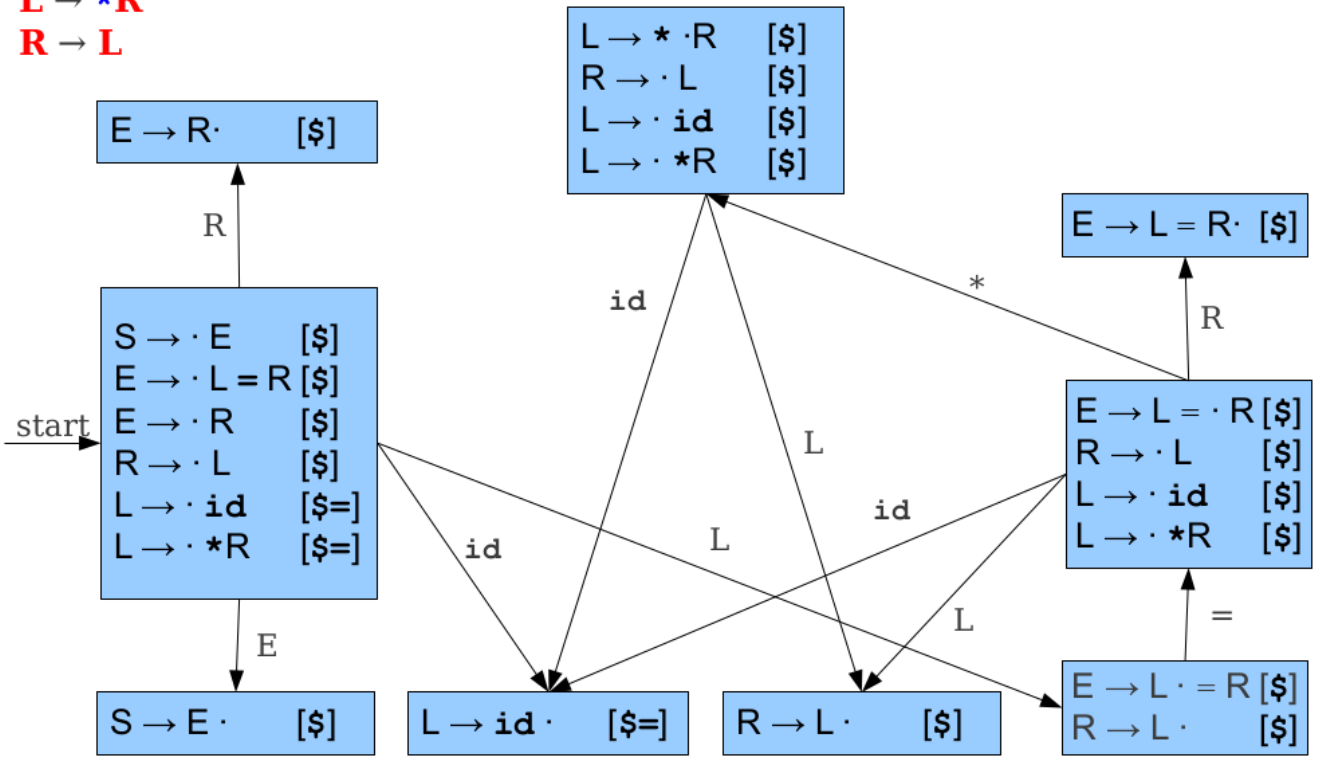
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



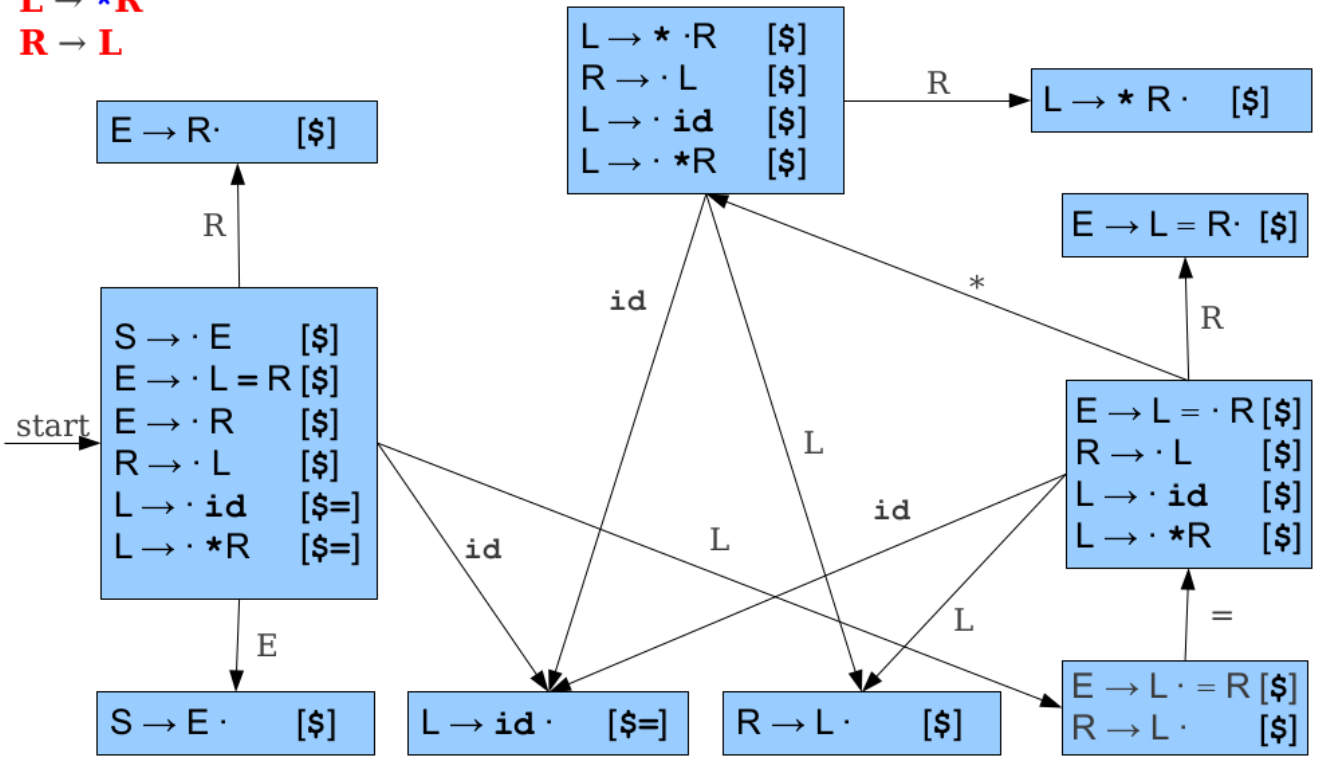
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



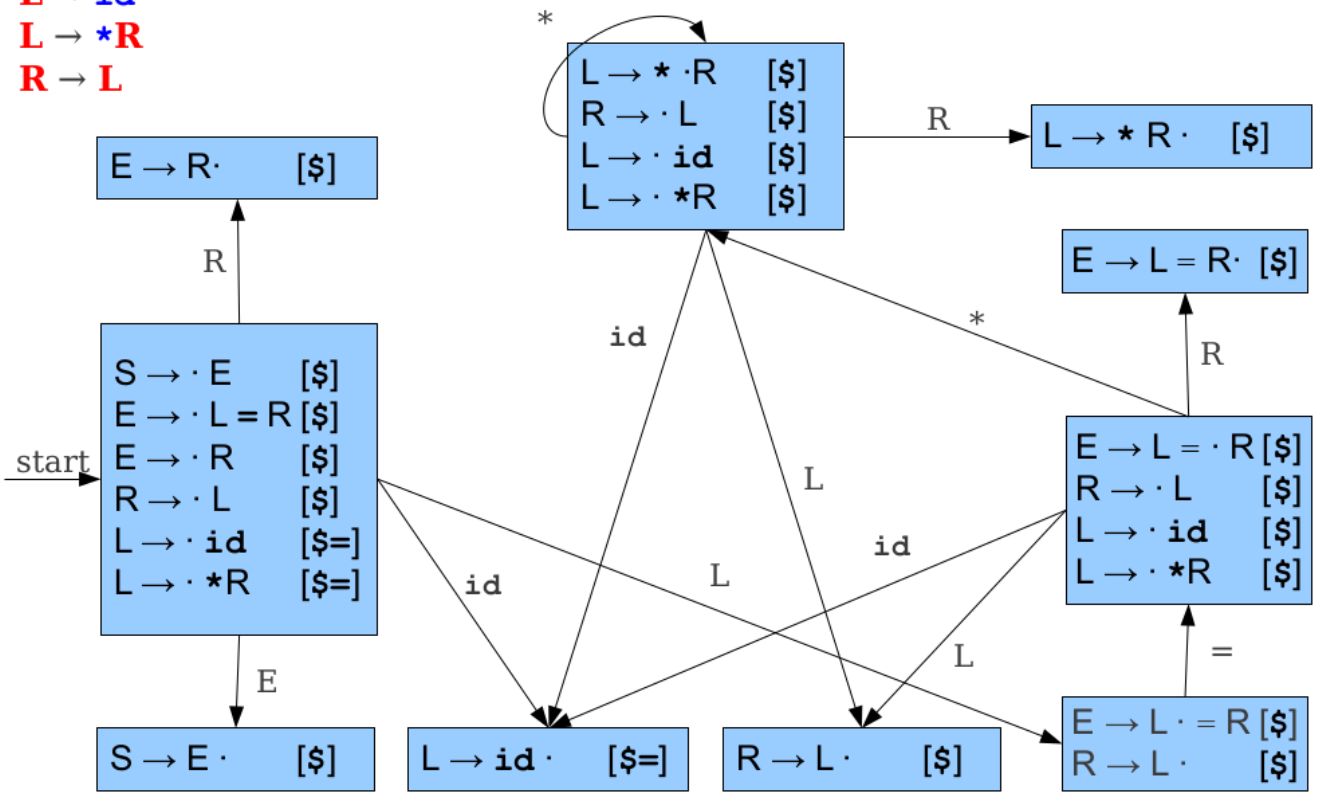
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



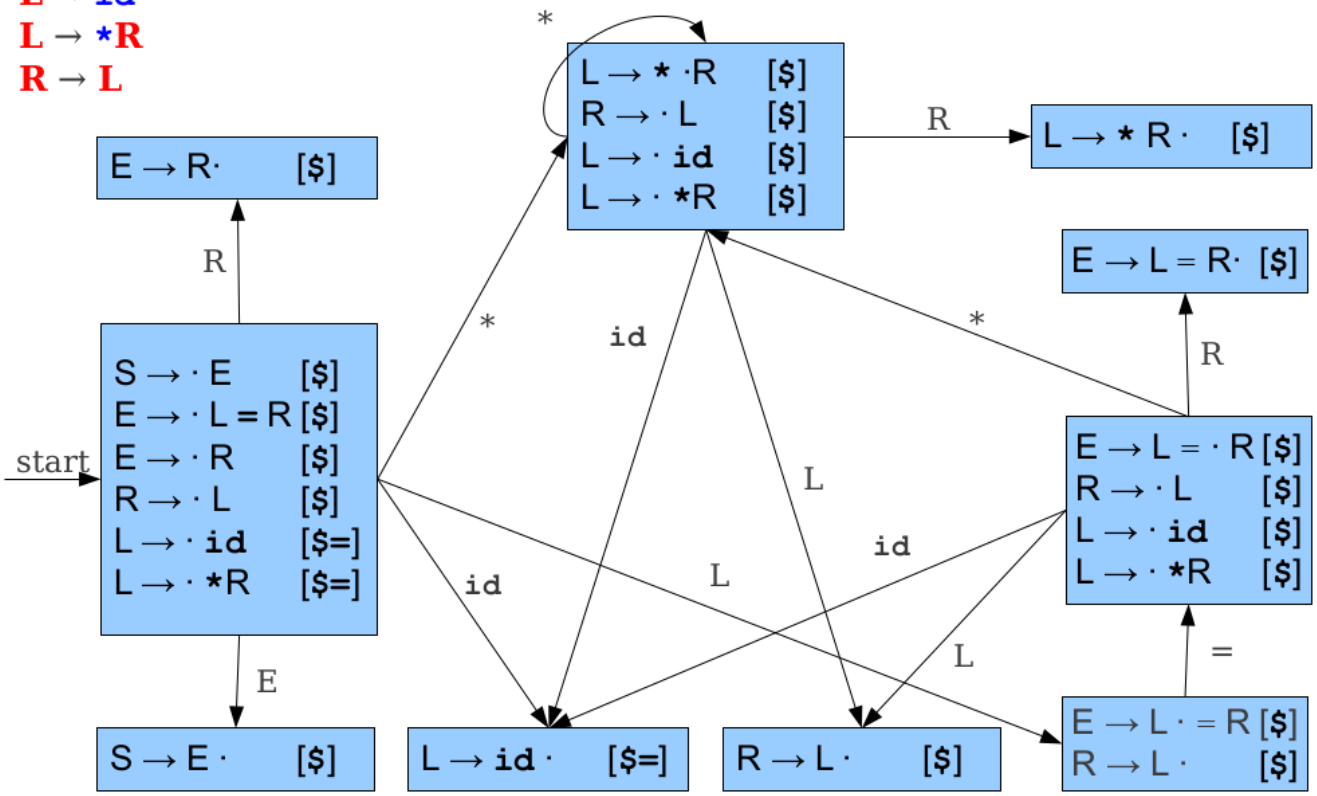
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



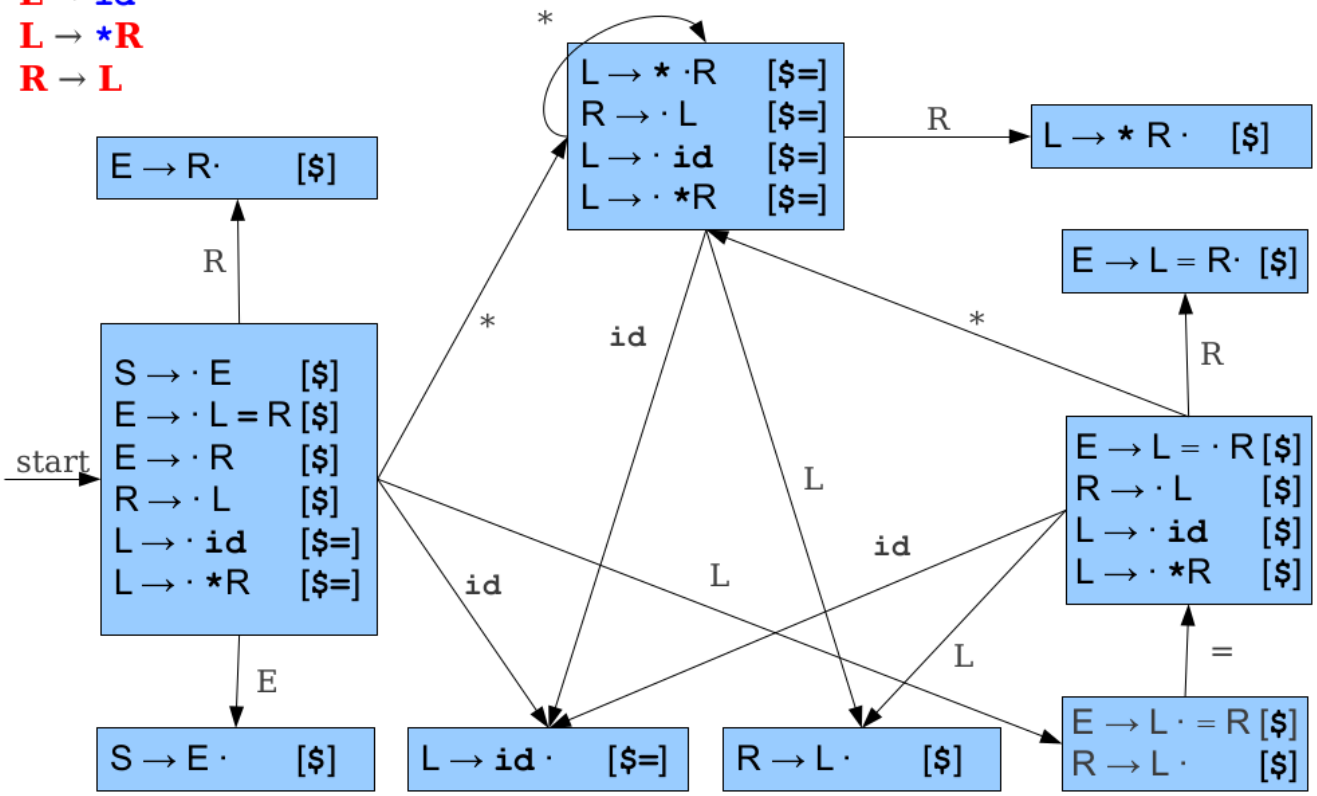
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



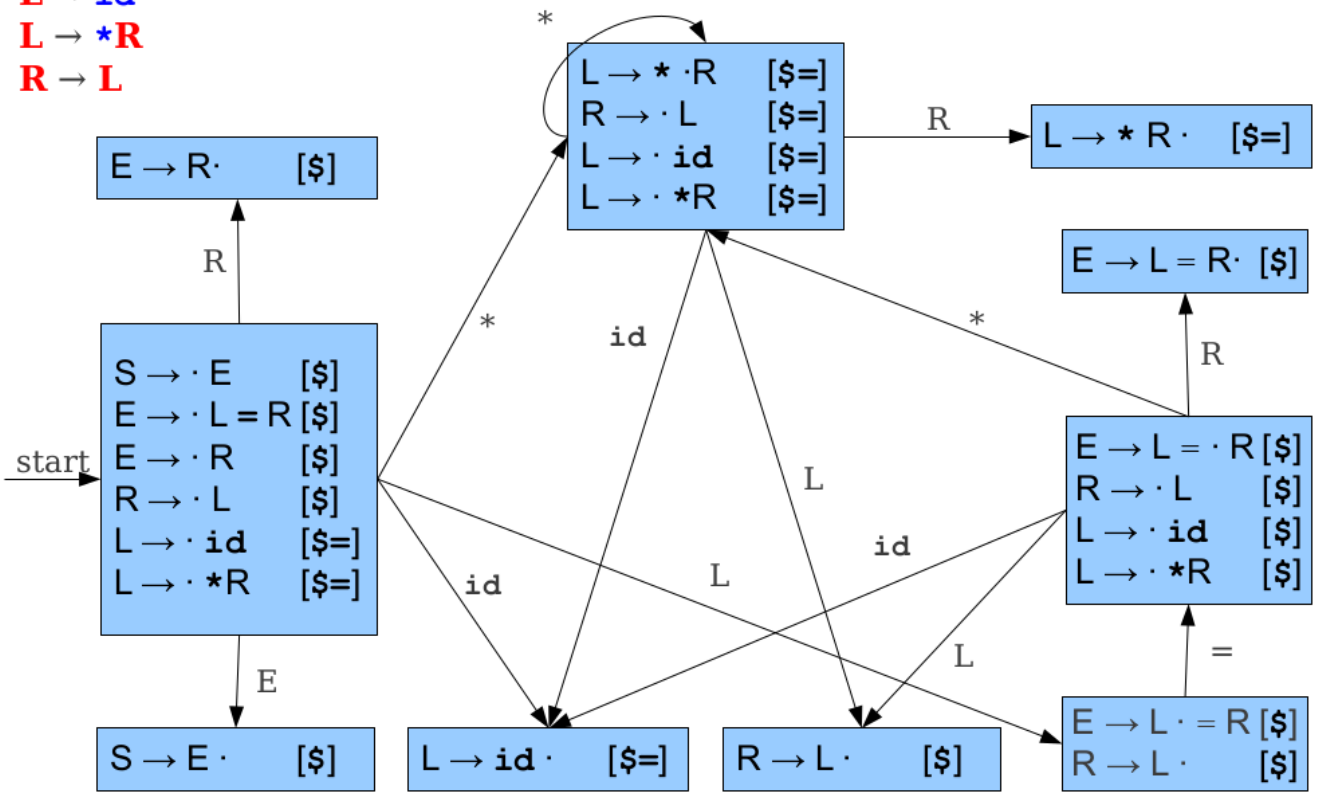
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



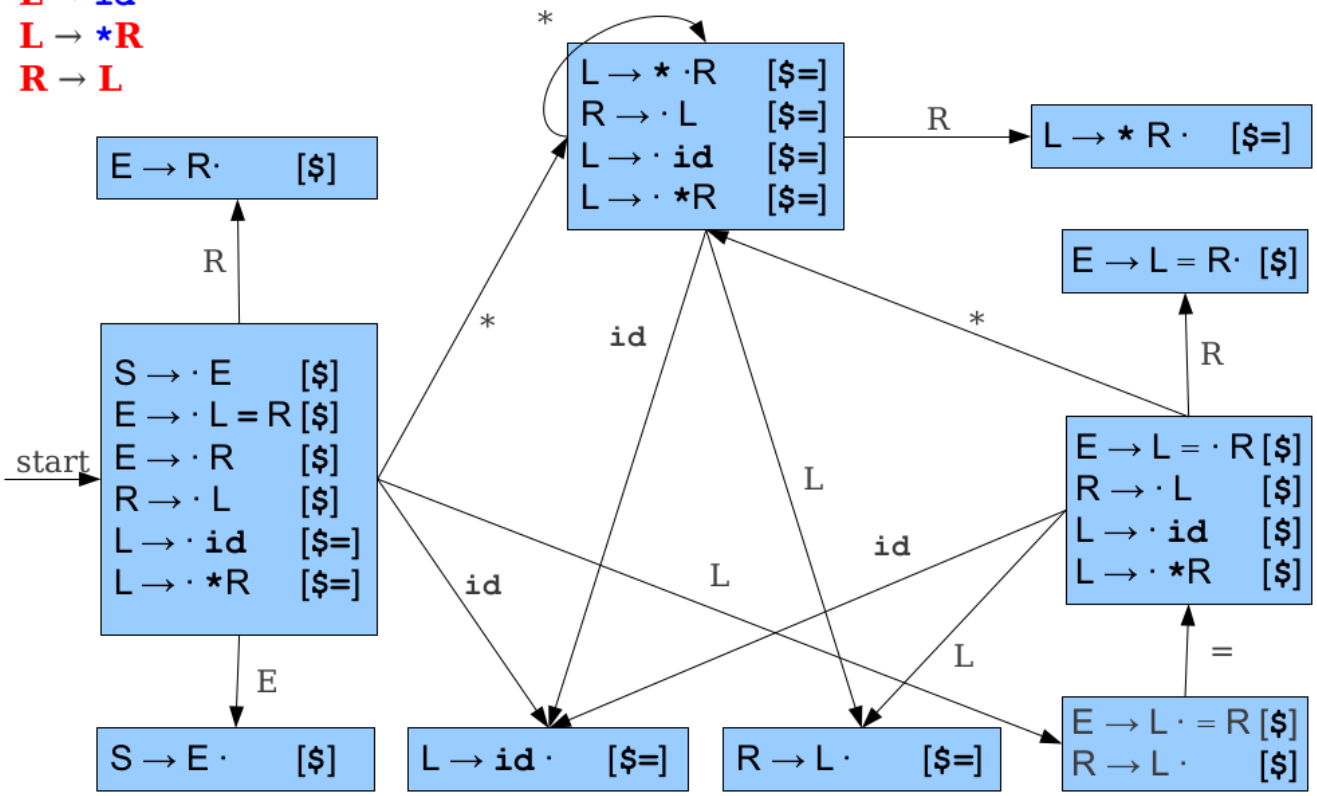
LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



LALR(1) Construction

S → E
E → L = R
E → R
L → id
L → *R
R → L



Analysis of our Algorithm

- Since we merge as we go, size of the partial automaton never exceeds size of overall automaton.
- However, this algorithm could be very slow in practice.
  - We might still have to generate all the LR(1) states, even if they immediately get merged.
- This can be very slow.

Example

1) E → E+T
2) E → T
3) T → T*F
4) T → F
5) F → (E)
6) F → id

Action Table | Goto Table

state	id	+	*	(	)	$		E	T	F
0	s5			s4				1	2	3
1		s6				acc				
2		r2	s7		r2	r2				
3		r4	r4		r4	r4				
4	s5			s4				8	2	3
5		r6	r6		r6	r6				
6	s5			s4					9	3
7	s5			s4						10
8		s6			s11					
9		r1	s7		r1	r1				
10		r3	r3		r3	r3				
11		r5	r5		r5	r5				

Example

stack	input	action	output
0	id*id+id$	shift 5	

Example

stack	input	action	output
0	id*id+id$	shift 5	
0id5	*id+id$	reduce by F→id	F→id
0F3	*id+id$	reduce by T→F	T→F
0T2	*id+id$	shift 7	
0T2*7	id+id$	shift 5	
0T2*7id5	+id$	reduce by F→id	F→id
0T2*7F10	+id$	reduce by T→T*F	T→T*F
0T2	+id$	reduce by E→T	E→T
0E1	+id$	shift 6	
0E1+6	id$	shift 5	
0E1+6id5	$	reduce by F→id	F→id
0E1+6F3	$	reduce by T→F	T→F
0E1+6T9	$	reduce by E→E+T	E→E+T
0E1	$	accept	

Outline

- Introduction
- Shift-reduce Parsing
- LR Parsing
- **Ambiguity**
- Error-handling

Ambiguity

- Real grammars are often ambiguous.
- How should we build a parser to try to combat it?

Ambiguity and Predictive Parsing

- The predictive parsers we have seen so far (LL(1), LR(0), SLR(1), LALR(1), LR(1)) only work on unambiguous grammars.
  - Intuitively: if grammar is ambiguous, cannot uniquely guess which production/reduction to use.
  - Formally proving this is somewhat involved.
- Most grammars for programming languages, unless cleverly written, are ambiguous.
- How can we handle this?

Parsing Ambiguous Grammars

- Consider this simple grammar for arithmetic expressions:

  $S \rightarrow E$

  $E \rightarrow E + E$

  $E \rightarrow E * E$

  $E \rightarrow \text{int}$

  $E \rightarrow (E)$

- This grammar is ambiguous.
  - e.g. Two trees for int + int * int
- What happens if we try parsing it?

SLR(1) Parsing with Ambiguity

1. **S** → **E**
2. **E** → **E + E**
3. **E** → **E * E**
4. **E** → **(E)**
5. **E** → **int**

FOLLOW(**S**) = { **$** }
FOLLOW(**E**) = { **+**, *****, **)** , **$** }

1
S → · E
E → · E + E
E → · E * E
E → · int
E → · (E)

2
S → E ·
E → E ·+ E
E → E ·* E

3
E → E + ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

4
E → E * ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

5
E → E + E ·
E → E · + E
E → E · * E

6
E → E * E ·
E → E · + E
E → E · * E

7
E → (·E)
E → · E + E
E → · E * E
E → · int
E → · (E)

8
E → (E ·)
E → E ·+ E
E → E ·* E

9
E → (E) ·

10
E → int ·

	int	+	*	(	)	$	E
1	s10			s7			s2
2		s3	s4			acc	
3	s10			s7			s5
4	s10			s7			s6
5		s3 r2	s4 r2		r2	r2	
6		s3 r3	s4 r3		r3	r3	
7	s10			s7			s8
8		s3	s4		s9		
9		r4	r4		r4	r4	
10		r5	r5		r5	r5	

Resolving Ambiguity

- Although the grammar is ambiguous, there is clearly one intended parse tree because of operator precedence.
- How can we use this precedence information to avoid LR conflicts?

Precedence Declarations

- Tell the parser generator about the associativity and precedence of certain rules.
- Productions can be left-associative, right-associative, or nonassociative.
- Productions can have their priorities ranked against one another.

SLR(1) Parsing with Ambiguity

1. **S → E**
2. **E → E + E** (Left-assoc, pri. 0)
3. **E → E * E** (Left-assoc, pri. 1)
4. **E → (E)**
5. **E → int**

1
S → · E
E → · E + E
E → · E * E
E → · int
E → · (E)

2
S → E ·
E → E ·+ E
E → E ·* E

3
E → E + ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

4
E → E * ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

5
E → E + E ·
E → E · + E
E → E · * E

6
E → E * E ·
E → E · + E
E → E · * E

7
E → (·E)
E → · E + E
E → · E * E
E → · int
E → · (E)

8
E → (E ·)
E → E ·+ E
E → E ·* E

9
E → (E) ·

10
E → int ·

	int	+	*	(	)	$	E
1	s10			s7			s2
2		s3	s4			acc	
3	s10			s7			s5
4	s10			s7			s6
5		s3 r2	s4 r2		r2	r2	
6		s3 r3	s4 r3		r3	r3	
7	s10			s7			s8
8		s3	s4		s9		
9		r4	r4		r4	r4	
10		r5	r5		r5	r5	

SLR(1) Parsing with Ambiguity

1. **S → E**
2. **E → E + E** (Left-assoc, pri. 0)
3. **E → E * E** (Left-assoc, pri. 1)
4. **E → (E)**
5. **E → int**

1
S → · E
E → · E + E
E → · E * E
E → · int
E → · (E)

2
S → E ·
E → E ·+ E
E → E ·* E

3
E → E + ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

4
E → E * ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

5
E → E + E ·
E → E · + E
E → E · * E

6
E → E * E ·
E → E · + E
E → E · * E

7
E → (·E)
E → · E + E
E → · E * E
E → · int
E → · (E)

8
E → (E ·)
E → E ·+ E
E → E ·* E

9
E → (E) ·

10
E → int ·

	int	+	*	(	)	$	E
1	s10			s7			s2
2		s3	s4			acc	
3	s10			s7			s5
4	s10			s7			s6
5		s3 r2	s4 r2		r2	r2	
6		s3 r3	s4 r3		r3	r3	
7	s10			s7			s8
8		s3	s4		s9		
9		r4	r4		r4	r4	
10		r5	r5		r5	r5	

SLR(1) Parsing with Ambiguity

1. **S → E**
2. **E → E + E** (Left-assoc, pri. 0)
3. **E → E * E** (Left-assoc, pri. 1)
4. **E → (E)**
5. **E → int**

1
S → · E
E → · E + E
E → · E * E
E → · int
E → · (E)

2
S → E ·
E → E ·+ E
E → E ·* E

3
E → E + ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

4
E → E * ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

5
E → E + E ·
E → E · + E
E → E · * E

6
E → E * E ·
E → E · + E
E → E · * E

7
E → (·E)
E → · E + E
E → · E * E
E → · int
E → · (E)

8
E → (E ·)
E → E ·+ E
E → E ·* E

9
E → (E) ·

10
E → int ·

	int	+	*	(	)	$	E
1	s10			s7			s2
2		s3	s4			acc	
3	s10			s7			s5
4	s10			s7			s6
5		s3 r2	s4		r2	r2	
6		s3 r3	s4 r3		r3	r3	
7	s10			s7			s8
8		s3	s4		s9		
9		r4	r4		r4	r4	
10		r5	r5		r5	r5	

SLR(1) Parsing with Ambiguity

1. **S → E**
2. **E → E + E** (Left-assoc, pri. 0)
3. **E → E * E** (Left-assoc, pri. 1)
4. **E → (E)**
5. **E → int**

1
S → · E
E → · E + E
E → · E * E
E → · int
E → · (E)

2
S → E ·
E → E ·+ E
E → E ·* E

3
E → E + ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

4
E → E * ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

5
E → E + E ·
E → E · + E
E → E · * E

6
E → E * E ·
E → E · + E
E → E · * E

7
E → (·E)
E → · E + E
E → · E * E
E → · int
E → · (E)

8
E → (E ·)
E → E ·+ E
E → E ·* E

9
E → (E) ·

10
E → int ·

	int	+	*	(	)	$	E
1	s10			s7			s2
2		s3	s4			acc	
3	s10			s7			s5
4	s10			s7			s6
5		s3 r2	s4		r2	r2	
6		s3 r3	s4 r3		r3	r3	
7	s10			s7			s8
8		s3	s4		s9		
9		r4	r4		r4	r4	
10		r5	r5		r5	r5	

SLR(1) Parsing with Ambiguity

1. **S → E**
2. **E → E + E** (Left-assoc, pri. 0)
3. **E → E * E** (Left-assoc, pri. 1)
4. **E → (E)**
5. **E → int**

1
S → · E
E → · E + E
E → · E * E
E → · int
E → · (E)

2
S → E ·
E → E ·+ E
E → E ·* E

3
E → E + ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

4
E → E * ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

5
E → E + E ·
E → E · + E
E → E · * E

6
E → E * E ·
E → E · + E
E → E · * E

7
E → (·E)
E → · E + E
E → · E * E
E → · int
E → · (E)

8
E → (E ·)
E → E ·+ E
E → E ·* E

9
E → (E) ·

10
E → int ·

	int	+	*	(	)	$	E
1	s10			s7			s2
2		s3	s4			acc	
3	s10			s7			s5
4	s10			s7			s6
5		s3 r2	s4		r2	r2	
6		r3	s4 r3		r3	r3	
7	s10			s7			s8
8		s3	s4		s9		
9		r4	r4		r4	r4	
10		r5	r5		r5	r5	

SLR(1) Parsing with Ambiguity

1. **S → E**
2. **E → E + E** (Left-assoc, pri. 0)
3. **E → E * E** (Left-assoc, pri. 1)
4. **E → (E)**
5. **E → int**

1
S → · E
E → · E + E
E → · E * E
E → · int
E → · (E)

2
S → E ·
E → E ·+ E
E → E ·* E

3
E → E + ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

4
E → E * ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

5
E → E + E ·
E → E · + E
E → E · * E

6
E → E * E ·
E → E · + E
E → E · * E

7
E → (·E)
E → · E + E
E → · E * E
E → · int
E → · (E)

8
E → (E ·)
E → E ·+ E
E → E ·* E

9
E → (E) ·

10
E → int ·

	int	+	*	(	)	$	E
1	s10			s7			s2
2		s3	s4			acc	
3	s10			s7			s5
4	s10			s7			s6
5		s3 r2	s4		r2	r2	
6		r3	s4 r3		r3	r3	
7	s10			s7			s8
8		s3	s4		s9		
9		r4	r4		r4	r4	
10		r5	r5		r5	r5	

SLR(1) Parsing with Ambiguity

1. **S → E**
2. **E → E + E** (Left-assoc, pri. 0)
3. **E → E * E** (Left-assoc, pri. 1)
4. **E → (E)**
5. **E → int**

1
S → · E
E → · E + E
E → · E * E
E → · int
E → · (E)

2
S → E ·
E → E ·+ E
E → E ·* E

3
E → E + ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

4
E → E * ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

5
E → E + E ·
E → E · + E
E → E · * E

6
E → E * E ·
E → E · + E
E → E · * E

7
E → (·E)
E → · E + E
E → · E * E
E → · int
E → · (E)

8
E → (E ·)
E → E ·+ E
E → E ·* E

9
E → (E) ·

10
E → int ·

	int	+	*	(	)	$	E
1	s10			s7			s2
2		s3	s4			acc	
3	s10			s7			s5
4	s10			s7			s6
5		r2	s4		r2	r2	
6		r3	r3		r3	r3	
7	s10			s7			s8
8		s3	s4		s9		
9		r4	r4		r4	r4	
10		r5	r5		r5	r5	

SLR(1) Parsing with Ambiguity

1. **S → E**
2. **E → E + E** (Rgt.-assoc, pri. 0)
3. **E → E * E** (Rgt.-assoc, pri. 1)
4. **E → (E)**
5. **E → int**

1
S → · E
E → · E + E
E → · E * E
E → · int
E → · (E)

2
S → E ·
E → E ·+ E
E → E ·* E

3
E → E + ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

4
E → E * ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

5
E → E + E ·
E → E · + E
E → E · * E

6
E → E * E ·
E → E · + E
E → E · * E

7
E → (·E)
E → · E + E
E → · E * E
E → · int
E → · (E)

8
E → (E ·)
E → E ·+ E
E → E ·* E

9
E → (E) ·

10
E → int ·

	int	+	*	(	)	$	E
1	s10			s7			s2
2		s3	s4			acc	
3	s10			s7			s5
4	s10			s7			s6
5		s3 r2	s4		r2	r2	
6		r3	s4 r3		r3	r3	
7	s10			s7			s8
8		s3	s4		s9		
9		r4	r4		r4	r4	
10		r5	r5		r5	r5	

SLR(1) Parsing with Ambiguity

1. **S → E**
2. **E → E + E** (Rgt.-assoc, pri. 0)
3. **E → E * E** (Rgt.-assoc, pri. 1)
4. **E → (E)**
5. **E → int**

1
S → · E
E → · E + E
E → · E * E
E → · int
E → · (E)

2
S → E ·
E → E ·+ E
E → E ·* E

3
E → E + ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

4
E → E * ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

5
E → E + E ·
E → E · + E
E → E · * E

6
E → E * E ·
E → E · + E
E → E · * E

7
E → (·E)
E → · E + E
E → · E * E
E → · int
E → · (E)

8
E → (E ·)
E → E ·+ E
E → E ·* E

9
E → (E) ·

10
E → int ·

	int	+	*	(	)	$	E
1	s10			s7			s2
2		s3	s4			acc	
3	s10			s7			s5
4	s10			s7			s6
5		s3 r2	s4		r2	r2	
6		r3	s4 r3		r3	r3	
7	s10			s7			s8
8		s3	s4		s9		
9		r4	r4		r4	r4	
10		r5	r5		r5	r5	

SLR(1) Parsing with Ambiguity

1. **S → E**
2. **E → E + E** (Rgt.-assoc, pri. 0)
3. **E → E * E** (Rgt.-assoc, pri. 1)
4. **E → (E)**
5. **E → int**

1
S → · E
E → · E + E
E → · E * E
E → · int
E → · (E)

2
S → E ·
E → E ·+ E
E → E ·* E

3
E → E + ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

4
E → E * ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

5
E → E + E ·
E → E · + E
E → E · * E

6
E → E * E ·
E → E · + E
E → E · * E

7
E → (·E)
E → · E + E
E → · E * E
E → · int
E → · (E)

8
E → (E ·)
E → E ·+ E
E → E ·* E

9
E → (E) ·

10
E → int ·

	int	+	*	(	)	$	E
1	s10			s7			s2
2		s3	s4			acc	
3	s10			s7			s5
4	s10			s7			s6
5		s3	s4		r2	r2	
6		r3	s4		r3	r3	
7	s10			s7			s8
8		s3	s4		s9		
9		r4	r4		r4	r4	
10		r5	r5		r5	r5	

Resolving Conflicts with Precedence

- When choosing whether to reduce a rule containing t or shift the terminal r:
  - If t has higher priority, reduce.
  - If r has higher priority, shift.
  - If t and r have the same priority:
    - If t is left-associative, reduce.
    - If t is right-associative, shift.
    - If t is non-associative, error.

Outline

- Introduction
- Shift-reduce Parsing
- LR Parsing
- Ambiguity
- **Error-handling**

Error-handling

- What should the parser do when it encounters an error?
  - How do you report errors intelligently?
    - Could just say “syntax error,” but we'd like more detailed messages.
  - How do you continue parsing after an error?
  - How do we resume parsing after an error?

SLR(1) Parsing with Ambiguity

1. **S** → **E**
2. **E** → **E + E** (Left-assoc, pri. 0)
3. **E** → **E * E** (Left-assoc, pri. 1)
4. **E** → **(E)**
5. **E** → **int**

1
S → · E
E → · E + E
E → · E * E
E → · int
E → · (E)

2
S → E ·
E → E ·+ E
E → E ·* E

3
E → E + ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

4
E → E * ·E
E → · E + E
E → · E * E
E → · int
E → · (E)

5
E → E + E ·
E → E · + E
E → E · * E

6
E → E * E ·
E → E · + E
E → E · * E

7
E → (·E)
E → · E + E
E → · E * E
E → · int
E → · (E)

8
E → (E ·)
E → E ·+ E
E → E ·* E

9
E → (E) ·

10
E → int ·

	int	+	*	(	)	$	E
1	s10			s7			s2
2		s3	s4			acc	
3	s10			s7			s5
4	s10			s7			s6
5		r2	s4		r2	r2	
6		r3	r3		r3	r3	
7	s10			s7			s8
8		s3	s4		s9		
9		r4	r4		r4	r4	
10		r5	r5		r5	r5	

Error-handling

- From simple to complex
  - Panic mode
  - Error productions
  - Automatic local or global correction

Panic Mode

- Simplest, most popular method
- When an error is detected:
  - Discard tokens until one with a clear role is found
  - Continue from there

- Such tokens are called synchronizing tokens
  - Typically the statement or expression terminators
  - Synchronizing tokens are chosen so that the parser can recover quickly after one is found.

Panic Mode

- Consider the erroneous expression
  - (1 + +2) + 3

- Panic-mode recovery:
  - Skip ahead to next integer and then continue

Using Panic Mode

- When parser encounters an error in a configurating set, search for a production containing an error term.
  - Repeatedly shift tokens onto the stack until the synchronizing token is found.
  - Reduce using the error rule.
- Resume parsing as normal.

Error Productions

- Idea: specify in the grammar known common mistakes
- Essentially promotes common errors to alternative syntax

Error Productions

- Example:
  - Write 5 x instead of 5 * x
- Grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow int$

$E \rightarrow (E)$

Error Productions

- Example:
  - Write 5 x instead of 5 * x
- Grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow \text{int}$

$E \rightarrow (E)$

$E \rightarrow E\ E$ (error production)

Error Productions

- Example:
  - Write 5 x instead of 5 * x
- Grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow int$

$E \rightarrow (E)$

$E \rightarrow E\ E$ (error production)

$E \rightarrow E +$ (error production)

$E \rightarrow E *$ (error production)

Error Productions

- Example:
  - Write 5 x instead of 5 * x
- Grammar:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow int$

$E \rightarrow (E)$

$E \rightarrow E\ E$ (error production)

$E \rightarrow E +$ (error production)

$E \rightarrow E *$ (error production)

- Disadvantage
  - Complicates the grammar

Summary

- We've just covered main powerful BU parsing algorithms:
  - LR(0)
  - LR(1)
  - SLR(1)
  - LALR(1)

Question?