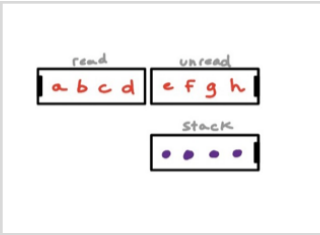
	Bottom Up	Top Down
Goes from:	input string → start symbol	start symbol → input string
Produces:	rightmost derivation	leftmost derivation
Works for:	left-associative grammars	right-associative grammars

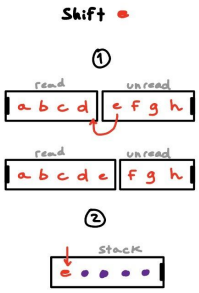
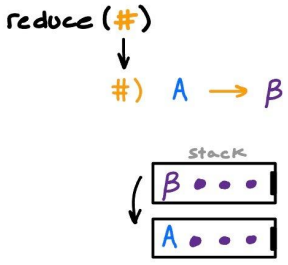
Algorithm Overview

3 Items:



1. **read**
2. **unread**
3. **stack**

2 Actions:

Shift	Reduce
	

Algorithm	
Read input 1 at a time, L → R:	
if [TOS == RHS of any rule]	**Reduce**(rule)
else	**shift**

Accept Input when:

1. all input read
2. stack contains just the start-symbol

Example

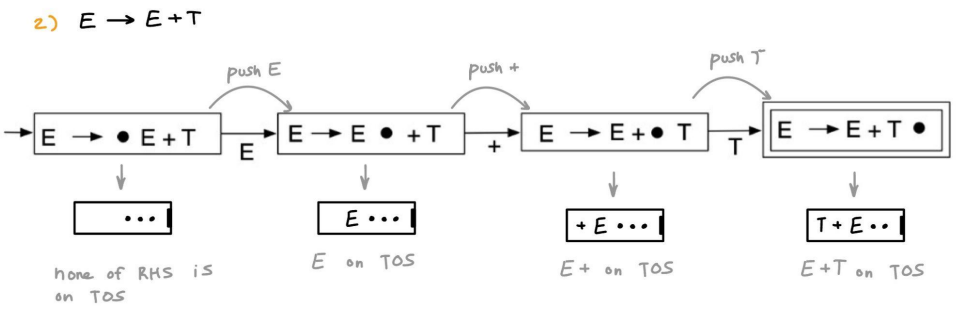
1) S` → ⊢ S ⊣
2) S → A y B
3) A → a b
4) A → c d
5) B → ε
6) B → w x

Input ⊢ a b y w x ⊣

Read	Unread	Stack	Action
ε	⊢ a b y w x ⊣		Shift ⊢
⊢	a b y w x ⊣	⊢	Shift a
⊢ a	b y w x ⊣	a ⊢	Shift b
⊢ a b	y w x ⊣	b a ⊢	reduce(3): pop b,a , push A
⊢ a b	y w x ⊣	y A ⊢	Shift y
⊢ a b y	w x ⊣	w y A ⊢	Shift w
⊢ a b y w	x ⊣	x w y A ⊢	Shift x
⊢ a b y w x	⊣	B y A ⊢	reduce(6): pop x,w , push B
⊢ a b y w x	⊣	S ⊢	reduce(2): pop ByA , push S
⊢ a b y w x ⊣	ε	⊣ S ⊢	Shift ⊣
⊢ a b y w x ⊣	ε	S`	reduce(1): pop ⊣S⊢ , push S`
⊢ a b y w x ⊣	ε	S`	Accept

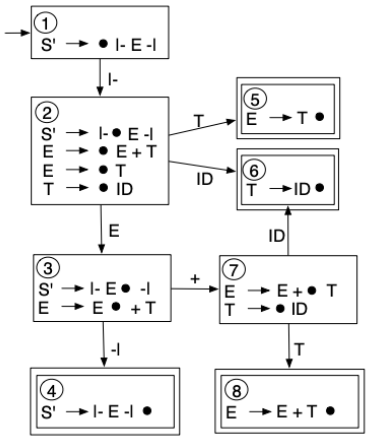
LR(0) Parsing DFA

- **states:** items (how much of the RHS of the rule is on the stack)
- **Accept States:** productions to reduce



Example:

$(1)\ S' \rightarrow \vdash\ \mathrm{E}\ \dashv$
$(2)\ \mathrm{E} \rightarrow \mathrm{E} + \mathrm{T}$
$(3)\ \mathrm{E} \rightarrow \mathrm{T}$
$(4)\ \mathrm{T} \rightarrow \mathrm{ID}$



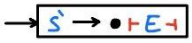
Building an LR(0) DFA

1	start state = 1st rule
2	For every NT/T to the right of the bookmark, X: • Create a transition to a new state on symbol X with the *bookmark* pushed 1 ahead
5	Mark states containing **reducible items** as accept-states
3	If a NT is to the right of a bookmark: • Add all rules to the state with X on the LHS Repeat
4	Repeat step 2-3 until no new states are discovered

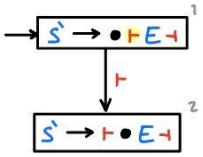
EX 1

1) S` → ⊢E⊣
2) E → E + T
3) E → T
4) T → ID

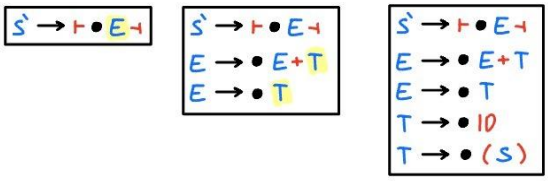
1



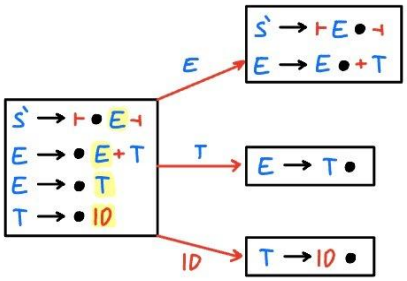
2

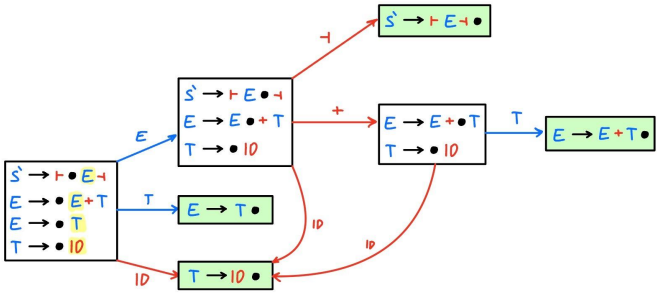


3



2



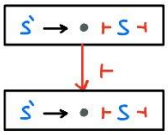


Example:

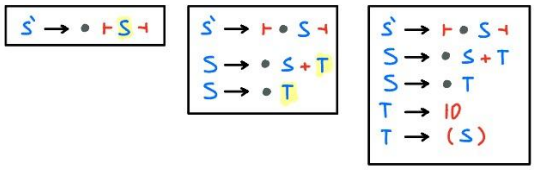
1) S` → ⊢ S ⊣
2) S → S + T
3) S → T
4) T → id
5) T → (S)

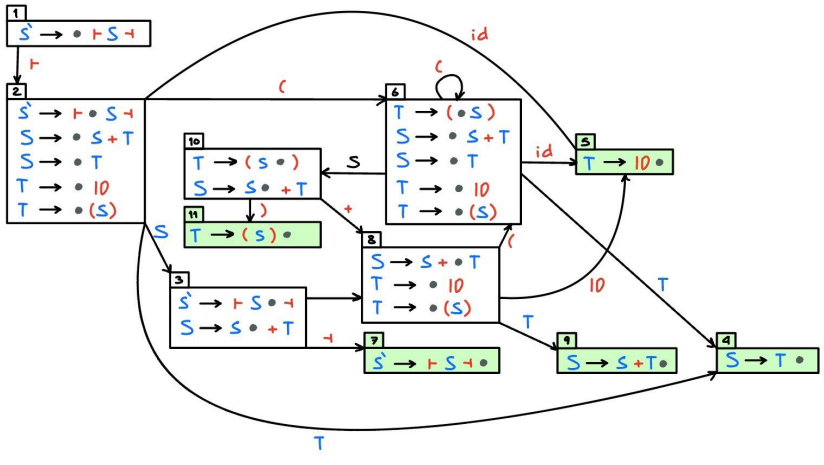
1

2



3





LR(0) parsing algorithm

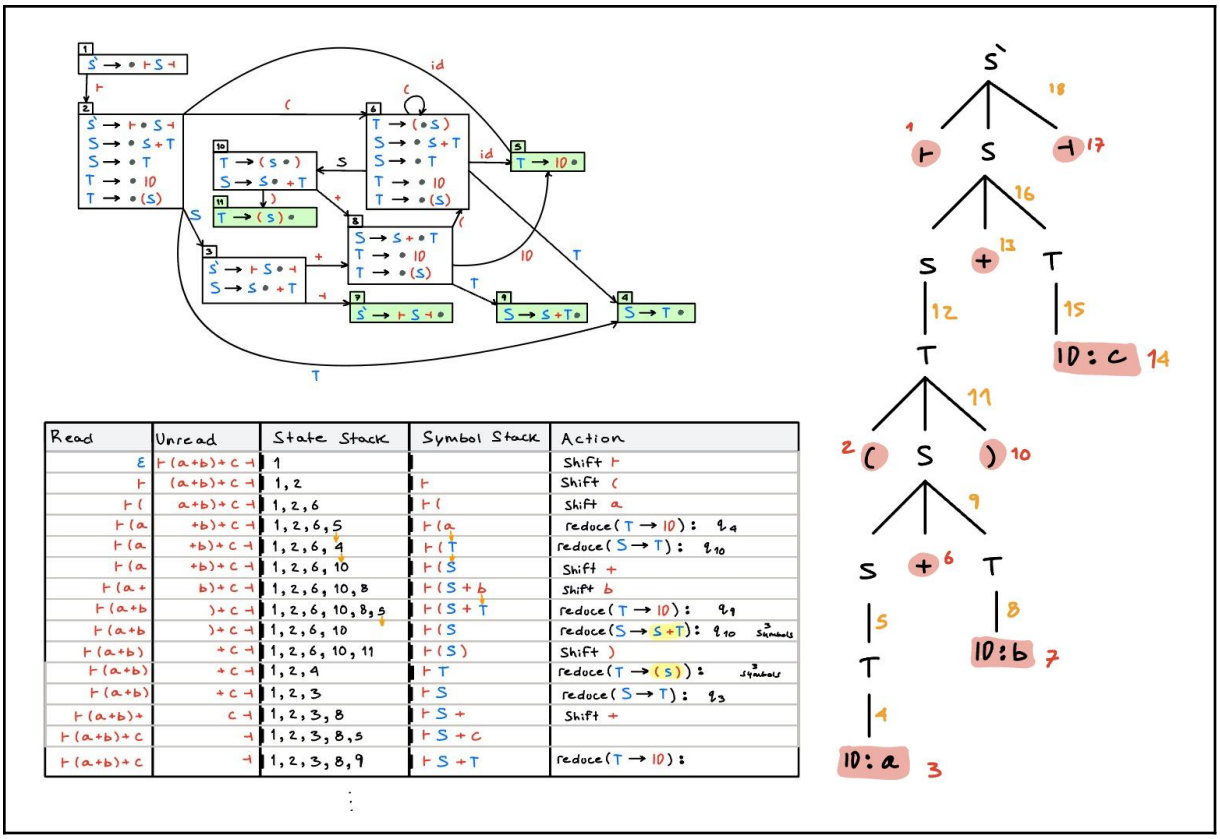
Run through **LR(0) DFA**

Items:

1. **Symbol Stack**
2. **State Stack:** DFA state numbers

1	State stack containing the start state
2	if [**reduce-state**] ● **reduce** ○ *n* times (# symbols on RHS of reduce-rule) ■ symbols.pop ■ states.pop ● symbols.push(LHS) ● states.push(DFA: new stateStack.top, new symbolStack.top) else: **shift** ● symbols.push(symbol) ● states.push(newState)
3	**Accept:** Pushed start symbol
4	**Reject:** 1. cannot reduce 2. can't shift

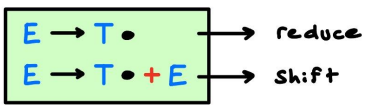
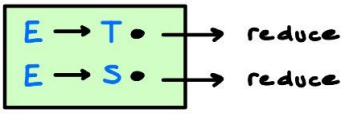
Example:



Conflicts

Right-Recursive grammars are not LR(0)

A grammar is LR(0) ⇔ LR(0) DFA does not have any **shift-reduce** or **reduce-reduce conflicts**.

Conflict	Why	Ex
Shift - Reduce	DFA state w' irreducible and reducible items	
Reduce - Reduce	When a state has two reducible items. Can't decide between the two.	

SLR(1)

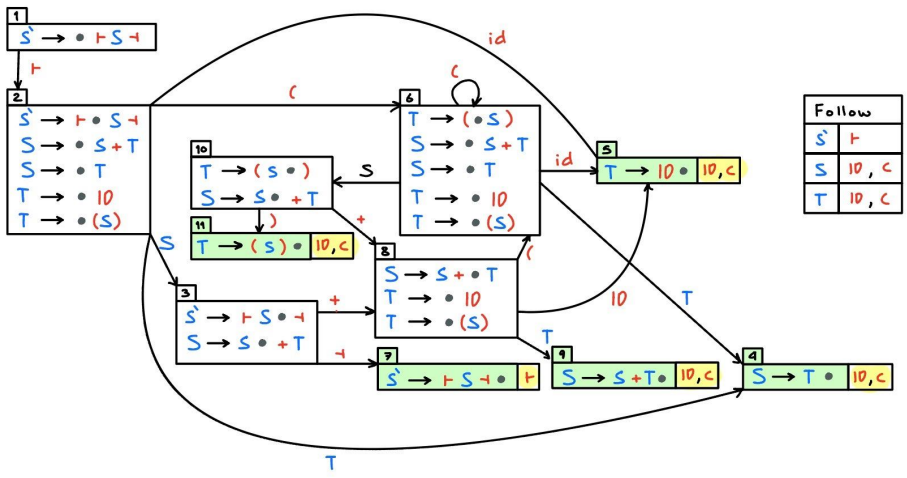
Simplified-LR(1) (1 lookahead)

Reduce (LHS → RHS)	**Shift**
Next Symbol is in **Follow(LHS)**	Otherwise

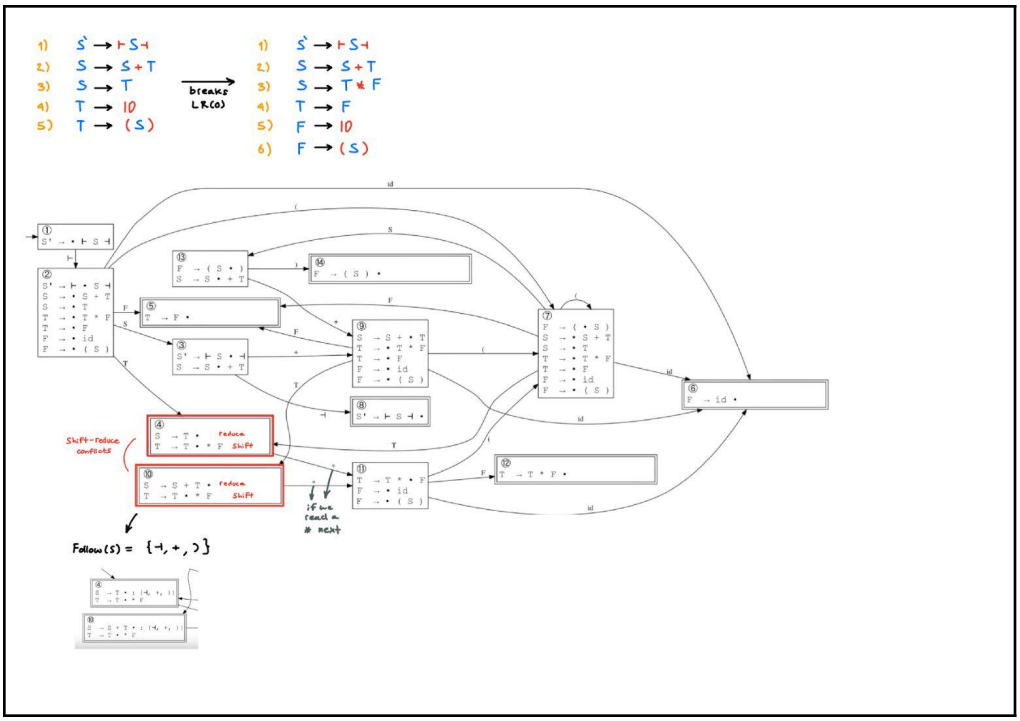
SLR(1) DFA

- **LR(0) DFA** + Follow Sets
- For each reducible item, add Follow(LHS)

Example:



Example 2:



SLR(1) Algorithm

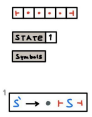
If No Conflict: reduce

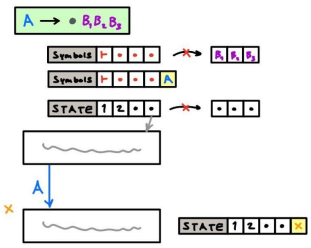
Reduce-Shift Conflict:

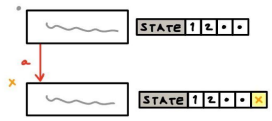
- next input symbol is in the FollowSet ⇒ **Reduce**
- else: **shift**

Reduce-Reduce:

- pick the reduction that has the next input symbol in its FollowSet

1	states.push (start-state)	
FOR EACH SYMBOL (x) IN INPUT		

2	while [**reduce-state**]
	• **reduce** x (# symbols in RHS)
	  ○ symbols.pop
	  ○ states.pop
	• symbols.push(LHS)
	• states.push(**newState**)
		

	else: **shift** • symbols.push(**x**) • states.push(newState)	
END IF:		
1	**Accept:** 1. EOF is on symbols 2. RHS of the rule for the grammar's start symbol is on the stack	
2	**Reject** 1. can't reduce 2. can't shift	

Generate Parse Trees

The algorithms have found a derivation. Given a derivation, a parse tree is easy to obtain.

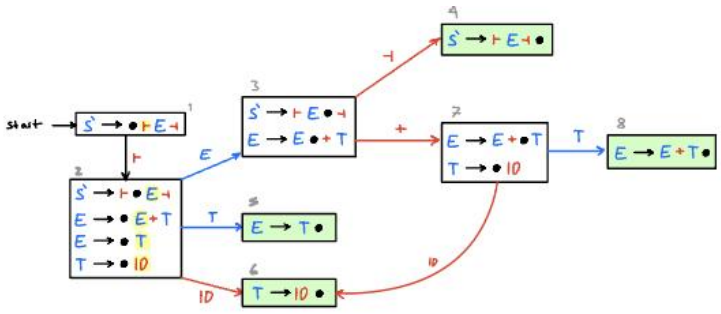
SymbolStack ⇒ TreeStack

	SymbolStack	**TreeStack**
shift	Push symbol onto symbolStack	Push a node onto the treeStackonto that contains the symbol
reduce	Pop the RHS from symbolStack Push the LHS onto symbolStack	Pops the subtree nodes that represent the RHS of the rule Push a new tree: • root = LHS • children = RHS

Example

1) S` → ⊢E⊣
2) E → E + T
3) E → T
4) T → ID

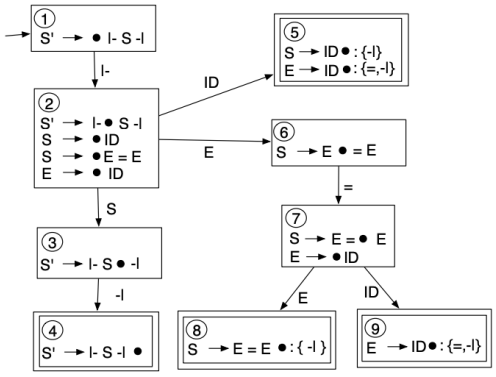
Input: ⊢ ID + ID ⊣



Read	States	Symbols	Action	Tree Stack
	1			
⊢	1,2	⊢	shift ⊢	⊢
⊢ ID	1,2,6	⊢ ID	shift ID	⊢ ID
⊢ ID	1,2,5	⊢ T	reduce (T→ID)	⊢ T(ID)
⊢ ID	1,2,3	⊢ E	reduce (T→E)	⊢ E(T(ID))
⊢ ID +	1,2,3,7	⊢ E +	shift +	⊢ E(T(ID)) +
⊢ ID + ID	1,2,3,7,6	⊢ E + ID	shift ID	⊢ E(T(ID)) + ID
⊢ ID + ID	1,2,3,7,8	⊢ E + T	reduce(T→ID)	⊢ E(T(ID)) + T(ID)
⊢ ID + ID	1,2,3	⊢ E	reduce(E→E+T)	⊢ E(E(T(ID)) + T(ID))
⊢ ID + ID ⊣	1,2,3,4	⊢ E	shift (⊣)	⊢ E(E(T(ID)) + T(ID)) ⊣
⊢ ID + ID ⊣	1,2,3,4	⊢ E ⊣	shift (⊣)	⊢ E(E(T(ID)) + T(ID)) ⊣
⊢ ID + ID ⊣	1	∅	reduce(S`→⊢E⊣)	S`(⊢ E(E(T(ID)) + T(ID)) ⊣)

SLR(1) Limitations

In the reduce-reduce conflict: the lookahead is in both follow sets

Example
$S' \rightarrow \vdash \ \mathrm{S} \ \dashv$ $\mathrm{S} \rightarrow \mathrm{ID}$ $\mathrm{S} \rightarrow \mathrm{E} = \mathrm{E}$ $\mathrm{E} \rightarrow \mathrm{ID}$ $\mathrm{Follow(S)} = \{\dashv\}$ $\mathrm{Follow(E)} = \{=, \dashv\}$ 
State 5: - Both have EOF in their following sets - If we have an EOF lookahead, which one do we pick?

Just because a symbol is in the Follow set of a non-terminal does not mean that there is a viable derivation if we rely on just that information.

Notes:

- Grammars that can be parsed by an LL(1) algorithm can also be parsed by an SLR(1) algorithm.
- There exist grammars that can be parsed by an LL(1) algorithm that cannot be parsed by an SLR(1) or LALR(1) algorithm.