Bottom-Up Parsing Algorithms

- LR(k) parsing
  - L: scan input Left to right
  - R: produce Rightmost derivation
  - k tokens of lookahead
- LR(0)
  - zero tokens of look-ahead
- SLR
  - Simple LR: like LR(0), but uses FOLLOW sets to build more "precise" parsing tables
  - LR(0) is a toy, so we focus on SLR
- Reading: Section 4.7

Problem: when to shift, when to reduce?

- Recall our favorite grammar:

  E → T + E | T

  T → int * T | int | (E)
- The step

  T * int + int → int * int + int

  is not part of any rightmost derivation
- Hence, reducing first int to T was a mistake
- *How to know when to reduce and when to shift?*

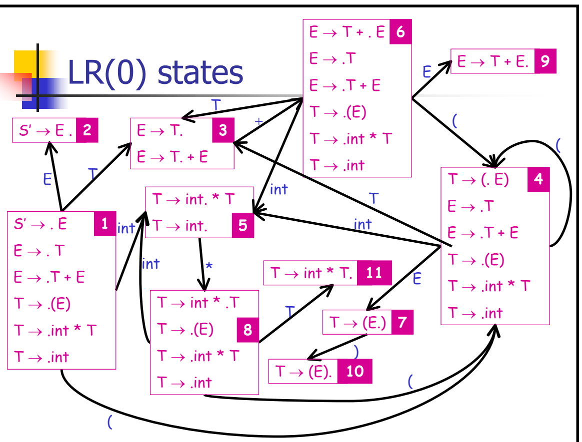
What we need for LR parsing

- LR(0) states
  - describe states in which the parser can be
  - Note: LR(0) states are used by both LR(0) and SLR parsers
- Parsing tables
  - transitions between LR(0) states,
  - actions to take when transiting:
    - shift, reduce, accept, error
- How to construct LR(0) states?
- How to construct parsing tables?
- How to drive the parser?

LR(0) state = set of LR(0) items

- An LR(0) item [X → α . β] says that
  - the parser is looking for an X
  - it has an α on top of the stack
  - expects to find input string derived from β
- Notes:
  - [X → α . aβ] means that if a is on the input, it can be shifted (resulting in αa . β). That is:
    - *a* is a correct token to see on the input, and
    - shifting *a* would not "over-shift" (still a viable prefix).
  - [X →α.] means that we could reduce α to X

LR(0) states



Naïve SLR Parsing Algorithm

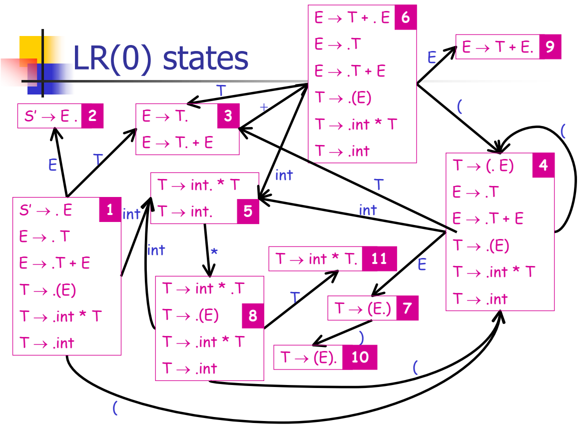
1. Let M be LR(0) state machine for G
   - each state contains a set I of LR(0) items
2. Let $|x_1 \ldots x_n\$$ be initial configuration
3. Repeat until configuration is S|$
   - Let α|ω be current configuration
   - Run M on current stack α
   - If M rejects α, report parsing error
   - If M accepts α, let a be next input
     - Shift if [X → β. a γ] ∈ Items
     - Reduce if [X → β.] ∈ Items and a ∈ Follow(α)

       … β | a … → … | X a …
     - Report parsing error if neither applies

Notes

- If there is a conflict in the last step, grammar is not SLR(k)
- k is the amount of lookahead
  - In practice k = 1

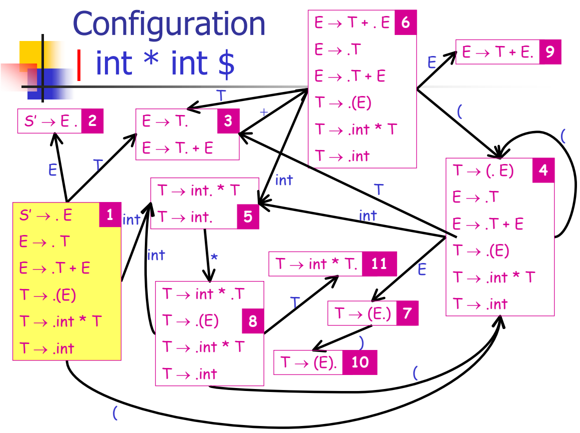
LR(0) states



SLR Example

Configuration	*DFA Halt State*	*Action*
| int * int $	1	

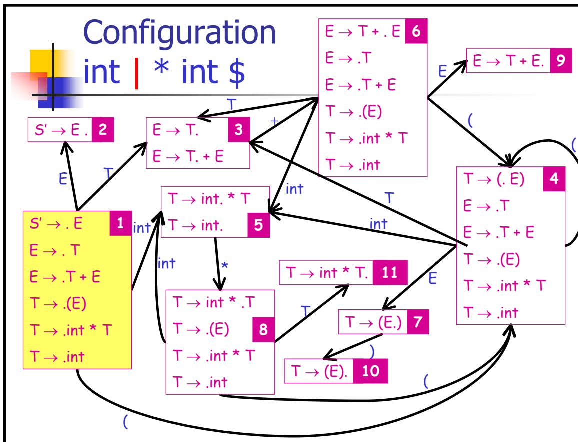
Configuration
| int * int $



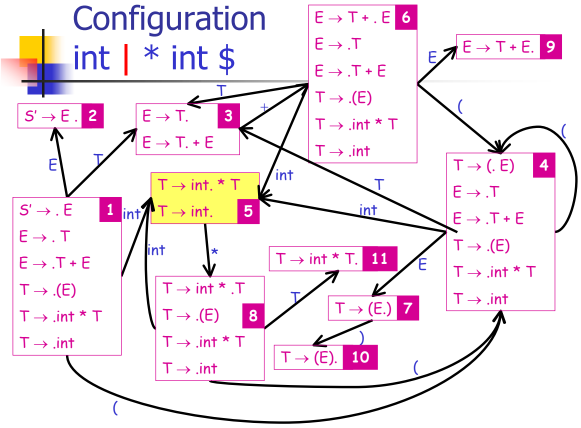
SLR Example

Configuration	*DFA Halt State*	*Action*
| int * int $	1	shift
int | * int $	5	

Configuration
int | * int $



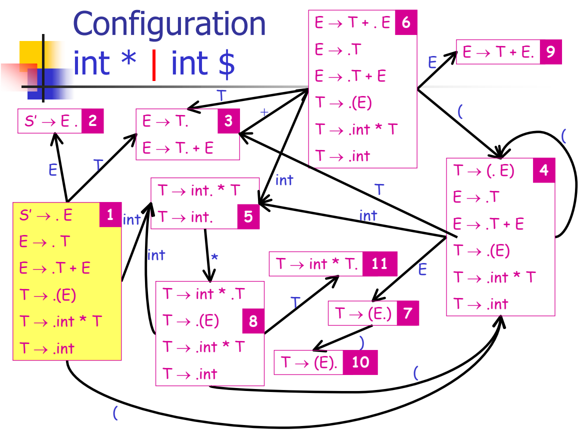
Configuration
int | * int $



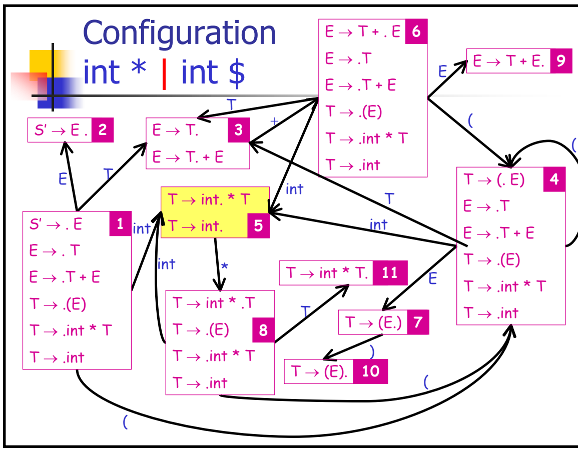
SLR Example

Configuration	*DFA Halt State*	*Action*
| int * int $	1	shift
int | * int $	5 * not in Follow(T)	shift
int * | int $	8	

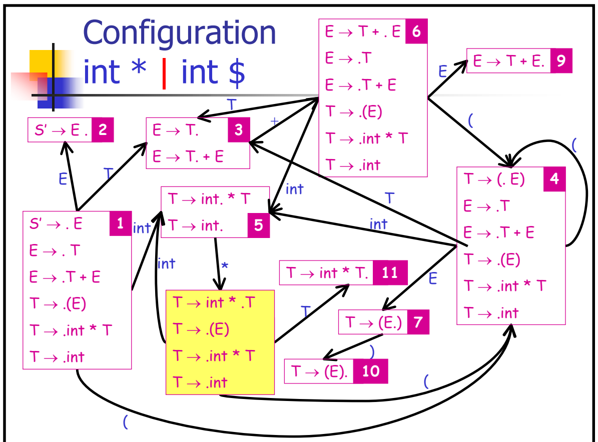
Configuration
int * | int $



Configuration
int * | int $



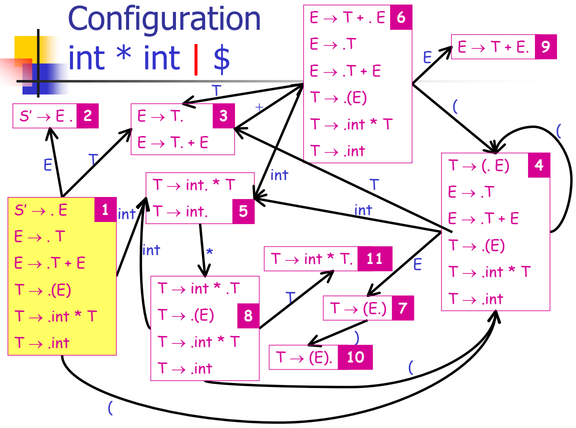
Configuration
int * | int $



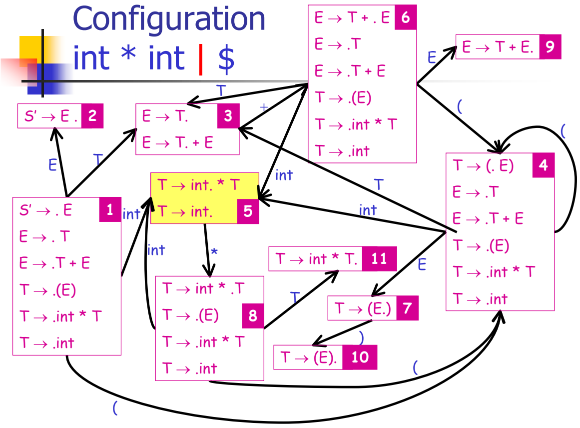
SLR Example

Configuration	*DFA Halt State*	*Action*
| int * int $	1	shift
int | * int $	5 * not in Follow(T)	shift
int * | int $	8	shift
int * int | $	5	

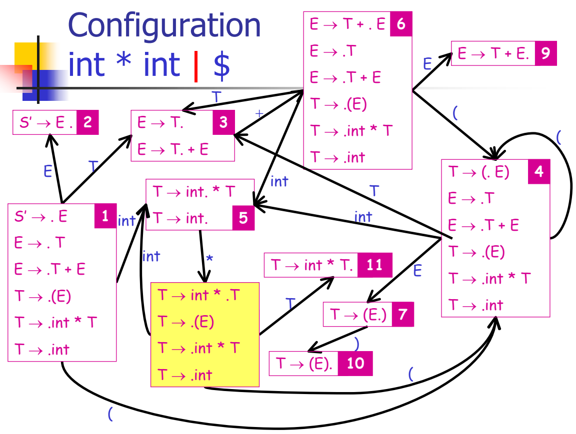
Configuration
int * int | $



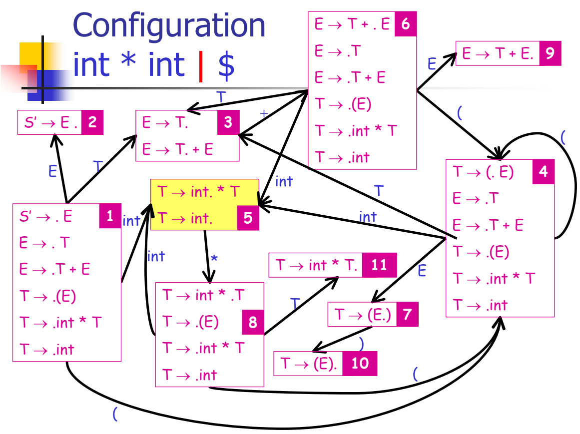
Configuration
int * int | $



Configuration
int * int | $



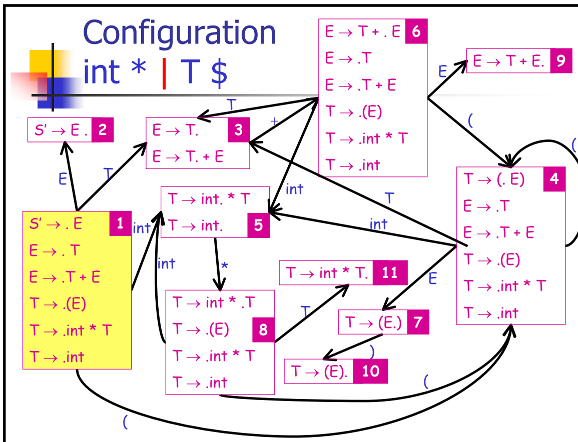
Configuration
int * int | $



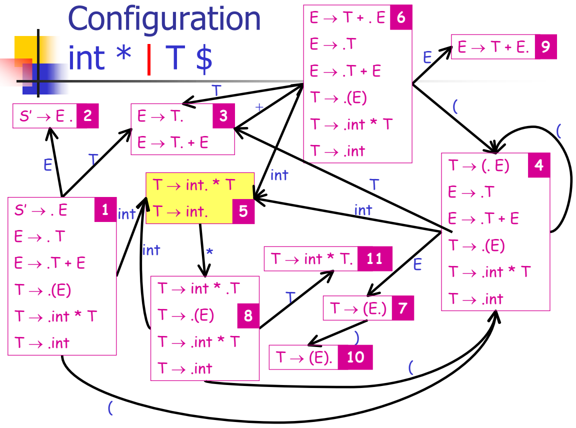
SLR Example

Configuration	*DFA Halt State*	*Action*
| int * int $	1	shift
int | * int $	5 * not in Follow(T)	shift
int * | int $	8	shift
int * int | $	5 $ ∈ Follow(T)	reduce T→int
int * | T $	8	

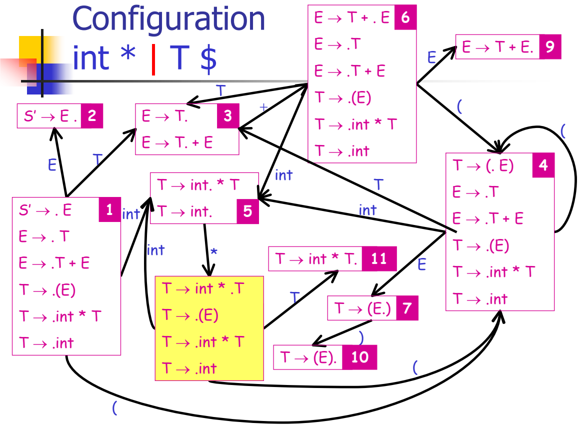
Configuration
int * | T $



Configuration
int * | T $



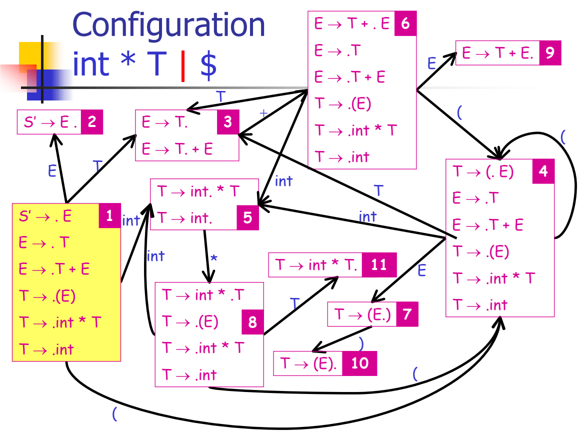
Configuration
int * | T $



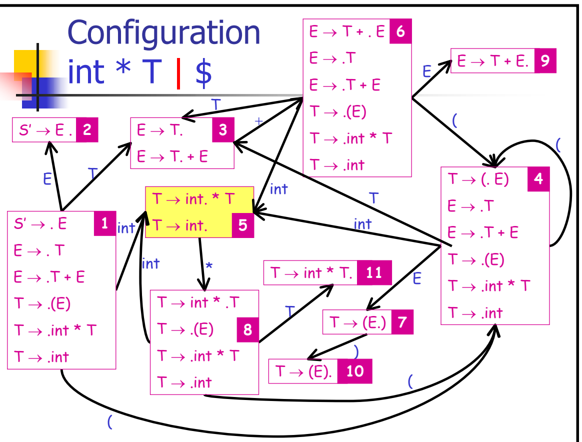
SLR Example

Configuration	*DFA Halt State*	*Action*
| int * int $	1	shift
int | * int $	5 * not in Follow(T)	shift
int * | int $	8	shift
int * int | $	5 $ ∊ Follow(T)	reduce T→int
int * | T $	8	shift
int * T | $	11	

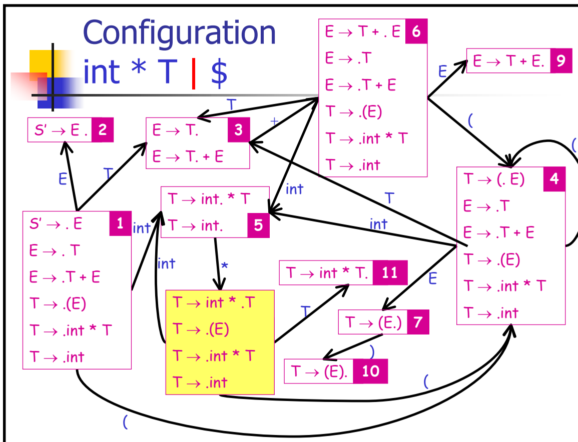
Configuration
int * T | $



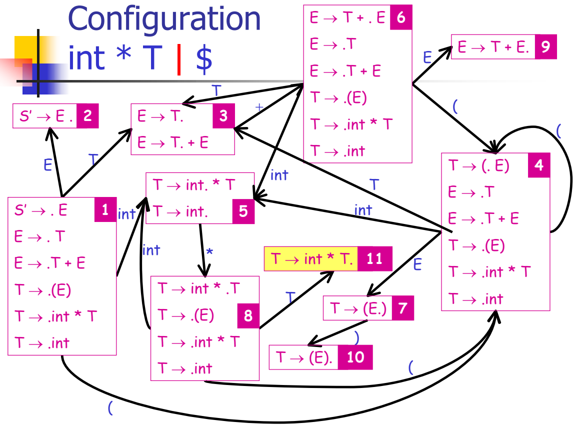
Configuration
int * T | $



Configuration
int * T | $



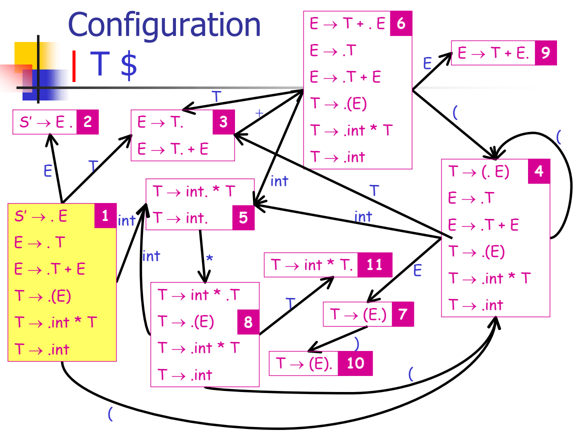
Configuration
int * T | $



SLR Example

Configuration	*DFA Halt State*	*Action*
| int * int $	1	shift
int | * int $	5 * not in Follow(T)	shift
int * | int $	8	shift
int * int | $	5 $ ∈ Follow(T)	reduce T→int
int * | T $	8	shift
int * T | $	11 $ ∈ Follow(T)	reduce T→int * T
| T $	1	

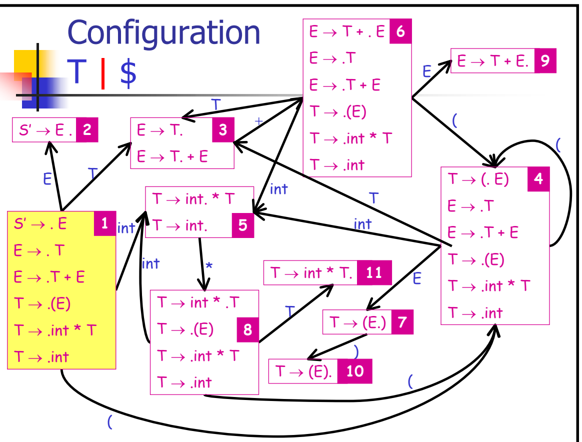
Configuration
| T $



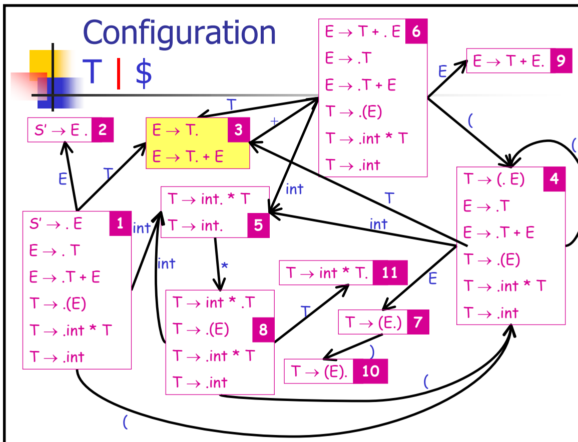
SLR Example

Configuration	*DFA Halt State*	*Action*
| int * int $	1	shift
int | * int $	5 * not in Follow(T)	shift
int * | int $	8	shift
int * int | $	5 $ ∈ Follow(T)	reduce T→int
int * | T $	8	shift
int * T | $	11 $ ∈ Follow(T)	reduce T→int * T
| T $	1	shift
T | $	3	

Configuration
T | $



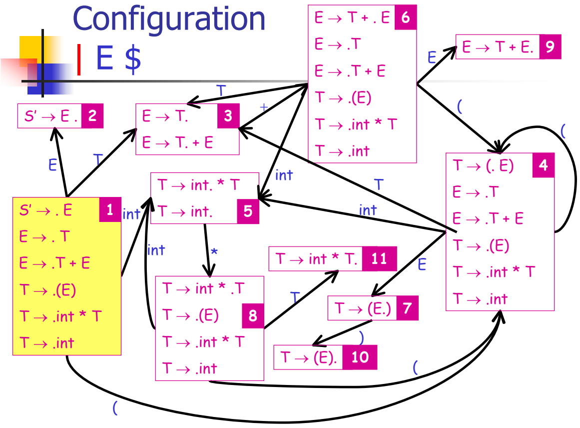
Configuration
T | $



SLR Example

Configuration	*DFA Halt State*	*Action*
| int * int $	1	shift
int | * int $	5 * not in Follow(T)	shift
int * | int $	8	shift
int * int | $	5 $ ∈ Follow(T)	reduce T→int
int * | T $	8	shift
int * T | $	11 $ ∈ Follow(T)	reduce T→int * T
| T $	1	shift
T | $	3 $ ∈ Follow(E)	reduce E→T
| E $	1	

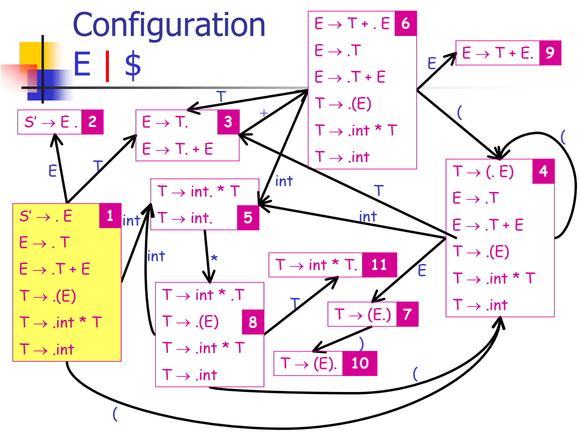
Configuration | E $



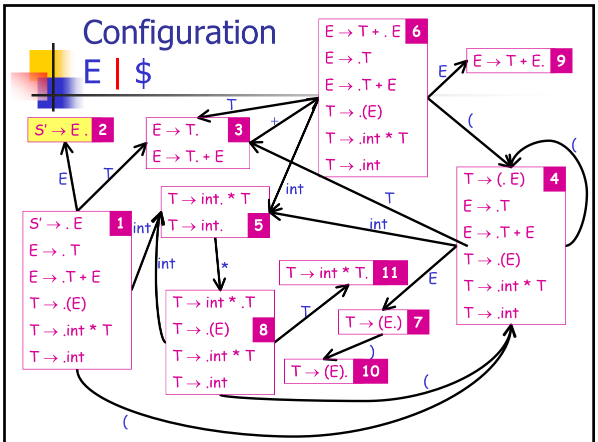
SLR Example

Configuration	*DFA Halt State*	*Action*
| int * int $	1	shift
int | * int $	5 * not in Follow(T)	shift
int * | int $	8	shift
int * int | $	5 $ ∈ Follow(T)	reduce T→int
int * | T $	8	shift
int * T | $	11 $ ∈ Follow(T)	reduce T→int * T
| T $	1	shift
T | $	3 $ ∈ Follow(E)	reduce E→T
| E $	1	shift
E | $	2	ACCEPT

Configuration E | $



Configuration E | $



Notes

- Can also use one more state:
  - it accepts in state "S' → E $. "
  - i.e., it accepts in configuration E$|, not in E|$.
- Rerunning the automaton at each step is wasteful
  - Most of the work is repeated

An Improvement

- Remember the state of the automaton on each prefix of the stack
- Change stack to contain pairs
  ⟨ DFA State , Symbol ⟩

An Improvement (Cont.)

- For a stack
  ⟨ $state_1$, sym_1 ⟩ . . . ⟨ $state_n$, sym_n ⟩
  $state_n$ is the final state of the DFA on sym_1 … sym_n
- Detail: bottom of stack is ⟨start,any⟩ where
  - any is any dummy state
  - start is the start state of the DFA

Goto Table

- Define Goto[i,A] = j if $state_i \to^A state_j$
- Goto is just the transition function of the DFA
  - One of two parsing tables

Refined Parser Moves

- Shift x
  - Push ⟨a, x⟩ on the stack
  - a is current input
  - x is a DFA state
- Reduce $X \to \alpha$
  - As before
- Accept
- Error

Action Table

For each state s_i and terminal a

- If s_i has item $X \to \alpha.a\beta$ and Goto[i,a] = j then Action[i,a] = shift j
- If s_i has item $X \to \alpha.$ and $a \in$ Follow(X) and $X \neq S'$ then Action[i,a] = reduce $X \to \alpha$
- If s_i has item $S' \to S.$ then action[i,$] = accept
- Otherwise, action[i,a] = error

SLR Parsing Algorithm

```
Let Input = w$ be initial input
Let J = 1
Let DFA state 1 have item S' → .S
Let stack = ⟨ 1 , dummy ⟩
  repeat
      case action[top_state(stack),Input_J] of
           shift k:  push ⟨ k, Input_J ⟩, J++
           reduce X → A:
               pop |A| pairs,
               replace Input_{J-|A|} to Input_{J-1} with X
               J = J - |A|
           accept: halt normally
           error: halt and report error
```

Notes on SLR Parsing Algorithm

- Note that the algorithm uses only the DFA states and the input
  - The stack symbols are never used!
- However, we still need the symbols for semantic actions

Constructing SLR states

- LR(0) state machine
  - encodes all strings that are valid on the stack
  - each valid string is a configuration, and hence corresponds to a state of the LR(0) state machine
  - each state tells us what to do (shift or reduce?)

Example SLR Parse Table

	int	*	+	(	)	$	E	T
1	s5			s4			s2	s3
2						acc		
3			s6		r2	r2		
4	s5			s4			s7	s3
5		s8	r4		r4	r4		
6	s5			s4			s9	s3
7					s10			
8	s5			s4				s11
9					r1	r1		
10			r5		r5	r5		
11			r3		r3	r3		

1: E → T + E
2: E → T
3: T → int * T
4: T → int
5: T →(E)

Example SLR Parse

Stack	Input	J	Act
<1,?>	int * int $	1	s5
<5,int><1,?>		2	s8
<8,*><5,int><1,?>		3	s5
<5,int><8,*><5,int><1,?>		4	r4
<8,*><5,int><1,?>	int * T $	3	s11
<11,T> <8,*><5,int><1,?>		4	r3
<1,?>	T $	1	s3
<3,T><1,?>		2	r2
<1,?>	E $	1	s2
<2,E><1,?>		2	acc

Another Example

int * (int + int) * int $