Compilation Principle
编 译 原 理

第9讲：语法分析(6)

张献伟

xianweiz.github.io

DCS290, 03/30/2021

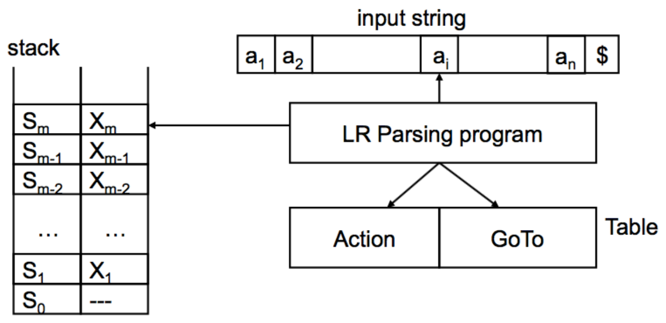
Review Questions (1)

- What does LR(k) mean?

  L: scan the input from left to right

  R: construct a rightmost derivation in reverse

  k: use k input symbols of lookahead



- What are the parts of a LR parser?

  Input buffer, stack, parse table, driver

- What are held in the stack of a LR parser?

  A sequence of states, and each has an associated grammar symbol

- The LR parsing table is split into two, what are they?

  Action table for terminals, Goto table for non-terminals

- What are the possible actions in Action table?

  Shift, reduce, accept, error

Review Questions (2)

- Action table entries can be si and rj, what are i and j?

  si: shift the input symbol and move to state I

  rj: reduce by production numbered j
- Item/Configuration: what does A → XYZ· mean?

  We have seen the body XYZ and it is time to reduce XYZ to A
- State: why we put the items into a configuration set?

  We hope to see one symbol in First(Y)

  Y → u|w

  A → X·YZ
  Y → ·u
  Y → ·w
- What is augmented grammar?

  Add one extra rule S' → S to guarantee only one 'acc' in the table
- What are the possible items of S' → S?

  S' → .S: initial item, haven't seen any input symbol

  S' → S.: accept item, have reduced the input string to start symbol

Example

(0) S' → S	(1) S → BB	(2) B → aB	(3) B → b
	S → ·BB	B → ·aB	
S' → ·S	S → B·B	B → a·B	B → ·b
S' → S·	S → BB·	B → aB·	B → b·

Initial item

Reduce item

Accept item

- **Closure**: the action of adding equivalent items to a set
  - Example: S' → ·S S → ·BB B → ·aB B → ·b
- Intuitively, A → α·Bβ means that we might next see a <u>substring derivable from Bβ</u> (_*sub*) as input. The _*sub* will have a prefix derivable from B by applying one of the B-productions.
  - Thus, we add items for all the B-productions, i.e., if B → γ is a production, we add B → ·γ in the closure

Example

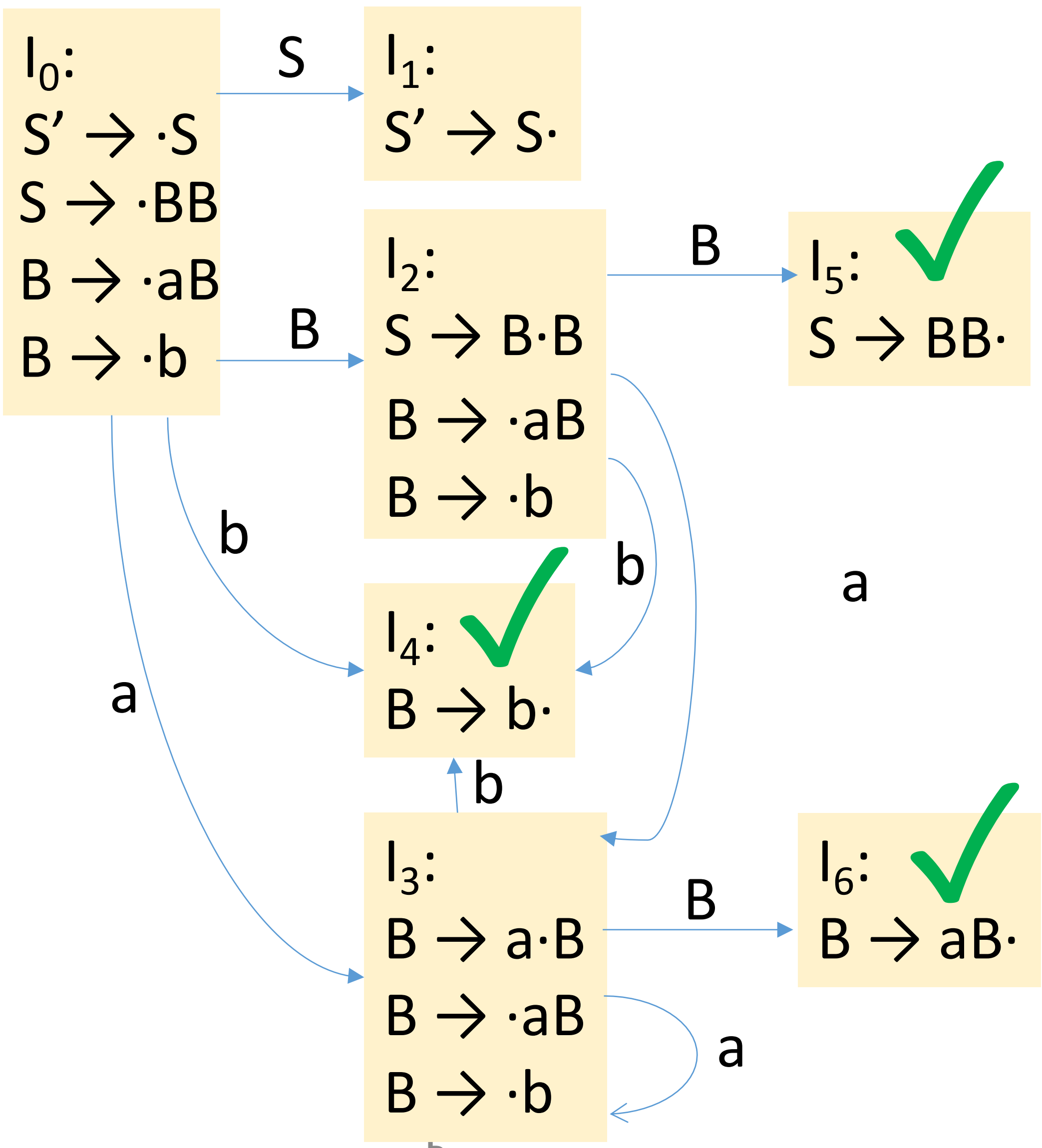
Grammar:

(0) S' → S

(1) S → BB

(2) B → aB

(3) B → b



Example (cont.)

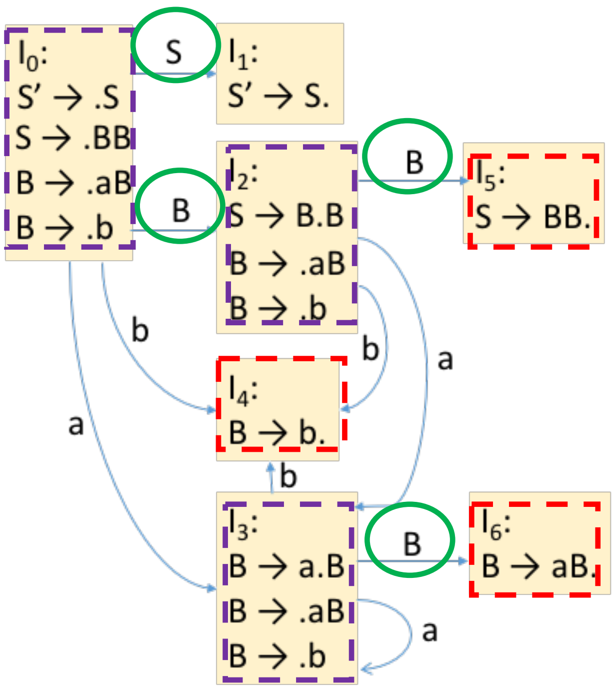
Grammar:

(0) S' → S

(1) S → BB

(2) B → aB

(3) B → b



State	ACTION			GOTO	
	a	b	$	S	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

CLOSURE()[闭包]

- **Closure of item sets**: if *I* is a set of items for a grammar *G*, then *closure(I)* is the set of items constructed from *I* by the two rules:
  - Initially, add every item in *I* to *CLOSURE(I)*
  - If $A \rightarrow \alpha \cdot B\beta$ is in *CLOSURE(I)* and $B \rightarrow \gamma$ is a production, then add item $B \rightarrow \cdot\gamma$ to *CLOSURE(I)*, if it is not already there
    - Apply this rule until no more new items can be added to *CLOSURE(I)*

Grammar:

(0) $S' \rightarrow S$

(1) $S \rightarrow BB$

(2) $B \rightarrow aB$

(3) $B \rightarrow b$

$S' \rightarrow \cdot S$ ⇒

$S' \rightarrow \cdot S$

$S \rightarrow \cdot BB$

$B \rightarrow \cdot aB$

$B \rightarrow \cdot b$

GOTO()[跳转]

- GOTO(I, X): returns state (set of items) that can be reached by advancing X
  - Where I is a set of items and X is a grammar symbol
  - The closure of the set of all items [A → αX·β] such that [A → α·Xβ] is in I
  - Used to define the transitions in the LR(0) automaton
    - The states of the automaton correspond to sets of items, and GOTO(I, X) specifies the transition from the state for I under input X

Grammar:

(0) S' → S

(1) S → BB

(2) B → aB

(3) B → b

I_0:

S' → ·S

S → ·BB

B → ·aB

B → ·b

$\xrightarrow{B}$

I_2:

S → B·B

B → ·aB

B → ·b

Construct LR(0) States

- Create augmented grammar G' for G
  - Given G: $S \rightarrow \alpha \mid \beta$, create G': $S' \rightarrow S \quad S \rightarrow \alpha \mid \beta$
  - Creates a single rule $S' \rightarrow S$ that when reduced, signals acceptance
- Create 1st state by performing a closure on initial item $S' \rightarrow \cdot S$
  - Closure(I): creates state from an initial set of items I
  - $\text{Closure}(\{S' \rightarrow \cdot S\}) = \{S' \rightarrow \cdot S, S \rightarrow \cdot\alpha, S \rightarrow \cdot\beta\}$
- Create additional states by performing a goto on each symbol
  - Goto(I, X): creates state that can be reached from I by advancing X
  - If α was single symbol, the following new state would be created:

    $\text{Goto}(\{S' \rightarrow \cdot S, S \rightarrow \cdot\alpha, S \rightarrow \cdot\beta\}, \alpha) =$

    $\text{Closure}(\{S \rightarrow \alpha\cdot\}) = \{S \rightarrow \alpha\cdot\}$
- Repeatedly perform gotos until there are no more states to add

Construct DFA

- Compute canonical LR(0) collection[规范LR(0)项集族, C], i.e., set of all states in DFA
  - One collection of sets of LR(0) items provides the basis for constructing a DFA that is used to make parsing decisions
  - Such an automaton is called an **LR(0) automaton**
    - Each state of the LR(0) automaton represents a set of items in the C
- All new states are added through goto(I, X)
  - State transitions are done on symbol X

```
void items(G') {
  C = { CLOSURE({[S' → ·S]}) };
  repeat
    for ( each state I in C )
        for ( each grammar symbol X )
          if ( GOTO(I, X) is not empty and not in C)
            add GOTO(I, X) to C;
  until no new states are added to C
}
```

LR(0) Automaton[自动机]

- The LR(0) automaton: each time we perform a shift we are following a transition to a new state
  - States: the sets of items in C
    - Start state: CLOSURE({[S' → ·S]})
    - State *j* refers to the state corresponding to the set of items I_j
  - Transitions are given by the GOTO function

- How can the automaton help with shift-reduce decisions?
  - Suppose that the string γ of grammar symbols takes the LR(0) automaton from the start state *0* to some state *j*
  - Then, shift on next input symbol *a* if state *j* has a transition on *a*
  - Otherwise, we choose to reduce
    - The items in state *j* tell us which production to use

The Example

Grammar:

(0) $S' \to S$

(1) $S \to BB$

(2) $B \to aB$

(3) $B \to b$

- S_0 = Closure({$S' \to .S$})

  = {$S' \to .S$, $S \to .BB$, $B \to .aB$, $B \to .b$}

- Goto(S_0, B) = closure({$S \to B.B$})

  S_2 = {$S \to B.B$, $B \to .aB$, $B \to .b$}

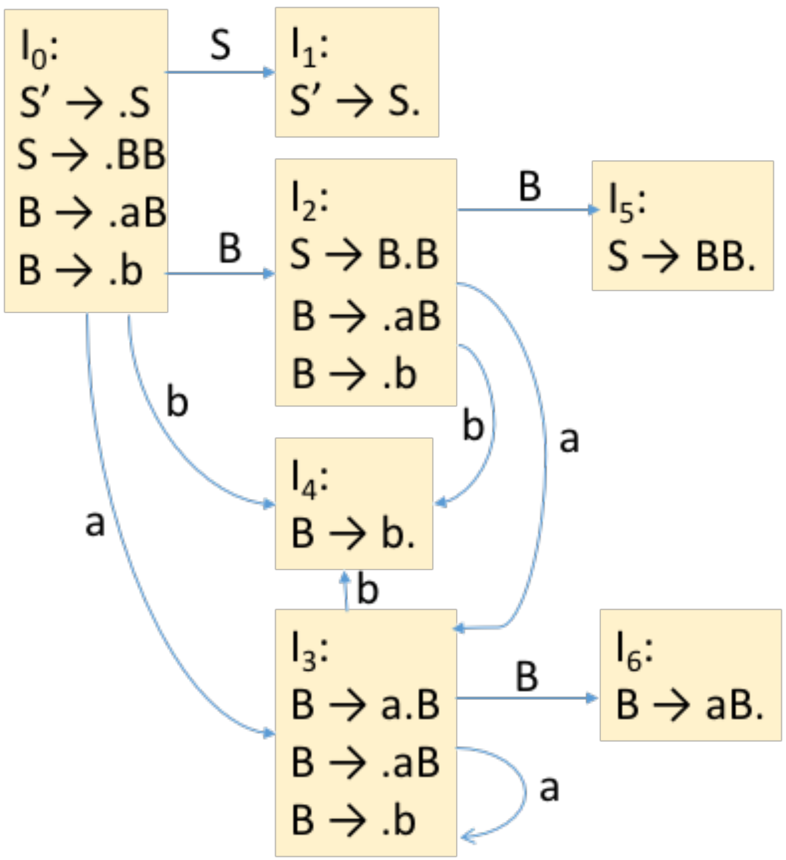
- Goto(S_0, a) = closure({$B \to a.B$})

  S_3 = {$B \to a.B$, $B \to .aB$, $B \to .b$}

- Goto(S_0, b) = closure({$B \to b.$})

  S_4 = {$B \to b.$}

… … …



Build Parse Table from DFA

- ACTION [*state, terminal symbol*]
- GOTO [*state, non-terminal symbol*]
- ACTION:
  - If [$A \rightarrow \alpha \cdot a\beta$] is in S_i and goto(S_i, a) = S_j, where "a" is a terminal
    then ACTION[S_i, a] = shift j (sj)
  - If [$A \rightarrow \alpha \cdot$] is in S_i and $A \rightarrow \alpha$ is rule number j
    then ACTION[S_i, a] = reduce j (rj)
  - If [$S' \rightarrow S_0 \cdot$] is in S_i then ACTION[S_i, \$] = accept
  - If no conflicts among 'shift' and 'reduce' (the first two 'if's)
    then this parser is able to parse the given grammar
- GOTO
  - if goto(S_i, A) = S_j then GOTO[S_i, A] = j
- All entries not filled are rejects

The Example

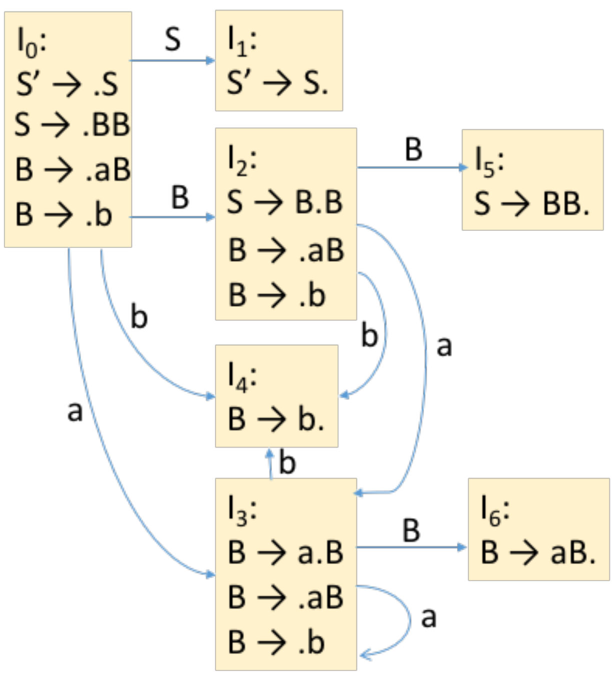
Grammar:

(0) S’ → S

(1) S → BB

(2) B → aB

(3) B → b



State	ACTION			GOTO	
	a	b	$	S	B
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

LR(0) Parsing

- Construct LR(0) automaton from the Grammar

- Idea: assume
  - Input buffer contains α
  - Next input is t
  - DFA on input α terminates in state s
- Reduce by $X \rightarrow \beta$ if
  - s contains item $X \rightarrow \beta\cdot$
- Shift if
  - s contains item $X \rightarrow \beta \cdot t\omega$
  - Equivalent to saying s has a transition labeled t

SUN YAT-SEN UNIVERSITY

LR(0) Parsing (cont.)

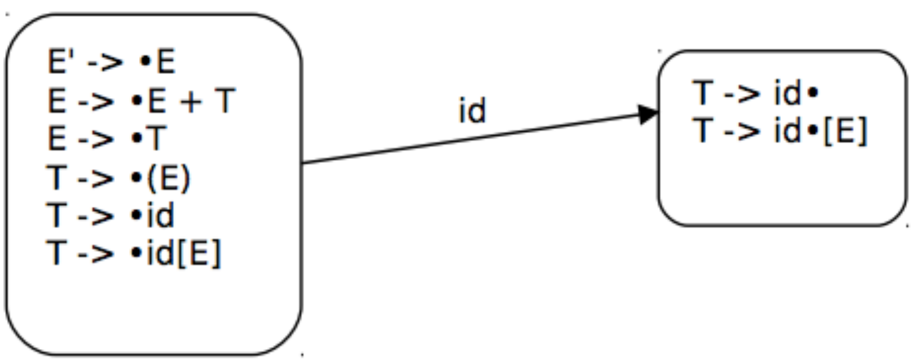
- The parser must be able to determine what action to take in each state without looking at any further input symbols
  - i.e. by only considering what the parsing stack contains so far
  - This is the '0' in the parser name

- In an LR(0) table, each state must only shift or reduce
  - Thus an LR(0) configurating set can only have exactly one reduce item
    - cannot have both shift and reduce items
  - E.g., if the grammar contains the production $A \rightarrow \varepsilon$, then the item $A \rightarrow \cdot\varepsilon$ will create a shift reduce conflict if there is any other nonnull production for A
    - ε-rules are fairly common programming language grammars

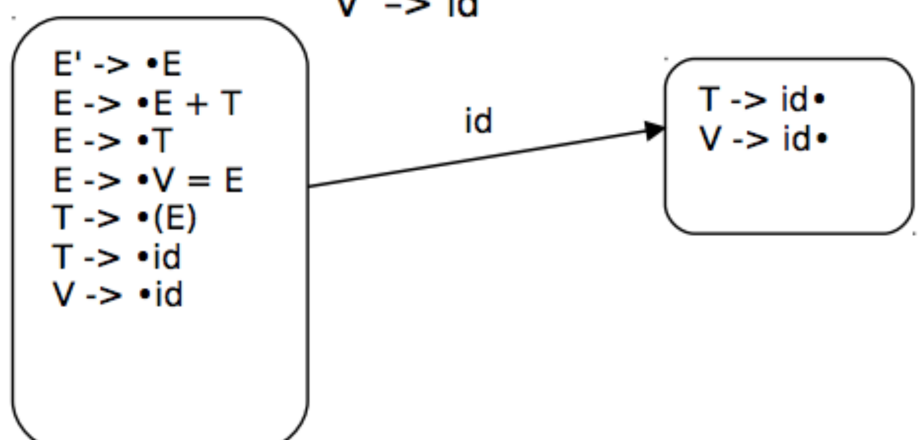
LR(0) Conflicts

- LR(0) has a reduce/reduce conflict if:
  - Any state has two reduce items:
  - $X \rightarrow \beta\cdot$ and $Y \rightarrow \omega\cdot$
- LR(0) has a shift/reduce conflict if:
  - Any state has a reduce item and a shift item:
  - $X \rightarrow \beta\cdot$ and $Y \rightarrow \omega\cdot t\sigma$

E' -> E
E -> E + T | T
T -> (E) | id | id[E]



E' -> E
E -> E + T | T | V = E
T -> (E) | id
V -> id



LR(0) Summary

- LR(0) is the simplest LR parsing
  - Table-driven shift-reduce parser
    - Action table[s, a] + Goto table[s, X]
  - Weakest, not used much in practice
  - Parses without using any lookahead

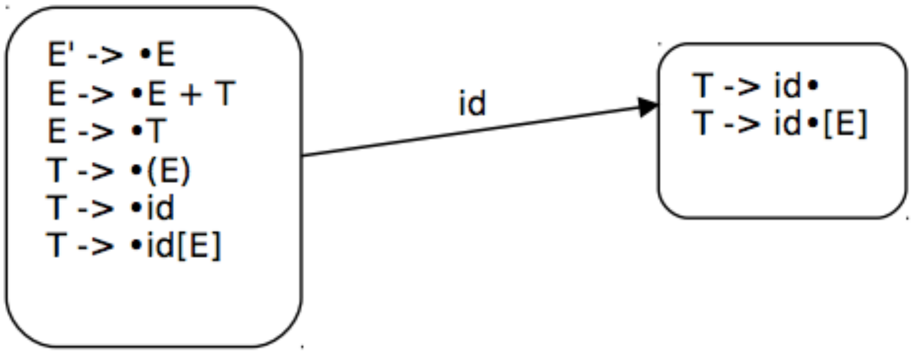
- Adding just one token of lookahead vastly increases the parsing power
  - LR(1)
  - SLR(1)
  - LALR(1)

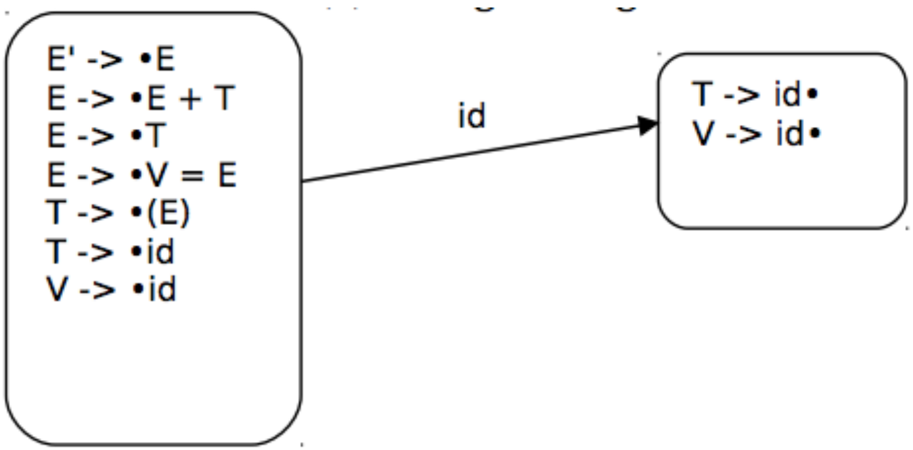
SLR(1) Parsing

- LR(0) conflicts are generally caused by **reduce** actions
  - If the item is complete, the parser must choose to reduce
    - Is this always appropriate?
    - The next upcoming token may tells us something different
  - What tokens may tell the reduction is not appropriate?
    - Perhaps **Follow(A)** could be useful here
- **SLR** = Simple LR
  - Use the same LR(0) configurating sets and have the same table structure and parser operation
  - The difference comes in assigning table actions
    - Use one token of lookahead to help arbitrate among the conflicts
    - Reduce only if the next input token is a member of the follow set of the nonterminal being reduced

SLR(1) Parsing (cont.)

- In the SLR(1) parser, it is allowable for there to be both shift and reduce items in the same state as well as multiple reduce items
  - The SLR(1) parser will be able to determine which action to take as long as the follow sets are disjoint.

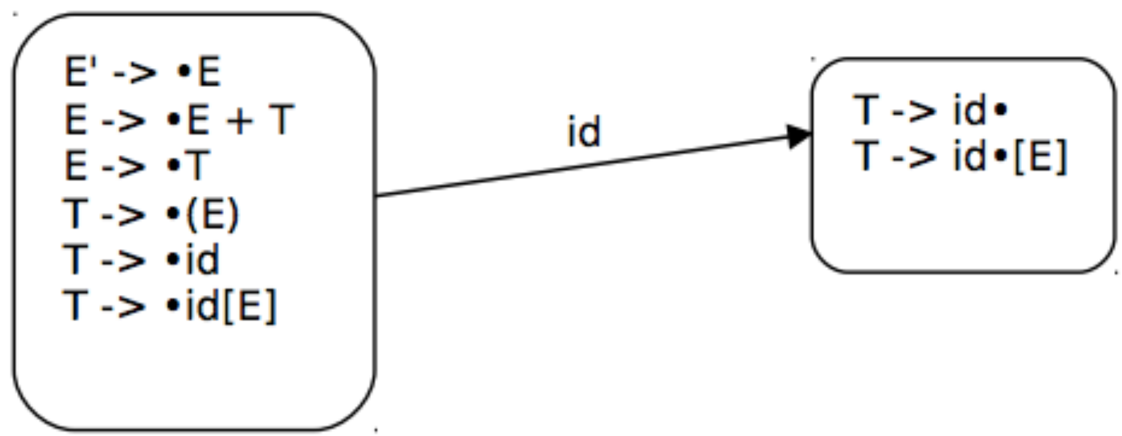




Example

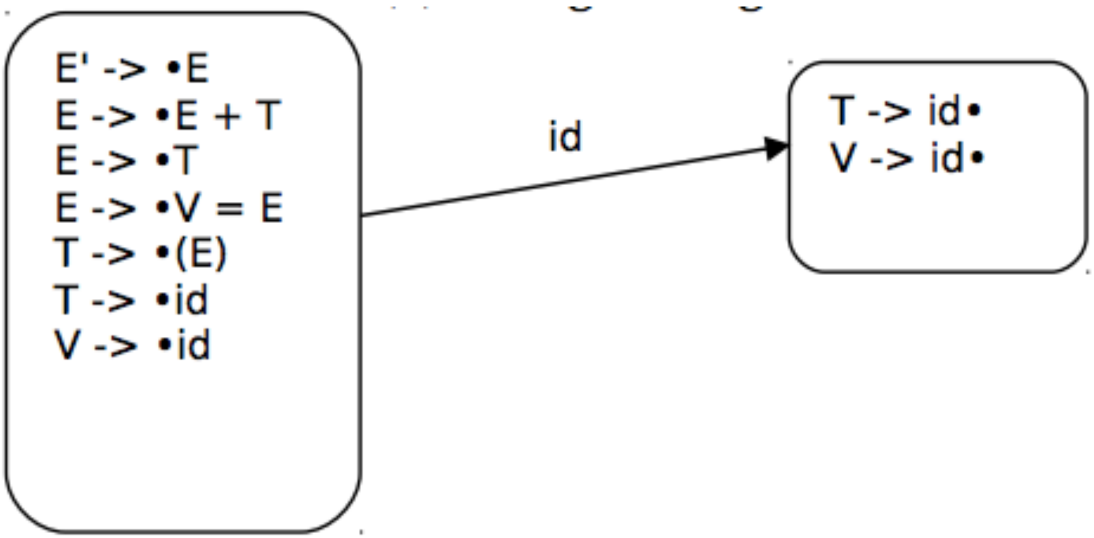
- First two LR(0) configurating sets entered if *id* is the first token of the input
  - LR(0) parser: the set on the right side has a shift-reduce conflict
  - SLR(1) parser:
    - Compute Follow(T) = { +,),], $ }, i.e., only reduce on those tokens
      - Follow(T) = Follow(E) = {+,),], $}
    - The input [will shift and there is no conflict

E' -> E
E -> E + T | T
T -> (E) | id | id[E]



Example (cont.)

- The first two LR(0) configurating sets entered if *id* is the first token of the input
  - LR(0) parser: the right set has a reduce-reduce conflict
  - SLR(1) parser:
    - Capable to distinguish which reduction to apply depending on the next input token
    - Compute Follow(T) = { +,), $ } and Follow(V) = { = }



E' -> E
E -> E + T | T | V = E
T -> (E) | id
V -> id

SLR(1) Grammars

- A grammar is SLR(1) if the following two conditions hold for each configurating set
- (1) For any item $A \rightarrow u \cdot xv$ in the set, with terminal x, there is no complete item $B \rightarrow w \cdot$ in that set with x in Follow(B)
  - In the tables, this translates no shift-reduce conflict on any state
- (2) For any two complete items $A \rightarrow u \cdot$ and $B \rightarrow v \cdot$ in the set, the follow sets must be disjoint, e.g. Follow(A) ∩ Follow(B) is empty
  - This translates to no reduce-reduce conflict on any state
  - If more than one nonterminal could be reduced from this set, it must be possible to uniquely determine which using only one token of lookahead

SLR(1) Limitations

- SLR(1) vs. LR(0)
  - Adding just one token of lookahead and using the Follow set greatly expands the class of grammars that can be parsed without conflict
- When we have a completed configuration (i.e., dot at the end) such as X –> u·, we know that it is reducible
  - We allow such a reduction whenever the next symbol is in Follow(X).
  - However, it may be that we should not reduce for every symbol in Follow(X), because the symbols below u on the stack preclude u being a handle for reduction in this case
  - In other words, SLR(1) states only tell us about the sequence on top of the stack, not what is below it on the stack
  - We may need to divide an SLR(1) state into separate states to differentiate the possible means by which that sequence has appeared on the stack

References

- Bottom-up Parsing, https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/100%20Bottom-Up%20Parsing.pdf
- SLR and LR(1) Parsing, https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/110%20LR%20and%20SLR%20Parsing.pdf
- MOOC-编译原理, https://www.icourse163.org/course/HIT-1002123007