

SLR(1) Parsing: What can go wrong?

Example: A simple grammar

- | | |
|------------------|----------------------|
| 1. $S' ::= S$ | 4. $L ::= * R$ |
| 2. $S ::= L = R$ | 5. $L ::= \text{id}$ |
| 3. $S ::= R$ | 6. $R ::= L$ |

Canonical LR(0) collection

- $I_0 : \{ [S' ::= \bullet S], [S ::= \bullet L = R], [S ::= \bullet R], [L ::= \bullet * R], [L ::= \bullet \text{id}], [R ::= \bullet L] \}$
 $I_1 : \{ [S' ::= S \bullet] \}$
 $I_2 : \{ [S ::= L \bullet = R], [R ::= L \bullet] \}$
 $I_3 : \{ [S ::= R \bullet] \}$
 $I_4 : \{ [L ::= * \bullet R], [R ::= \bullet L], [L ::= \bullet * R], [L ::= \bullet \text{id}] \}$
 $I_5 : \{ [L ::= \text{id} \bullet] \}$
 $I_6 : \{ [S ::= L = \bullet R], [R ::= \bullet L], [L ::= \bullet * R], [L ::= \bullet \text{id}] \}$
 $I_7 : \{ [L ::= * R \bullet] \}$
 $I_8 : \{ [R ::= L \bullet] \}$
 $I_9 : \{ [S ::= L = R \bullet] \}$

SLR(1) table construction

Consider the set of items I_2 . The action table is defined as follows:

- $[S ::= L \bullet = R]$ implies ACTION[2, =] = "shift 6"
- $[R ::= L \bullet]$ implies ACTION[2, =] = "reduce 6"

Due to multiple definitions of the position in the action table, the grammar is not SLR(1).

What can go wrong?

Two cases arise

shift/reduce

This is called a shift/reduce conflict. In general, it indicates an ambiguous construct in the grammar.

- May be able to modify the grammar to eliminate it
- May be able to resolve in favor of shifting

classic example: dangling else

reduce/reduce

This is called a reduce/reduce conflict. Again, it indicates an ambiguous construct in the grammar.

- often, no simple resolution
- parse a nearby language

classic example: PL/I call and subscript

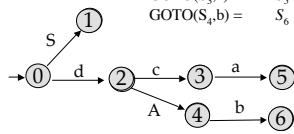
Some grammars are not SLR(1)

- SLR(1) parsers cannot parse some LR grammars.
- Problem is that lookahead information is added to LR(0) parser at the end of construction based on FOLLOW sets

Example

1. $S' ::= S$
2. $S ::= dca \mid dAb$
3. $A ::= c$

$START = S_0: \{[S' ::= \bullet S], [S ::= \bullet dca], [S ::= \bullet dAb]\}$
 $GOTO(S_0, S) = S_1: \{[S' ::= S \bullet]\}$
 $GOTO(S_0, d) = S_2: \{[S ::= d \bullet ca], [S ::= d \bullet Ab], [A ::= \bullet c]\}$
 $GOTO(S_2, c) = S_3: \{[S ::= dc \bullet a], [A ::= c \bullet]\}$
 $GOTO(S_2, A) = S_4: \{[S ::= dA \bullet b]\}$
 $GOTO(S_3, a) = S_5: \{[S ::= dca \bullet]\}$
 $GOTO(S_4, b) = S_6: \{[S ::= dAb \bullet]\}$



Added by closure

SLR(1) parse table

	d	c	A	\$
S_0	S_2	S_3	S_4	S_5
S_1				
S_2		S_3	S_4	
S_3				
S_4				
S_5				
S_6				

Added because S_3 contains $[A ::= c \bullet]$ and b is in FOLLOW(A)

This grammar can be parsed with an SLR(1) parser

Example : A non-SLR(1) grammar

0. $S' ::= S$
1. $S ::= dca \mid dAb \mid Aa$
2. $A ::= c$

LR(0) items

$START = S_0: \{[S' ::= \bullet S], [S ::= \bullet dca], [S ::= \bullet dAb], [S ::= \bullet Aa], [A ::= \bullet c]\}$
 $GOTO(S_0, S) = S_1: \{[S' ::= S \bullet]\}$
 $GOTO(S_0, d) = S_2: \{[S ::= d \bullet ca], [S ::= d \bullet Ab], [A ::= \bullet c]\}$
 $GOTO(S_2, c) = S_3: \{[S ::= dc \bullet a], [A ::= c \bullet]\}$
 $GOTO(S_2, A) = S_4: \{[S ::= dA \bullet b]\}$
 $GOTO(S_3, a) = S_5: \{[S ::= dca \bullet]\}$
 $GOTO(S_4, b) = S_6: \{[S ::= dAb \bullet]\}$
 $GOTO(S_0, A) = S_7: \{[S ::= A \bullet a]\}$
 $GOTO(S_7, a) = S_8: \{[S ::= Aa \bullet]\}$
 $GOTO(S_0, c) = S_9: \{[A ::= c \bullet]\}$

New production adds "a" to FOLLOW(A)

SLR(1) parse table

	d	c	A	\$
S_0	S_2	S_3	S_4	S_5
S_1				
S_2		S_3	S_4	
S_3				
S_4				
S_5				
S_6				
S_7				
S_8				
S_9				

Shift-reduce conflict!

This grammar cannot be parsed with an SLR(1) parser

LR(1)

We can get more powerful parser by keeping track of lookahead information **in the states of the parser**.

If, in a single left-to-right scan, we can construct a reverse rightmost derivation, while using at most a single token lookahead to resolve ambiguities, then the grammar is LR(1)

LR(k) items

The table construction algorithms use LR(k) items to represent the set of possible states in a parse

An LR(k) item is a pair $[\alpha, \beta]$, where

α is a production from G with a \bullet at some position in the rhs
 β is a lookahead string containing k symbols (terminals or eof)

What about LR(1) items?

- example LR(1) item: $[A ::= X \bullet Y Z, a]$
- LR(1) items have lookahead strings of length 1
- several LR(1) items may have the same **core**

$[A ::= X \bullet Y Z, a]$

$[A ::= X \bullet Y Z, b]$

we represent this as

$[A ::= X \bullet Y Z, \{a, b\}]$

LR(1) lookahead

What's the point of all these lookahead symbols?

- carry them along to allow us to choose correct reduction when there is any choice
- lookaheads are bookkeeping unless item has \bullet at right end.
 - ✓ in $[A ::= X \bullet Y Z, a]$, a has no direct use
 - ✓ in $[A ::= XY Z \bullet, a]$, a is useful

Recall, the SLR(1) construction uses LR(0) items!

The point

For $[A ::= \alpha \bullet, a]$ and $[B ::= \alpha \bullet, b]$, we can decide between reducing to A or B by looking at limited right context!

Canonical LR(1) items

The canonical collection of sets of LR(1) items:

- sets of valid items for viable prefixes of the grammar
- sets of items derivable from $[S' ::= \bullet S, \text{eof}]$ using **goto** and **closure** functions -- both functions preserve validity.

A LR(1) item $[A ::= \alpha \bullet \beta, a]$ is valid for a viable prefix γ if there is a derivation $S \Rightarrow_m^* \delta \alpha w \Rightarrow_m \delta \alpha \beta w$, where

- $\gamma = \delta \alpha$, and
- either a is the first symbol of w , or w is ϵ and a is **eof**.

Essentially,

- Each LR(1) item in a set in the canonical collection represents a state in an NFA that recognizes viable prefixes.
- Grouping these items together is really the DFA subset construction.

LR(1) closure

Given an item $[A ::= \alpha \bullet B\beta, a]$, its closure contains the item and any other items that can generate legal substrings to follow α .

Thus, if the parser has viable prefix α on its stack, a substring of the input should reduce to $B\beta$ (or for some other item $[B ::= \bullet \gamma, b]$ in the closure).

To compute $\text{closure}(I)$:

```
function closure(I)
  repeat
    new_item ← false
    for each item  $[A ::= \alpha \bullet B\beta, a] \in I$ ,
      each production  $B ::= \gamma \in G$ ,
      and each terminal  $b \in \text{FIRST}(\beta a)$ ,
      if  $[B ::= \bullet \gamma, b] \notin I$  then
        add  $[B ::= \bullet \gamma, b]$  to  $I$ 
        new_item ← true
    endif
  until (new_item = false)
  return I
```

LR(1) goto

Let I be a set of LR(1) items and X be a grammar symbol.

Then, $\text{goto}(I, X)$ is the closure of the set of all items $[A ::= \alpha X \bullet \beta, a]$ such that $[A ::= \alpha \bullet X\beta, a] \in I$

If I is the set of valid items for some viable prefix γ , then $\text{goto}(I, X)$ is the set of valid items for the viable prefix γX .

$\text{goto}(I, X)$ represents state after recognizing X in state I .

To compute $\text{goto}(I, X)$:

```
function goto(I, X)
  J ← set of items  $[A ::= \alpha X \bullet \beta, a]$ 
    such that  $[A ::= \alpha \bullet X\beta, a] \in I$ 
  J' ← closure(J)
  return J'
```

Collection of sets of LR(1) items

We start the construction of the canonical collection of LR(1) items with the item $[S' ::= \bullet S, \text{eof}]$, where

S' is the start symbol of the augmented grammar G'
 S is the start symbol of G , and
eof is the right end of string marker

To compute the collection of sets of LR(1) items

```
procedure items(G')
  C ← {closure({[S' ::= • S, eof])}}
  repeat
    new_item ← false
    for each set of items  $I$  in  $C$  and each grammar symbol  $X$ 
      such that  $\text{goto}(I, X) \neq \emptyset$  and  $\text{goto}(I, X) \notin C$ 
      add  $\text{goto}(I, X)$  to  $C$ 
      new_item ← true
    endfor
  until (new_item = false)
```

LR(1) table construction

The Algorithm

- construct the collection of sets of LR(1) items for G' .
- State i of the parser is constructed from I_i .
 - if $[A ::= \alpha \bullet a\beta, b] \in I_i$ and $\text{goto}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to **shift j**. (a must be a terminal)
 - new** if $[A ::= \alpha \bullet a] \in I_i$, then set $\text{ACTION}[i, a]$ to **reduce** $A ::= \alpha \bullet a$.
 - if $[S' ::= S \bullet, \text{eof}] \in I_i$, then set $\text{ACTION}[i, \text{eof}]$ to **accept**.
- If $\text{goto}(I_i, A) = I_j$, then set $\text{GOTO}[i, A]$ to j .
- All other entries in ACTION and GOTO are set to **error**.
- The initial state of the parser is the state constructed from the set containing the item $[S' ::= \bullet S, \text{eof}]$.**

Example

LR(1) items

START = S_0 : { $[S' ::= \bullet S, \text{eof}]$, $[S ::= \bullet dca, \text{eof}]$, $[S ::= \bullet dAb, \text{eof}]$, $[S ::= \bullet Aa, \text{eof}]$, $[A ::= \bullet c, a]$ }

GOTO(S_0, S) = S_1 : { $[S' ::= S \bullet, \text{eof}]$ }

GOTO(S_0, d) = S_2 : { $[S ::= d \bullet ca, \text{eof}]$, $[S ::= d \bullet Ab, \text{eof}]$, $[A ::= \bullet c, b]$ }

GOTO(S_2, c) = S_3 : { $[S ::= dc \bullet a, \text{eof}]$, $[A ::= c \bullet, b]$ }

GOTO(S_2, A) = S_4 : { $[S ::= dA \bullet b, \text{eof}]$ }

GOTO(S_3, a) = S_5 : { $[S ::= dca \bullet, \text{eof}]$ }

GOTO(S_4, b) = S_6 : { $[S ::= dAb \bullet, \text{eof}]$ }

GOTO(S_0, A) = S_7 : { $[S ::= A \bullet a, \text{eof}]$ }

GOTO(S_7, a) = S_8 : { $[S ::= Aa \bullet, \text{eof}]$ }

GOTO(S_0, c) = S_9 : { $[A ::= c \bullet, a]$ }

$[S ::= dc \bullet a, \text{eof}]$ indicates ACTION[2,a] = shift a
 $[A ::= c \bullet, b]$ indicates ACTION[2,b] = reduce

No conflict!
 This grammar is LR(1)

0. $S' ::= S$
1. $S ::= dca \mid dAb \mid Aa$
2. $A ::= c$

Example

How about this one?

- | | |
|------------------|----------------------|
| 1. $S' ::= S$ | 4. $L ::= *R$ |
| 2. $S ::= L = R$ | 5. $L ::= \text{id}$ |
| 3. $S ::= R$ | 6. $R ::= L$ |

Canonical LR(1) collection

I_0 : { $[S' ::= \bullet S, \text{eof}]$, $[S ::= \bullet L = R, \text{eof}]$, $[S ::= \bullet R, \text{eof}]$, $[L ::= \bullet *R, \{=, \text{eof}\}]$, $[L ::= \bullet \text{id}, \{=, \text{eof}\}]$, $[R ::= \bullet L, \text{eof}]$ }

I_1 : { $[S0 ::= S \bullet, \text{eof}]$ }

I_2 : { $[S ::= L \bullet = R, \text{eof}]$, $[R ::= L \bullet, \text{eof}]$ }

I_3 : { $[S ::= R \bullet, \text{eof}]$ }

I_4 : { $[L ::= \bullet *R, \{=, \text{eof}\}]$, $[R ::= \bullet L, \{=, \text{eof}\}]$, $[L ::= \bullet *R, \{=, \text{eof}\}]$, $[L ::= \bullet \text{id}, \{=, \text{eof}\}]$ }

I_5 : { $[L ::= \text{id} \bullet, \{=, \text{eof}\}]$ }

I_6 : { $[S ::= L = \bullet R, \text{eof}]$, $[R ::= \bullet L, \text{eof}]$, $[L ::= \bullet *R, \text{eof}]$, $[L ::= \bullet \text{id}, \text{eof}]$ }

I_7 : { $[L ::= *R \bullet, \{=, \text{eof}\}]$ }

I_8 : { $[R ::= L \bullet, \{=, \text{eof}\}]$ }

I_9 : { $[S ::= L = R \bullet, \text{eof}]$ }

I_{10} : { $[R ::= L \bullet, \text{eof}]$ }

I_{11} : { $[L ::= *R, \text{eof}]$, $[R ::= \bullet L, \text{eof}]$, $[L ::= \bullet *R, \text{eof}]$, $[L ::= \bullet \text{id}, \text{eof}]$ }

I_{12} : { $[L ::= \text{id} \bullet, \text{eof}]$ }

I_{13} : { $[L ::= *R \bullet, \text{eof}]$ }

FOLLOW(S') = {eof}
 FOLLOW(S) = {eof}
 FOLLOW(L) = {=, eof}
 FOLLOW(R) = {=, eof}

$[S ::= L \bullet = R]$ indicates ACTION[2, =] = "shift"
 $[R ::= L \bullet]$ indicates ACTION[2, eof] = "reduce"

No conflict! This grammar is LR(1)

An LR Parsing Engine

A deterministic finite automaton applied to the stack and taken the lookahead as input is used to guide the parsing actions.

Consider the following grammar rules:

1 $S \rightarrow S ; S$	4 $E \rightarrow \text{id}$	8 $L \rightarrow E$
2 $S \rightarrow \text{id} := E$	5 $E \rightarrow \text{num}$	9 $L \rightarrow L , E$
3 $S \rightarrow \text{print} (L)$	6 $E \rightarrow E + E$	
	7 $E \rightarrow (S , E)$	

What are the shift-reduce parse actions for the program:

a := 7;
b := c + (d := 5 + 6, d)

