

LR parsing

There are three commonly used algorithms to build tables for an “LR” parser:

1. $SLR(1)$ = $LR(0)$ + FOLLOW

- smallest class of grammars
- smallest tables (number of states)
- simple, fast construction

2. $LR(1)$

- full set of $LR(1)$ grammars
- largest tables (number of states)
- slow, large construction

3. $LALR(1)$

- intermediate sized set of grammars
- same number of states as $SLR(1)$
- canonical construction is slow and large
- better construction techniques exist

An $LR(1)$ parser for either ALGOL or PASCAL has several thousand states, while an $SLR(1)$ or $LALR(1)$ parser for the same language may have several hundred states

Viab!e prefix

A *viab!e prefix* is

1. a prefix of a right-sentential form that does not continue past the right end of the rightmost handle of that sentential form[†], or
2. a prefix of a right-sentential form that can appear on the stack of a shift-reduce parser.

If the viable prefix is a proper prefix (that is, a handle), it is possible to add terminals onto its end to form a right-sentential form.

As long as the prefix represented by the stack is viable, the parser has not seen a detectable error.

[†] If the grammar is unambiguous, there is a unique rightmost handle. $LR(k)$ grammars are unambiguous. Operator grammars may be ambiguous, but are still parsed with *shift-reduce* parsers.

SLR(1) parsing

Viable prefix of a right-sentential form:

- contains both terminals and nonterminals
- recognized with NFA or DFA

Building a *SLR* parser

- begin with NFA for recognizing viable prefixes
- construct DFA for recognizing viable prefixes
- augment with FOLLOW to disambiguate reductions

States in the NFA are *LR(0)* items

States in the DFA are sets of *LR(0)* items

LR(0) items

An $LR(0)$ item is a string $[\alpha]$, where

α is a production from G with a \bullet at some position in the *rhs*

The \bullet indicates how much of an item we have seen at a given state in the parse.

$[A ::= \bullet XYZ]$ indicates that the parser is looking for a string that can be derived from XYZ

$[A ::= XY \bullet Z]$ indicates that the parser has seen a string derived from XY and is looking for one derivable from Z

$LR(0)$ items (*no lookahead*)

$A ::= XYZ$ generates 4 $LR(0)$ items.

1. $[A ::= \bullet XYZ]$
2. $[A ::= X \bullet YZ]$
3. $[A ::= XY \bullet Z]$
4. $[A ::= XYZ \bullet]$

Canonical $LR(0)$ items

The $SLR(1)$ table construction algorithm uses a specific set of sets of $LR(0)$ items

These sets are called the *canonical collection of sets of $LR(0)$ items* for a grammar G

The canonical collection represents the set of valid states for the LR parser

The items in each set of the canonical collection fall into two classes:

kernel items: items where \bullet is not at the left end of the *rhs* and $[S' ::= \bullet S]$

non-kernel items: all items where \bullet is at the left end of *rhs*

LR(0) items

Each $LR(0)$ item corresponds to a point in the parse

To generate a *parser state* from a kernel item, we take its closure

\Rightarrow if $[A ::= \alpha \bullet B\beta] \in I_j$, then, in state j , the parser might next see a string derivable from $B\beta$

\Rightarrow to form its closure, add all items of the form $[B ::= \bullet\gamma] \in G$

† An “augmented grammar” is one where the start symbol appears only on the *lhs* of productions

For the rest of LR parsing, we will assume the grammar is augmented with a production $S' ::= S$

Canonical $LR(0)$ items

The canonical collection of $LR(0)$ items:

- set of items derivable from $[S' ::= \bullet S]$
- set of all items that can derive the final configuration

Essentially,

- each set in the canonical collection of sets of $LR(0)$ items represents a state in an NFA that recognizes viable prefixes.
- Grouping together is really the subset construction, §3.6

To construct the canonical collection we need two functions:

- $\text{closure}(I)$
- $\text{GOTO}(I, X)$

Closure(I)

Given an item $[A ::= \alpha \bullet B\beta]$, 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, the input should reduce to $B\beta$ (or γ for some other item $[B ::= \bullet\gamma]$ in the closure).

To compute closure(I)

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

Goto(I, X)

Let I be a set of $LR(0)$ items and X be a grammar symbol.

Then, $\text{GOTO}(I, X)$ is the closure of the set of all items

$$[A ::= \alpha X \bullet \beta] \text{ such that } [A ::= \alpha \bullet X \beta] \in I$$

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 ::= αX • β]
    such that [A ::= α • Xβ] ∈ I
  J' ← closure(J)
  return J'
```

Collection of sets of LR(0) items

We start the construction of the collection of sets of LR(0) items with the item $[S' ::= \bullet S]$, where

S' is the start symbol of the augmented grammar G'

S is the start symbol of G

To compute the collection of sets of LR(0) items

```
procedure items( $G'$ )
   $S_0 \leftarrow \text{closure}(\{[S' ::= \bullet S]\})$ 
  Items  $\leftarrow \{ S_0 \}$ 
  ToDo  $\leftarrow \{ S_0 \}$ 
  while ToDo not empty do
    remove  $S_i$  from ToDo
    for each grammar symbol  $X$  do
       $S_{new} \leftarrow \text{goto}(S_i, X)$ 
      if  $S_{new}$  is a new state then
        Items  $\leftarrow \text{Items} \cup \{ S_{new} \}$ 
        ToDo  $\leftarrow \text{Items} \cup \{ S_{new} \}$ 
      endif
    endfor
  endwhile
  return Items
```

LR(0) machines

LR(0) DFA

- states – canonical sets of *LR(0)* items
- edges – goto transitions
- recognizes all viable prefixes of handles
- no lookahead

To be able to recognize viable prefixes of the language (instead of the handles), we must be able to reduce handles to nonterminals

Reducing a handle (rhs of production) to a nonterminal can be viewed as:

- returning to state at beginning of handle
- making transition on nonterminal

To return to state at beginning of the handle, we must use the stack!

SLR(1) tables

SLR(1) parser

- augment *LR(0)* machine
- add FOLLOW information using one token of lookahead
- encoded as ACTION, GOTO tables

ACTION table

- for each [state, lookahead] pair
- have we reached end of handle?
- if not, shift
- if at end of handle, reduce
- may also accept or error
- use lookahead to guide decision

GOTO table

- for each [state, nonterminal] pair
- pick state to go to after reduction
- look at nonterminal at top of stack

SLR(1) table construction

The Algorithm

1. construct the collection of sets of $LR(0)$ items for G' .
2. State i of the parser is constructed from I_i .
 - (a) if $[A ::= \alpha \bullet a\beta] \in I_i$ and $\text{goto}(I_i, a) = I_j$, then set $\text{ACTION}[i, a]$ to “*shift j*”. (a must be a terminal)
 - (b) if $[A ::= \alpha \bullet] \in I_i$, then set $\text{ACTION}[i, a]$ to “*reduce A ::= α*” for all a in $\text{FOLLOW}(A)$.
 - (c) if $[S' ::= S \bullet] \in I_i$, then set $\text{ACTION}[i, \text{eof}]$ to “*accept*”.
3. If $\text{goto}(I_i, A) = I_j$, then set $\text{GOTO}[i, A]$ to j .
4. All other entries in ACTION and GOTO are set to “*error*”
5. The initial state of the parser is the state constructed from the set containing the item $[S' ::= \bullet S]$.

SLR(1) parser example

The Grammar

1		E	::=	T + E
2				T
3		T	::=	id

The Augmented Grammar

0		S'	::=	E
1		E	::=	T + E
2				T
3		T	::=	id

Symbol	FIRST	FOLLOW
S'	{ id }	{ eof }
E	{ id }	{ eof }
T	{ id }	{ +, eof }

Example LR(0) states

S_0 : [$S' ::= \bullet E$],
[$E ::= \bullet T + E$],
[$E ::= \bullet T$],
[$T ::= \bullet id$]

S_1 : [$S' ::= E \bullet$]

S_2 : [$E ::= T \bullet + E$],
[$E ::= T \bullet$]

S_3 : [$T ::= id \bullet$]

S_4 : [$E ::= T + \bullet E$],
[$E ::= \bullet T + E$],
[$E ::= \bullet T$],
[$T ::= \bullet id$]

S_5 : [$E ::= T + E \bullet$]

Example GOTO function

Start

$$S_0 \leftarrow \text{closure} (\{ [S ::= \bullet E] \})$$

Iteration 1

$$\text{goto}(S_0, E) = S_1$$

$$\text{goto}(S_0, T) = S_2$$

$$\text{goto}(S_0, \text{id}) = S_3$$

Iteration 2

$$\text{goto}(S_2, +) = S_4$$

Iteration 3

$$\text{goto}(S_4, \text{id}) = S_3$$

$$\text{goto}(S_4, E) = S_5$$

$$\text{goto}(S_4, T) = S_2$$

Example ACTION and GOTO tables

	ACTION			GOTO	
	id	+	\$	expr	term
S_0	shift 3	—	—	1	2
S_1	—	—	accept	—	—
S_2	—	shift 4	reduce 2	—	—
S_3	—	reduce 3	reduce 3	—	—
S_4	shift 3	—	—	5	2
S_5	—	—	reduce 1	—	—

Stack	Input	Action
\$ 0	id + id \$	shift 3
\$ 0 id 3	+ id \$	reduce 3 (T ::= id)
\$ 0 T 2	+ id \$	shift 4
\$ 0 T 2 + 4	id \$	shift 3
\$ 0 T 2 + 4 id 3	\$	reduce 3 (T ::= id)
\$ 0 T 2 + 4 T 2	\$	reduce 2 (E ::= T)
\$ 0 T 2 + 4 E 5	\$	reduce 1 (E ::= T + E)
\$ 0 E 1	\$	accept

What can go wrong?

Rules 2a, 2b, & 2c in the $SLR(1)$ table construction algorithm can multiply define a position in ACTION
If this happens, the grammar is not $SLR(1)$.

Two cases arise

shift/reduce

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

- can modify the grammar to eliminate it
- can 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