LR(1) parsers

LR(1) table construction algorithm

1. build I, the *canonical collection of sets of* LR*(1) items*
   (a) $I_0 \leftarrow \texttt{closure}(\{[S' \rightarrow \bullet S, \texttt{eof}]\})$
   (b) repeat until no sets are added
      for $I_j \in I$ and $X \in NT \cup T$
         if $goto(I_j, X)$ is a new set, add it to I
2. iterate through $I_j \in I$, filling in the ACTION table
3. fill in the GOTO table

What does I_j "mean"?

- $[A \rightarrow X \bullet YZ, \alpha] \Rightarrow$ have recognized X & YZ would be valid
- $[A \rightarrow X \bullet YZ, \alpha] \Rightarrow [Y \rightarrow \bullet\beta, \gamma]$ & $[Y \rightarrow \bullet\delta, \eta]$ are also valid, where $\gamma, \eta \in \mathrm{FIRST}(\mathrm{Z}\alpha)$
- recognizing Y takes parser to $[A \rightarrow XY \bullet Z, \alpha]$

I_j represents all the simultaneously valid states

LR(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(1) states

S_0: [S′ ::= • E , \$],
[E ::= • T + E , \$], FIRST(ϵ \$) = \$
[E ::= • T , \$], FIRST(ϵ \$) = \$
[T ::= • **id** , +] FIRST(+ E \$) = +
[T ::= • **id** , \$] FIRST(ϵ \$) = \$

S_1: [S′ ::= E • , \$]

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

S_3: [T ::= **id** • , +]
[T ::= **id** • , \$]

S_4: [E ::= T + • E , \$],
[E ::= • T + E , \$], FIRST(ϵ \$) = \$
[E ::= • T , \$], FIRST(ϵ \$) = \$
[T ::= • **id** , +] FIRST(+ E \$) = +
[T ::= • **id** , \$] FIRST(ϵ \$) = \$

S_5: [E ::= T + E • , \$]

Example GOTO function

Start

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

Iteration 1

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

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

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

Iteration 2

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

Iteration 3

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

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

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

Example ACTION and GOTO tables

The Augmented Grammar

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

	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	—	—

The "reduce" actions are determined by the lookahead entries in the LR(1) items (instead of FOLLOW as in SLR parsers)

The *dfa*, **ACTION** and **GOTO** tables have the exact same format for both SLR(1) and LR(1) parsers

Resolving parse conflicts

Parse conflicts possible when certain LR items are found in the same state.

Depending on parser, may choose between LR items using lookahead.

Legal lookahead for LR items must be disjoint, else conflict exists.

	Shift-Reduce $[\ A ::= \alpha \bullet\ , \delta\ ]$ $[\ B ::= \beta \bullet \gamma, \eta\ ]$	Reduce-Reduce $[\ A ::= \alpha \bullet\ , \delta\ ]$ $[\ B ::= \beta \bullet, \eta\ ]$
LR(0)	conflict	conflict
SLR(1)	FOLLOW(A) $\cap$ FIRST(γ)	FOLLOW(A) $\cap$ FOLLOW(B)
LR(1)	$\delta \cap$ FIRST(γ)	$\delta \cap \eta$

SLR(1) parsing example

The Grammar

$$
\begin{array}{lll}
\mathrm{S}' & ::= & \mathrm{G} \\
\mathrm{G} & ::= & \mathrm{E}\ \texttt{=}\ \mathrm{E} \mid \texttt{id} \\
\mathrm{E} & ::= & \mathrm{E}\ \texttt{+}\ \mathrm{T} \mid \mathrm{T} \\
\mathrm{T} & ::= & \mathrm{T}\ \texttt{*}\ \texttt{f} \mid \texttt{id}
\end{array}
$$

S_0: $[\ \mathrm{S}' ::= \bullet\ \mathrm{G}\ ]$

$[\ \mathrm{G} ::= \bullet\ \mathrm{E}\ \texttt{=}\ \mathrm{E}\ ]$

$[\ \mathrm{G} ::= \bullet\ \texttt{id}\ ]$

$[\ \mathrm{E} ::= \bullet\ \mathrm{E}\ \texttt{+}\ \mathrm{T}\ ]$

$[\ \mathrm{E} ::= \bullet\ \mathrm{T}\ ]$

$[\ \mathrm{T} ::= \bullet\ \mathrm{T}\ \texttt{*}\ \texttt{id}\ ]$

$[\ \mathrm{T} ::= \bullet\ \texttt{id}\ ]$

S_1: $[\ \mathrm{G} ::= \texttt{id}\ \bullet\ ]$ $\quad$ FOLLOW(G) $= \{\ \$\ \}$

$[\ \mathrm{T} ::= \texttt{id}\ \bullet\ ]$ $\quad$ FOLLOW(T) $= \{\ \$,\ \texttt{*},\ \texttt{+},\ \texttt{=}\ \}$

Reduce-reduce conflict in S_1 for lookahead \$!

LR(1) parsing example

The Grammar

$$
\begin{array}{lcl}
\mathrm{S}' & ::= & \mathrm{G} \\
\mathrm{G} & ::= & \mathrm{E} \texttt{ = } \mathrm{E} \mid \texttt{id} \\
\mathrm{E} & ::= & \mathrm{E} \texttt{ + } \mathrm{T} \mid \mathrm{T} \\
\mathrm{T} & ::= & \mathrm{T} \texttt{ * } \texttt{id} \mid \texttt{id}
\end{array}
$$

S_0: $[\ \mathrm{S}' ::= \bullet\ \mathrm{G}\ ,\ \{\ \$\ \}\ ]$

$[\ \mathrm{G} ::= \bullet\ \mathrm{E} \texttt{ = } \mathrm{E}\ ,\ \{\ \$\ \}\ ]$

$[\ \mathrm{G} ::= \bullet\ \texttt{id}\ ,\ \{\ \$\ \}\ ]$

$[\ \mathrm{E} ::= \bullet\ \mathrm{E} \texttt{ + } \mathrm{T}\ ,\ \{\ \texttt{=},\ \texttt{+}\ \}\ ]$

$[\ \mathrm{E} ::= \bullet\ \mathrm{T}\ ,\ \{\ \texttt{=},\ \texttt{+}\ \}\ ]$

$[\ \mathrm{T} ::= \bullet\ \mathrm{T} \texttt{ * } \texttt{id}\ ,\ \{\ \texttt{=},\ \texttt{+},\ \texttt{*}\ \}\ ]$

$[\ \mathrm{T} ::= \bullet\ \texttt{id}\ ,\ \{\ \texttt{=},\ \texttt{+},\ \texttt{*}\ \}\ ]$

S_1: $[\ \mathrm{G} ::= \texttt{id}\ \bullet\ ,\ \{\ \$\ \}\ ]$

$[\ \mathrm{T} ::= \texttt{id}\ \bullet\ ,\ \{\ \texttt{=},\ \texttt{+},\ \texttt{*}\ \}\ ]$

Reduce-reduce conflict in S_1 resolved by lookahead!

LALR(1) parsing

LR(1) parsers have many more states than SLR(1) parsers (approximately factor of ten for Pascal).

LALR(1) parsers have same number of states as SLR(1) parsers, but with more power due to lookahead in states.

Define the *core* of a set of $LR(1)$ items to be the set of $LR(0)$ items derived by ignoring the lookahead symbols.

Thus, the two sets

- $\{[A \Rightarrow \alpha \bullet \beta, \mathsf{a}], [A \Rightarrow \alpha \bullet \beta, \mathsf{b}]\}$, and
- $\{[A \Rightarrow \alpha \bullet \beta, \mathsf{c}], [A \Rightarrow \alpha \bullet \beta, \mathsf{d}]\}$

have the same core.

Key Idea:

If two sets of $LR(1)$ items, I_i and I_j, have the same core, we can merge the states that represent them in the `ACTION` and `GOTO` tables.

LALR(1) table construction

There are two approaches to constructing LALR(1) parsing tables

Approach 1: Build LR(1) sets of items, then merge.

1. For each core present among the set of LR(1) items, find all sets having that core and replace these sets by their union
2. Update the *goto* function to reflect the replacement sets

The resulting algorithm has large space requirements

LALR(1) table construction

A more space efficient algorithm can be derived by observing that:

- we can represent I_i by its *kernel*, those items that are either the initial item $[S' \rightarrow \bullet S, \texttt{eof}]$ or do not have the $\bullet$ at the left end of the *rhs*.
- we can compute *shift*, *reduce*, and *goto* actions for the state derived from I_i directly from *kernel*(I_i).

This method avoids building the complete canonical collection of sets of LR(1) items.

Approach 2: Build LR(0) sets of items, then generate lookahead information.

1. Construct kernels of LR(0) sets of items
2. Initialize lookaheads of each kernel item
3. Compute when lookaheads propagate
4. Propagate lookaheads

LALR(1) properties

LALR(1) parsers have same number of states as SLR(1) parsers (core LR(0) items are the same)

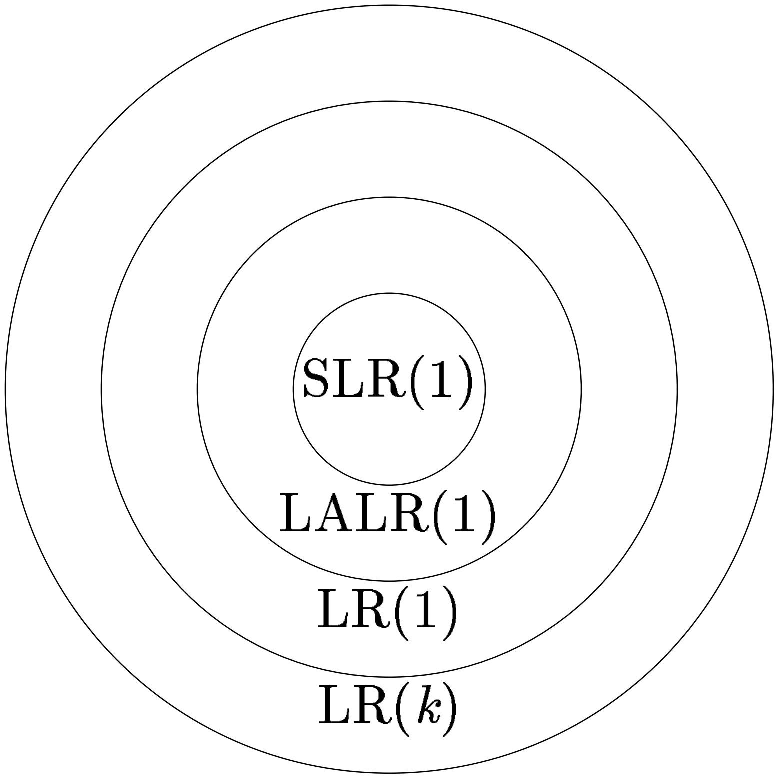
May perform *reduce* rather than *error*.

But will catch error before more input is processed.

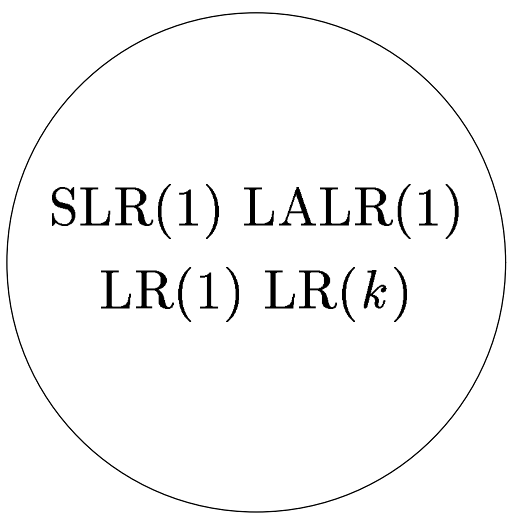
LALR derived from LR with no shift-reduce conflict will also have no shift-reduce conflict.

LALR may create reduce-reduce conflict not in LR from which LALR is derived.

$\mathtt{LR}(k)$ languages



Grammars



Languages

Operator precedence parsers

Another approach to shift-reduce parsing is to use *operator precedence.*

Given S $\Rightarrow^* \alpha S_1 S_2 \beta$, there are three possible *precedence relations* between S_1 and S_2.

1. S_1 in handle, S_2 not $\quad S_1 > S_2$
   (S_1 reduced before S_2)
2. both in handle $\quad S_1 = S_2$
   (reduced at same time)
3. S_2 in handle, S_1 not $\quad S_1 < S_2$
   (S_2 reduced before S_1)

A handle is thus composed of:

$<>, < = >, < = = >, \ldots$

To decide whether to shift or reduce, compare top of stack with lookahead (ignoring nonterminals):

- Shift if $<$ or $=$
- Reduce if $>$

Left end of handle is marked by first $<$ found

Parsing example

The Grammar

$$E ::= E + E \mid E * E \mid \texttt{id}$$

	+	*	id	$
+	$>$	$<$	$<$	$>$
*	$>$	$>$	$<$	$>$
id	$>$	$>$		$>$
$	$<$	$<$	$<$	$>$

Stack	Input	Precedence
$\$$	$\texttt{id} + \texttt{id} * \texttt{id}\ \$$	$\$ < \texttt{id}$
$\$ < \texttt{id}$	$+ \texttt{id} * \texttt{id}\ \$$	$\texttt{id} > +$
$\$ < E$	$+ \texttt{id} * \texttt{id}\ \$$	$\$ < +$
$\$ < E +$	$\texttt{id} * \texttt{id}\ \$$	$+ < \texttt{id}$
$\$ < E + < \texttt{id}$	$* \texttt{id}\ \$$	$\texttt{id} > *$
$\$ < E + < E$	$* \texttt{id}\ \$$	$+ < *$
$\$ < E + < E *$	$\texttt{id}\ \$$	$* < \texttt{id}$
$\$ < E + < E * < \texttt{id}$	$\$$	$\texttt{id} > \$$
$\$ < E + < E * E$	$\$$	$* > \$$
$\$ < E + E$	$\$$	$+ > \$$
$\$ < E$	$\$$	$\$ > \$$

Parsing review

Recursive Descent A hand coded recursive descent parser directly encodes a grammar (typically an LL(1) grammar) into a series of mutually recursive procedures. It has most of the linguistic limitations of LL(1).

LL(k) An LL(k) parser must be able to recognize the use of a production after seeing only the first k symbols of its right hand side.

LR(k) An LR(k) parser must be able to recognize the occurrence of the right hand side of a production after having seen all that is derived from that right hand side with k symbols of lookahead.

Parsing review

	Advantages	*Disadvantages*
top-down recursive descent	fast locality simplicity error detection	hand-coded maintenance no left recursion associativity
LL(1)	simple method fast automatable	$\text{LL}(1) \subset \text{LR}(1)$ no left recursion associativity
operator precedence	simple method very fast small table associativity	$L(G) \neq L(\text{parser})$ error detection no ambiguity
LR(1)	fast *det. langs.* early error det. automatable associativity	working sets table size error recovery