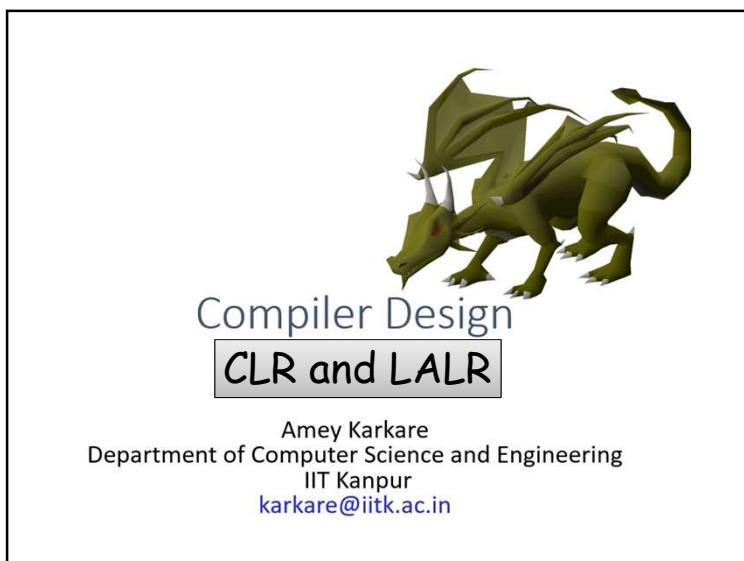


follow set is not enough to decide whether to shift/reduce.



Canonical LR Parsing

- Carry extra information in the state so that wrong reductions by $A \rightarrow \alpha$ will be ruled out
 - Redefine LR items to include a terminal symbol as a second component (look ahead symbol) $\{ \text{lookahead symbol} \} \subset \text{follow set}$
 - The general form of the item becomes $[A \rightarrow \underline{\alpha}, \underline{\beta}, a]$ which is called LR(1) item.
 - Item $[A \rightarrow \alpha., a]$ calls for reduction only if next input is a. The set of symbols "a"s will be a subset of $\text{Follow}(A)$.

$\alpha \nparallel \alpha$

Closure(I)

repeat
 for each item $[A \rightarrow \alpha.B\beta, a]$ in I
 for each production $B \rightarrow y$ in G'
 and for each terminal b in $\text{First}(B\beta)$
 add item $[B \rightarrow \cdot y, b]$ to I
 until no more additions to I

This is left context. We are adding $[B \Rightarrow \gamma, b]$ provided we have seen Δ .

$$\text{follow}(c) = \text{first}(c) = \{c, d, \$\}$$

Example

Consider the following grammar

$S' \rightarrow S$
 $S \rightarrow CC$
 $C \rightarrow cC \quad | \quad d$

} anything that follows(S) \in follow(C)
= \$

Compute closure(I) where $I = \{[S' \rightarrow .S, S]\}$

$S' \rightarrow S,$	\$	$s \rightarrow r, -$	$\text{first}(S) = \$$
$S \rightarrow .CC,$	\$		
$C \rightarrow .cC,$	c		
$C \rightarrow .cC,$	d		
$C \rightarrow .d,$	c		
$C \rightarrow .d,$	d		

So here, we are expanding first C & this will never change & following it 3

C.C \Rightarrow 2nd C can have \$ following, so we'll include those rules as well.

Example

Construct sets of LR(1) items for the grammar on previous slide

$I_0: S' \rightarrow .S,$	\$	$I_4: \text{goto}(I_0, d)$
$S \rightarrow .CC,$	\$	$C \rightarrow d,$
$C \rightarrow .CC,$	c/d	c/d
$C \rightarrow .d,$	c/d	
$I_1: \text{goto}(I_0, S)$	\$	$I_5: \text{goto}(I_2, C)$
$S \rightarrow S.,$	\$	$S \rightarrow CC.,$
$I_2: \text{goto}(I_0, C)$	\$	$I_6: \text{goto}(I_2, C)$
$S \rightarrow CC,$	\$	$C \rightarrow .C.C,$
$C \rightarrow .CC,$	\$	$C \rightarrow .C,$
$C \rightarrow .d,$	\$	$C \rightarrow .d,$
$I_3: \text{goto}(I_0, C)$	c/d	$I_7: \text{goto}(I_2, d)$
$C \rightarrow c.C,$	c/d	$C \rightarrow d,$
$C \rightarrow .c.C,$	c/d	\$
$C \rightarrow .d,$	c/d	
		$I_8: \text{goto}(I_3, C)$
		$C \rightarrow CC.,$
		c/d
		$I_9: \text{goto}(I_6, C)$
		$C \rightarrow cC.,$
		\$

5

Construction of Canonical LR parse table

- Construct $C = \{I_0, \dots, I_n\}$ the sets of LR(1) items.
- If $[A \rightarrow \alpha.a\beta, b]$ is in I_i and $\text{goto}(I_i, a) = I_j$, then action[i, a] = shift j
- If $[A \rightarrow \alpha., a]$ is in I_i , then action[i, a] reduce $A \rightarrow \alpha$
- If $[S' \rightarrow S., \$]$ is in I_i , then action[i, \\$] = accept
- If $\text{goto}(I_i, A) = I_j$, then goto[i, A] = j for all non terminals A

6

Parse table

State	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

7

Notes on Canonical LR Parser

- Consider the grammar discussed in the previous two slides. The language specified by the grammar is c^*dc^*d .
- When reading input cc...dcc...d the parser shifts cs into stack and then goes into state 4 after reading d. It then calls for reduction by $C \rightarrow d$ if following symbol is c or d.
- IF \$ follows the first d then input string is c^*d which is not in the language; parser declares an error.
- On an error canonical LR parser never makes a wrong shift/reduce move. It immediately declares an error.

✓ Problem: Canonical LR parse table has a large number of states

We'll use a parser which is b/w
SLR and CLR \Rightarrow LALR

8

LALR Parse table

- Look Ahead LR parsers
- Consider a pair of similar looking states (same kernel and different lookahead) in the set of LR(1) items
 $I_4: C \rightarrow d., c/d$ $I_7: C \rightarrow d., \$$
- Replace I_4 and I_7 by a new state I_{47} consisting of $(C \rightarrow d., c/d/\$)$
- Similarly I_3 & I_6 and I_8 & I_9 form pairs
- Merge LR(1) items having the same core

9

Construct LALR parse table

- Construct $C = \{I_0, \dots, I_n\}$ set of LR(1) items
- For each core present in LR(1) items find all sets having the same core and replace these sets by their union
- Let $C' = \{J_0, \dots, J_m\}$ be the resulting set of items
- Construct action table as was done earlier
- Let $J = I_1 \cup I_2 \cup \dots \cup I_k$
 since I_1, I_2, \dots, I_k have same core, goto(J,X) will have the same core.
- Let $K = \text{goto}(I_1, X) \cup \text{goto}(I_2, X) \cup \dots \cup \text{goto}(I_k, X)$ then goto(J,X)=K

goto states get merged as the same core exists
 eq. If $A \rightarrow a.XB, \dots$ $\Rightarrow A \rightarrow a.X.B, \dots$
 $I_j: A \rightarrow a.XB, \dots \Rightarrow T_K$

Notes on LALR parse table

- Modified parser behaves as original except that it will reduce $C \rightarrow d$ on inputs like ccd . The error will eventually be caught before any more symbols are shifted.
CLR detect immediately
- In general core is a set of LR(0) items and LR(1) grammar may produce more than one set of items with the same core.
- Merging items never produces shift/reduce conflicts but may produce reduce/reduce conflicts.
- $\text{follow}(A) \rightarrow \text{subset of follow}(A)$
 SLR and LALR parse tables have same number of states.
they differ in lookahead symbols

LALR parse table ...

State	c	d	\$	S	C
0	s36	s47		1	2
1			acc		
2	s36	s47			5
36	s36	s47			89
47	r3	r3	r3		
5			r1		
89	r2	r2	r2		

11

12

☐ Can all unambiguous grammars be parsed using CLR \Rightarrow No.

☐ Shift-reduce conflicts can occur in CLR
 reduce-reduce

☐ LALR can't have new shift-reduce conflicts

* All practical parsers are LALR
 \rightarrow if/else

Notes on LALR parse table...

- Merging items may result into conflicts in LALR parsers which did not exist in LR parsers $\xrightarrow{\text{CLR}}$
- New conflicts can not be of shift-reduce kind:
 - Assume there is a shift-reduce conflict in some state of LALR parser with items $\{[X \rightarrow \alpha, a], [Y \rightarrow \gamma.a\beta, b]\}$
 - Then there must have been a state in the LR parser with the same core
 - Contradiction; because LR parser did not have conflicts
- LALR parser can have new reduce-reduce conflicts
 - Assume states $\{[X \rightarrow \alpha, a], [Y \rightarrow \beta, b]\}$ and $\{[X \rightarrow \alpha, b], [Y \rightarrow \beta, a]\}$
 - Merging the two states produces $\{[X \rightarrow \alpha, a/b], [Y \rightarrow \beta, a/b]\}$

13

14

Notes on LALR parse table...

- LALR parsers are not built by first making canonical LR parse tables
- There are direct, complicated but efficient algorithms to develop LALR parsers
- Relative power of various classes
 - $\text{SLR}(1) \leq \text{LALR}(1) \leq \text{LR}(1)$
 - $\text{SLR}(k) \leq \text{LALR}(k) \leq \text{LR}(k)$
 - $\text{LL}(k) \leq \text{LR}(k)$

Ambiguous
 Unambiguous
 CLR
 LALR
 SLR
 SLR with 2 lookahead
 may be better than
 LALR with 1 lookahead

Error Recovery

- An error is detected when an entry in the action table is found to be empty.
- Panic mode error recovery can be implemented as follows:
 - scan down the stack until a state S with a goto on a particular nonterminal A is found.
 - discard zero or more input symbols until a symbol a is found that can legitimately follow A .
 - stack the state $\text{goto}[S, A]$ and resume parsing.
- Choice of A, a :** Normally A is chosen from non terminals representing major program pieces such as an expression, statement or a block. For example if A is the nonterminal **stmt**, a might be **semicolon** or **end**.

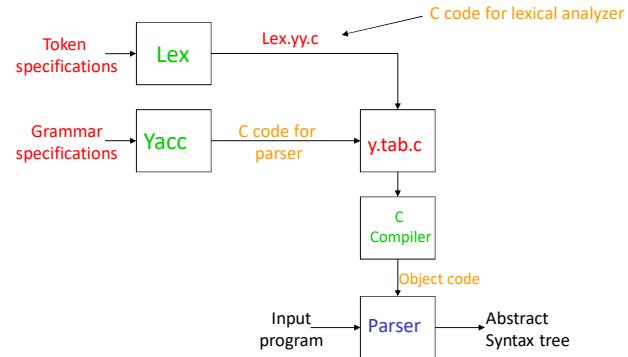
15

Parser Generator

- Some common parser generators
 - YACC: Yet Another Compiler Compiler
 - Bison: GNU Software
 - ANTLR: ANOther Tool for Language Recognition
- Yacc/Bison source program specification (accept LALR grammars)
 - declaration
 - %%
 - translation rules
 - %%
 - supporting C routines

16

Yacc and Lex schema



Refer to YACC Manual

17

Bottom up parsing ...

- A more powerful parsing technique
- LR grammars – more expensive than LL
- Can handle left recursive grammars
- Can handle virtually all the programming languages
- Natural expression of programming language syntax
- Automatic generation of parsers (Yacc, Bison etc.)
- Detects errors as soon as possible
- Allows better error recovery

18