CS412/413

Introduction to
Compilers and Translators
Spring '00

Lecture 6: LR grammars and automatic parser generators

Outline

- Review of shift-reduce parsers
- Limitations of LR(0) grammars
- SLR, LR(1), LALR parsers
- parser generators

Administration

- Programming Assignment 1 due now!
- Homework 2 due in 1 week
- Programming Assignment 2 due in 2 + ε weeks

Bottom-up parsing

- Apply productions *backwards* as *reductions*
- Builds parse tree from terminal symbols up towards start symbol
- Shift-reduce parser constructs right-most derivation using sequence of *shift* and *reduce* operations

Shift-reduce parsing

- Parsing is a sequence of *shifts* and *reduces*
- **Shift --** push look-ahead token onto stack

stack	input	
(	1+2+(3+4))+5	*shift* 1
(1	+2+(3+4))+5	

- **Reduce** -- Replace symbols γ in top of stack with non-terminal symbol X, corresponding to production X → γ (pop γ, push X)

stack	input	
(S+E	+(3+4))+5	*reduce* S→ S+*E*
(S	+(3+4))+5	

LR(0) states

- Problem: when to reduce & what production?
- Idea: state summarizes parser stack prefix, keeps track of what productions might need to be the next reduce
  - what tokens can be shifted
  - when a production is complete

state

$E \rightarrow .\ id$
$E \rightarrow (\ .\ S\ )$

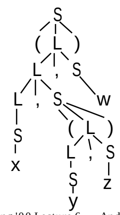
item

- A state is a set of *items:* partially completed productions that might appear in derivation

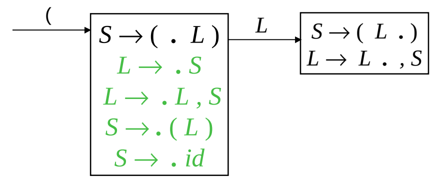
An LR(0) grammar: lists

$S \to id \mid (L)$
$L \to L , S \mid S$

(x,(y,z),w)

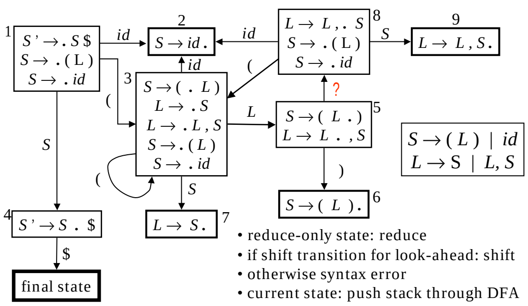


Constructing states



$S \to (L) \mid id$
$L \to S \mid L, S$

Full DFA (Appel p. 63)



- reduce-only state: reduce
- if shift transition for look-ahead: shift
- otherwise syntax error
- current state: push stack through DFA

Determining current state

- Run parser stack through the DFA
- State tells us what productions might be reduced next

stack	input	
((L,	x), y)	state = ?
		action = ?
((*id*,(		?

Optimization

- Attach parser state to each stack entry
- Don't have to traverse DFA all over again after each reduction
- Just start from state at top of stack when RHS γ is removed, take single step for LHS non-terminal.

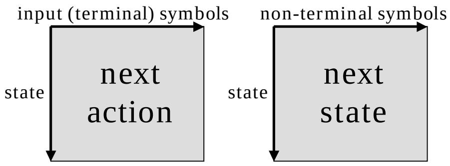
((*L*)	, y)	state = 6
(*S*	, y)	state = ?

$S \to (L) \mid id$
$L \to S \mid L, S$

- Stack entries labeled w/state

derivation	*stack*	*input*	*action*
((x),y) ←	${}_1$	((x),y)	shift, goto 3
((x),y) ←	${}_1({}_3$	(x),y)	shift, goto 3
((x),y) ←	${}_1({}_3({}_3$	x),y)	shift, goto 2
((x),y) ←	${}_1({}_3({}_3\mathrm{x}_2),y)$		reduce $S \to id$
(($\boldsymbol{S}$),y) ←	${}_1({}_3({}_3S_7),y)$		reduce $L \to S$
(($\boldsymbol{L}$),y) ←	${}_1({}_3({}_3L_5$	),y)	shift, goto 6
((L),y) ←	${}_1({}_3({}_3L_5)_6$	,y)	reduce $S \to (L)$
($\boldsymbol{S}$,y) ←	${}_1({}_3S_7$	,y)	reduce $L \to S$
($\boldsymbol{L}$,y) ←	${}_1({}_3L_5$	,y)	shift, goto 8
(L,y) ←	${}_1({}_3L_5,{}_8$	y)	shift, goto 9
(L,y) ←	${}_1({}_3L_5,{}_8\mathrm{y}_2$	)	reduce $S \to id$
(L,$\boldsymbol{S}$) ←	${}_1({}_3L_5,{}_8S_9$	)	reduce $L \to L , S$
($\boldsymbol{L}$) ←	${}_1({}_3L_5$	)	shift, goto 6
($\boldsymbol{L}$) ←	${}_1({}_3L_5)_6$		reduce $S \to (L)$
$\boldsymbol{S}$	${}_1S_4$	\$	*done*

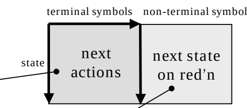
Implementation: LR parsing table



Action table
Used at every step to decide whether to shift or reduce

Goto table
Used only when reducing, to determine next state

Shift-reduce parser table



- Actions in table
  1. shift and goto state *n*
  2. reduce using $X \to \gamma$
     - pop symbols γ off stack
     - using state label of top (end) of stack, look up X in *goto table* and goto that state
- DFA + stack = push-down automaton (PDA)

List grammar parse table

	(	)	*id*	,	$	*S*	*L*
1	s3		s2			g4	
2	$S\to id$	$S\to id$	$S\to id$	$S\to id$	$S\to id$		
3	s3		s2			g7	g5
4					accept		
5		s6		s8			
6	$S\to(L)$	$S\to(L)$	$S\to(L)$	$S\to(L)$	$S\to(L)$		
7	$L\to S$	$L\to S$	$L\to S$	$L\to S$	$L\to S$		
8	s3		s2			g9	
9	$L\to L,S$	$L\to L,S$	$L\to L,S$	$L\to L,S$	$L\to L,S$		

LR(0) Limitations

- An LR(0) machine only works if states with reduce actions have a *single* reduce action -- in those states, *always* reduce ignoring input
- With more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Need to use look-ahead to choose

ok	shift /reduce	reduce / reduce
$\boxed{L \to L , S .}$	$L \to L , S .$ $S \to S . , L$	$L \to S , L .$ $L \to L .$

List grammar parse table

	(	)	*id*	,	$	*S*	*L*
1	s3		s2			4	
2	$S\to id$	$S\to id$	$S\to id$	$S\to id$	$S\to id$		
3	s3		s2			7	5
4					accept		
5		s6		s8			
6	$S\to(L)$	$S\to(L)$	$S\to(L)$	$S\to(L)$	$S\to(L)$		
7	$L\to S$	$L\to S$	$L\to S$	$L\to S$	$L\to S$		
8	s3		s2			9	
9	$L\to L,S$	$L\to L,S$	$L\to L,S$	$L\to L,S$	$L\to L,S$		

How about sum grammar?

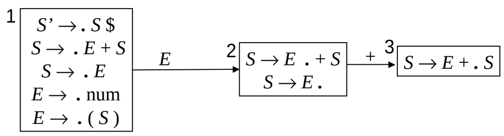
$$S \to S + E \mid E$$
$$E \to \text{num} \mid (S)$$

- This is LR(0)
- Right-associative version isn't:

$$S \to E + S \mid E$$
$$E \to \text{num} \mid (S)$$

LR(0) construction

$S \to E + S \mid E$
$E \to \text{num} \mid (S)$



What to do on +?

	+	\$	E
1			2
2	s3/S→E	S→E	

SLR grammars

$S \to E + S \mid E$
$E \to \text{num} \mid (S)$

- Idea: Only add reduce action to table if look-ahead symbol is in the *FOLLOW* set of the non-terminal being reduced
- Eliminates some conflicts
- *FOLLOW*(S) = { \$,) }
- Many language grammars are SLR

	+	\$	E
1			2
2	s3	S→E	

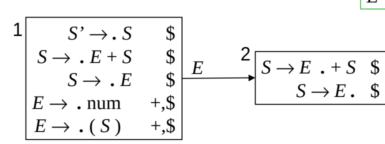
LR(1) parsing

- Gets as much power as possible out of 1 look-ahead symbol
- LR(1) grammar = recognizable by a shift/reduce parser with 1 look-ahead.
- LR(1) items keep track of look-ahead symbol expected to follow this production

LR(0): $S \to\ .\ S + E$
LR(1): $S \to\ .\ S + E \qquad +$

LR(1) construction

$S \to E + S \mid E$
$E \to \text{num} \mid (S)$



Know what to do if:
- reduce look-aheads distinct
- not to right of any dot

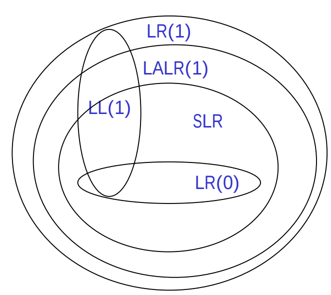
	+	\$	E
1			2
2	s3	S→E	

LALR grammars

- Problem with LR(1): too many states
- LALR(1) (Look-Ahead LR)
  - Merge any two LR(1) states whose items are identical except look-ahead
  - Results in smaller parser tables -- works extremely well in practice

$\boxed{\begin{array}{ll} S \to id\ . & + \\ S \to E\ . & \$ \end{array}} + \boxed{\begin{array}{ll} S \to id\ . & \$ \\ S \to E\ . & + \end{array}} = ?$

Classification of Grammars



How are parsers written?

- Automatic parser generators: yacc, bison, CUP
- Accept LALR(1) grammar specification
- *plus:* declarations of precedence, associativity

Associativity

$$S \rightarrow S + E \mid E$$
$$E \rightarrow \text{num} \mid (S)$$
$$\Downarrow$$
$$E \rightarrow E + E \mid \text{num} \mid (E)$$

What happens if we run this grammar through LALR construction?

Conflict!

$$E \rightarrow E + E \mid \text{num} \mid (E)$$

$$\boxed{\begin{array}{ll} E \rightarrow E + E\,. & + \\ E \rightarrow E\,. + E & +,\$ \end{array}}$$

shift/reduce conflict

1+2+3
 ^

shift: 1+(2+3)
reduce: (1+2)+3

Grammar in CUP

non terminal E; terminal PLUS, LPAREN...
precedence left PLUS;

"When shifting + conflicts with reducing a production containing +, choose reduce"

```
E ::= E  PLUS  E
    |  LPAREN  E  RPAREN
    |  NUMBER ;
```


Precedence

- Also can handle operator precedence

$$E \rightarrow E + E \mid T$$
$$T \rightarrow T \times T \mid \text{num} \mid (E)$$
$$\Downarrow$$
$$E \rightarrow E + E \mid E \times E$$
$$\mid \text{num} \mid (E)$$

Conflicts w/o precedence

$$E \rightarrow E + E \mid E \times E$$
$$\mid \text{num} \mid (E)$$

$$\boxed{\begin{array}{ll} E \rightarrow E\,. + E & \textit{any} \\ E \rightarrow E \times E\,. & + \end{array}} \quad \boxed{\begin{array}{ll} E \rightarrow E + E\,. & \times \\ E \rightarrow E\,. \times E & \textit{any} \end{array}}$$

Predecence in CUP

precedence left PLUS;
precedence left TIMES; // TIMES > PLUS
E ::= E PLUS E | E TIMES E | ...

$E \rightarrow E \,.\, + E$	*any*
$E \rightarrow E \times E \,.$	$+$

$E \rightarrow E + E \,.$	$\times$
$E \rightarrow E \,.\, \times E$	*any*

Rule: in conflict, choose **reduce** if production symbol higher precedence than shifted symbol; choose **shift** if vice-versa

Summary

- Look-ahead information makes SLR(1), LALR(1), LR(1) grammars expressive
- Automatic parser generators support LALR(1)
- Precedence, associativity declarations simplify grammar writing
- Can we use parsers for programs other than compilers?