

LR Parsers

- we still want to construct parse tree
 - find the leftmost node that has not yet been constructed, but all of whose children have been constructed
- we have a sequence of children, need to determine
 - appropriate sequence that represents the correct right-hand-side, and
 - which non-terminal does this sequence correspond to

- LR(k) parsers have
 - *stack*: maintain states and ‘constructed/seen children’, and
 - *input* – with k tokens *lookahead*
 - actions:
 - *shift* (push to top of stack - tokens from input)
 - *reduce* (pop contents from top of the stack - ‘children’, push to the top of the stack - ‘the parent’)
 - grammar rules give hypothesis which set of children are we looking for, and which action should we apply
 - LR(0) – no lookahead, just what’s on the stack.

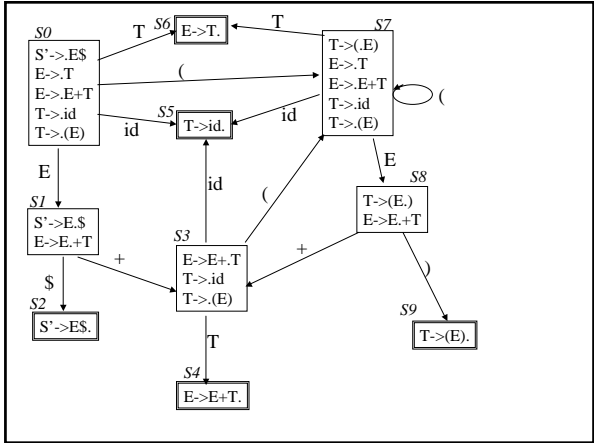
- LR item
 - *ab* is possible, and should be reduced to N if found, and
 - we have just recognized *a*
- $N \rightarrow a.b$ – shift item
- $N \rightarrow ab.$ – reduce item
- if ‘.’ is in front of a non-terminal, we have to make sure to consider all possible alternatives

LR(0)

$S' \rightarrow E \$$
 $E \rightarrow T \mid E + T$
 $T \rightarrow id \mid (E)$

- initial item set S_0
 - $S' \rightarrow . E \$$
 - $E \rightarrow . T$
 - $E \rightarrow . E + T$
 - $T \rightarrow . id$
 - $T \rightarrow . (E)$
- example with input $id + id \$$

- DFA applied to the stack to tell us actions to perform (shift/reduce/goto state)
- edges are grammar symbols, terminals and non-terminals
- states are represented with LR item sets
- actions:
 - shift: push token on stack, push new state on stack
 - reduce: pop k symbols for right-hand-side off the stack and push left-hand-side non terminal on stack
 - goto: advance past newly recognized non-terminal



	id	+	()	\$	E	T
S0	s5		s7		g1	g6
S1		s3		s2		
S2		r0 (S' → E\$)				
S3	s5		s7			g4
S4		r2 (E → E+T)				
S5		r3 (T → id)				
S6		r1 (E → T)				
S7	s5		s7		g8	g6
S8		s3		s9		
S9		r4 (T → (E))				

- how do we build DFA and parsing table:
- starting rule forms the initial item set
 - $I = S' \rightarrow \dots \$$
 - compute item all items sets. compute edges b/w item sets

```

Closure(I)
  repeat
    for each A → a.Xb in I
      for each X → g
        I := I U { X → .g }
    until I doesn't change

Goto(I,X)
  set J to empty
  for each A → a.Xb in I
    J := J U { A → aX.b }
  return Closure(J)

```

```

States := Closure{S'->.E$}
Edges := Empty
repeat
  for each state I in States
    for each item A->a.Xb in I
      let J be Goto(I,X)
      States := States U {J}
      Edges := Edges U {I->J on X}
until States and Edges don't change

```

- states with outgoing edges have corresponding shift entries
- states with *one* reduce item are reduce states for corresponding rule
- otherwise – conflict!!!

- shift-reduce conflict
T -> id [E], then S5:
T-> id .
T-> id . [E]
- reduce-reduce conflict
S' -> V := E\$, V-> id, then S5:
T-> id.
V->id.
- conflicts are frequent (always with eps-rules)
- LR(0) – reasonable size parsing table, but lots of conflicts... need something more powerful

SLR(1)

- Simple LR(1)
- Use FOLLOW Sets and lookahead. Determine which rule to apply/action to take based on FOLLOW set and next token.
E -> T . + E (shift)
E -> T . (reduce)
- in parsing table if lookahead matches a token from FOLLOW(E) then reduce action is valid.

- may still have conflicts:
- a state includes multiple production rules
- action should depend on lookahead for *that* rule in *that* state
- FOLLOW Sets contain info for all possible alternatives of N from all possible States in the PDA.

LR(1)

- keep lookahead set with each separate item
- parser more powerful, but increase in state
- LR(1) item:
 - $(A \rightarrow a.b, x)$ – production rule with position, a has been parsed and is on top of stack, ahead is a string derivable from “ $b\ x$ ”

```

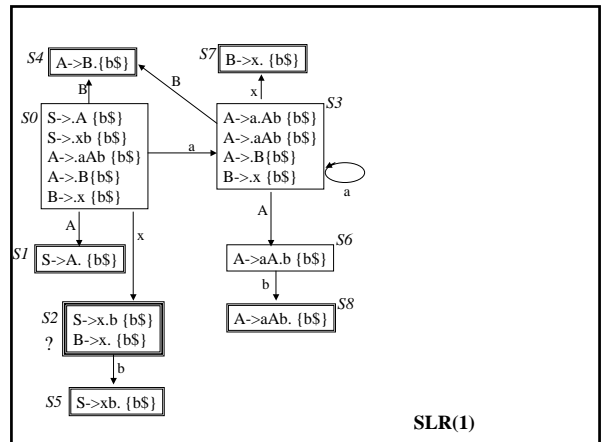
Closure(I)
  repeat
    for each  $(A \rightarrow a.Xb, z)$  in I
      for each  $X \rightarrow g$ 
         $I := I \cup \{ (X \rightarrow .g, w) \mid w \text{ in } \text{First}(bz) \}$ 
      until I doesn't change

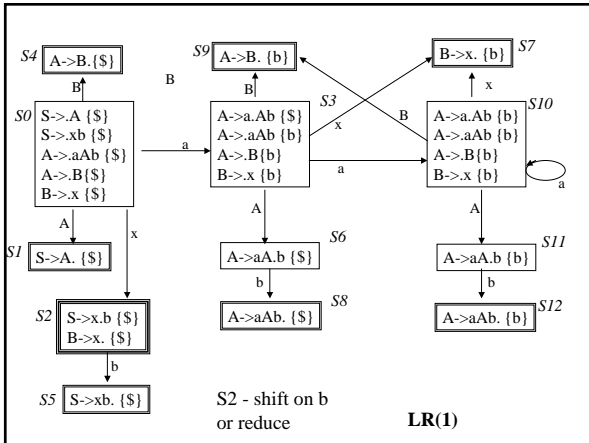
Goto(I, X)
  set J to empty
  for each  $(A \rightarrow a.Xb, z)$  in I
     $J := J \cup \{ (A \rightarrow aX.b, z) \}$ 
  return Closure(J)
    
```

LR(1) example

$S \rightarrow A$
 $S \rightarrow xb$
 $A \rightarrow aAb$
 $A \rightarrow B$
 $B \rightarrow x$

$\text{FOLLOW}(A) = \text{FOLLOW}(B) = \{b, \$\}$





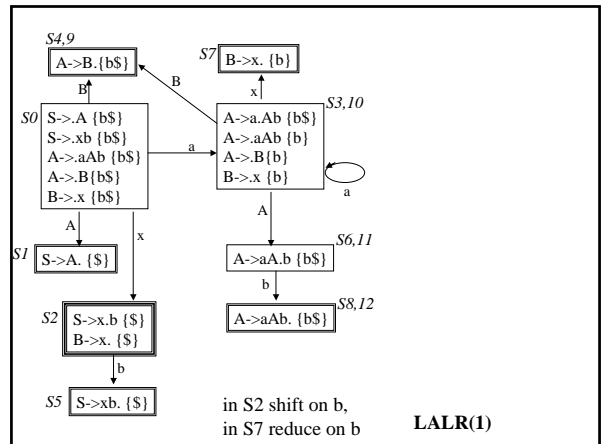
	a	b	x	\$	A	B
S_0	s3		s2		g1	g4
S_1				r0		
S_2		s5		r4		
S_3	s10		s7		g6	g9
S_4				r3		
S_5				r1		
S_6		s8				
S_7		r4				
S_8					r2	
S_9		r3				
S_{10}	s10		s7		g11	g9
S_{11}		s12				
S_{12}		r2				

LR(1)

LALR(1)

- LR(1) orders of magnitude size increase compared to LR(0)/SLR(1)
- with LR(1) may have many states with same cores, but different lookahead sets
- Look-Ahead LR – LALR(1) core contents are determined only by results of shifts from other states.

LR more powerful than LALR
 LALR almost as powerful, but smaller tables
 SML simplestweakest



	a	b	x	\$	A	B
<i>S0</i>	s3,10		s2		g1	g4,9
<i>S1</i>				r0		
<i>S2</i>		s5		r4		
<i>S3,10</i>	s3,10		s7		g6,11	g4,9
<i>S4,9</i>		r3		r3		
<i>S5</i>				r1		
<i>S6,11</i>		s8,12				
<i>S7</i>		r4				
<i>S8,12</i>		r2		r2		

LALR(1)

Ambiguous Grammars

- dangling else
 - if a then if b then s1 else s2:
 - if a then {if b then s1 else s2}
 - if a then {if b then s1} else s2
 - can rewrite grammar, or...
 - resolve by shifting!

- precedence
 - $E \rightarrow E + E \mid E * E$
 - $1 + 2 * 3$
 - can rewrite grammar, or...
 - add precedence and associativity directives
 - Shift: for right-assoc (^), or right operand higher precedence
 - Reduce: for left-assoc, or left operand has higher priority

ML-Yacc

- LALR parser generator
 - user declarations (e.g., help functions...)
 - %%
 - parser declarations (e.g., %term, %nonterm, %pos, %start, %prec, %eop, error recovery...)
 - %%
 - grammar rules (e.g., program: exp (exp), Tiger grammar rules)
 - declaration %verbose generates output – states and actions.

```
%nonassoc EQ NEQ
%left PLUS MINUS
%left TIMES DIV
%right EXP
```

- tokens have prec. based on decls; rules have prec. based on rightmost token.
 - if rule > token, reduce, else shift
 - if same, then %left -> reduce, %right -> shift
 - is nonassoc – error
- shift/reduce, and reduce/reduce conflicts – report error, resolve by shift (for S/R) or first reduction rule (for R/R)

Error Recovery

- ideally would not want to modify stack, but may need to
- special error symbol – you can build error production rules
- roll-back up to n tokens and check all possible alternative tokens (single-token substitution)
 - in ML-YACC can specify substitutions with %change
- semantic actions – specify values
 - %value