CS 4120
Introduction to Compilers

Ross Tate
Cornell University

Lecture 7: LR parsing and parser generators

Shift-reduce parsing

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

derivation	stack	input stream	action
(1+2+(3+4))+5 ←		(1+2+(3+4))+5	*shift*
(1+2+(3+4))+5 ←	(	1+2+(3+4))+5	*shift*
(1+2+(3+4))+5 ←	(1	+2+(3+4))+5	*reduce* $E \to$ n
(E+2+(3+4))+5 ←	(E	+2+(3+4))+5	*reduce* $S \to E$
(S+2+(3+4))+5 ←	(S	+2+(3+4))+5	*shift*
(S+2+(3+4))+5 ←	(S+	2+(3+4))+5	*shift*
(S+2+(3+4))+5 ←	(S+2	+(3+4))+5	*reduce* $E \to$ n
(S+E+(3+4))+5 ←	(S+E	+(3+4))+5	*reduce* $S \to S+E$
(S+(3+4))+5 ←	(S	+(3+4))+5	*shift*
(S+(3+4))+5 ←	(S+	(3+4))+5	*shift*
(S+(3+4))+5 ←	(S+(	3+4))+5	*shift*
(S+(3+4))+5 ←	(S+(3	+4))+5	*reduce* $E \to$ n

LR(0) states

- A state is a set of *items* keeping track of progress on possible upcoming reductions
- An *LR(0) item* is a production from the language with a separator "**.**" somewhere in the RHS of the production

state → $E \to \text{n}\ .$
item → $E \to (\ .\ S\ )$

- Stuff before "**.**" is already on stack (beginnings of possible γ's to be reduced)
- Stuff after "**.**" : what we might see next
- The prefixes α represented by state itself

LR(k) parsing

- As much power as possible out of parsing table with k lookahead symbols
- LR(1) grammar = recognizable by a shift/reduce parser with 1 lookahead
- LR(1) item = LR(0) + look-ahead symbols possibly following production

LR(0): S → . S + E

LR(1): S → . S + E +

Remaining input will reduce to S + E + ...

LR(1) state

- LR(1) state = set of LR(1) items
- LR(1) item = LR(0) item + 1 lookahead

$S \to S\ .\ + E$	+	$S \to S\ .\ + E$	+, $
$S \to S\ .\ + E$	$	shorthand	
$S \to S + .\ E$	n	$S \to S + .\ E$	n

LR(1) closure

Consider closure of item $A \to \beta\ .\ C\ \delta\quad \lambda$

Closure formed just as for LR(0) except

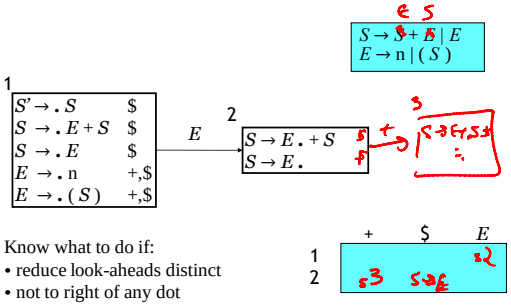
1. Lookahead symbols include characters following the non-terminal symbol to the right of dot: FIRST(δ)
2. If non-terminal symbol may produce last symbol of production (δ is nullable), lookahead symbols include lookahead symbols of production (λ)

$S' \to .\ S$ $
$S \to .\ E + S$ $
$S \to .\ E$ $
$E \to .\ \text{n}$ $
$E \to .\ (\ S\ )$ $

① ②

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

LR(1) construction



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

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 for look-ahead
  - Results in smaller parser tables—works extremely well in practice
  - Common technology for automatic parser generators

$S \to id\,.\;+$, $S \to E\,.\;\$$ + $S \to id\,.\;\$$, $S \to E\,.\;+$ = $S \to id\,.\;+,\$$, $S \to E\,.\;\$,+$

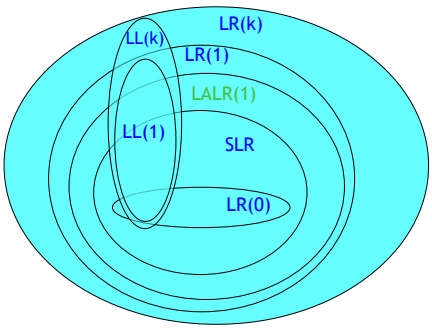
LALR(1) vs. LR(1)

S → a E c
→ a F d
→ b F c
→ b E d
E → e
F → e

LALR(1) as LR(0)+LA(1)

S → a E c
→ a F d
→ b F c
→ b E d
E → e
F → e

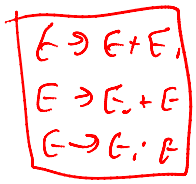
Classification of Grammars



How are parsers written?

- Automatic parser generators: yacc, bison, CUP
- Accept LALR(1) grammar specification
  - plus: declarations of precedence, associativity
  - output: LR parser code (inc. parsing table)
- Some parser generators accept LL(1), e.g. javacc – less powerful, or LL(k), e.g. ANTLR

Resolving Ambiguity



$E \rightarrow E + E \mid E * E$

When reducing op conflicts with shifting a production containing op
- choose reduce = left assoc.
- choose shift = right assoc.

When reducing op conflicts with shifting a production containing op'
- choose reduce = op ≥ op'
- choose shift = op < op'

13

Can we use parsers for programs *besides* compilers?

14