CS 412
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 7: LR parsing
7 Feb 01

Administration

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

Shift-reduce parsing

- Parsing is a sequence of *shift* and *reduce* operations
- Always constructs right-most derivation, backwards
- Parser state:
  - stack of terminals and non-terminals
  - unconsumed input is a string of terminals
  - Current derivation step is always stack+input

Derivation step	stack	unconsumed input
(1+2+(3+4))+5 ←		(1+2+(3+4))+5
(*E*+2+(3+4))+5 ←	(E	+2+(3+4))+5
(*S*+2+(3+4))+5 ←	(S	+2+(3+4))+5
(S+*E*+(3+4))+5 ←	(S+E	+(3+4))+5

Shift and Reduce Actions

- Parsing is a sequence of *shifts* and *reductions*
- **Shift** : move look-ahead token to stack

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

Action Selection Problem

- Given stack σ and look-ahead symbol *b*, should parser:
  - **shift** *b* onto the stack (making it σ*b*)
  - **reduce** some production $X \rightarrow \gamma$ assuming that stack has the form $\alpha\ \gamma$ (making it αX)
- If stack has form $\alpha\ \gamma$, should apply reduction $X \rightarrow \gamma$ (or shift) depending on stack prefix α
  - α is different for different possible reductions, since γ's have different length.
  - How to keep track of possible reductions?

Parser States

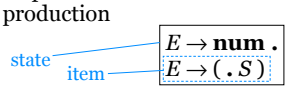
- Goal: know what reductions are legal at any given point
- Idea: summarize all possible stack prefixes α as a finite parser *state*
- Parser state is computed by a DFA that reads in the stack σ
- Accept states of DFA: unique reduction!
- Summarizing discards information
  - affects what grammars parser handles
  - affects size of DFA (number of states)

LR(0) parser

- **L**eft-to-right scanning, **R**ight-most derivation, "**zero**" look-ahead characters
- Too weak to handle most language grammars (*e.g.*, "sum" grammar)
- But will help us understand shift-reduce parsing

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

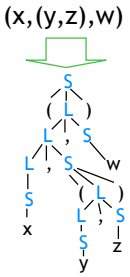


- Stuff before "**.**" is already on stack (beginnings of possible γs to be reduced)
- Stuff after "**.**" : what we might see next
- The prefixes α represented by state itself

An LR(0) grammar: non-empty lists

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

x (x,y) (x, (y,z), w)
((((x)))) (x, (y, (z, w)))



Start State & Closure

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

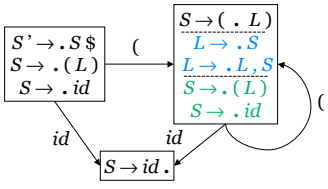
DFA start state: $S' \rightarrow . S \$$ —*closure*→ $S' \rightarrow . S \$$, $S \rightarrow . (L)$, $S \rightarrow . id$

Constructing a DFA to read stack:

- First step: augment grammar with prod'n $S' \rightarrow S \$$
- Start state of DFA: empty stack = $S' \rightarrow . S \$$
- *Closure* of a state adds items for all productions whose LHS occurs in an item in the state, just after "**.**"
  - set of possible productions to be reduced next
  - Added items have the "**.**" located at the beginning: no symbols for these items on the stack yet

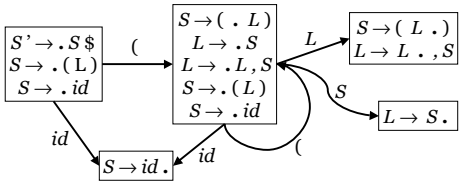
Applying terminal symbols

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



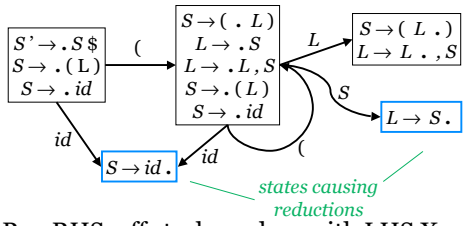
In new state, include all items that have appropriate input symbol just after dot, advance dot in those items, *and take closure.*

Applying non-terminals



- Non-terminals on stack treated just like terminals (except added by reductions)

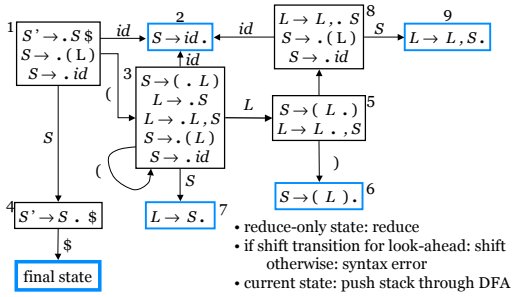
Applying reduce actions



- Pop RHS off stack, replace with LHS X (X→γ), rerun DFA (e.g. (x))

Full DFA (Appel p. 63)

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



- 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*,(		?

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

Parsing example: ((x),y)

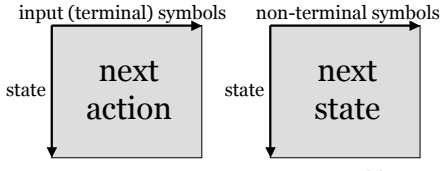
derivation	*stack*	*input*	*action*
((x),y) ←	$_1$	((x),y)	shift, goto 3
((x),y) ←	$_1(_3$	(x),y)	shift, goto 3
((x),y) ←	$_1(_3(_3$	x),y)	shift, goto 2
((x),y) ←	$_1(_3(_3x_2$	),y)	reduce $S \to id$
((***S***),y) ←	$_1(_3(_3S_7$	),y)	reduce $L \to S$
((***L***),y) ←	$_1(_3(_3L_5$	),y)	shift, goto 6
((*L*),y) ←	$_1(_3(_3L_5)_6$	,y)	reduce $S \to (L)$
(***S***,y) ←	$_1(_3S_7$	,y)	reduce $L \to S$
(***L***,y) ←	$_1(_3L_5$	,y)	shift, goto 8
(*L*,y) ←	$_1(_3L_5,_8$	y)	shift, goto 9
(*L*,y) ←	$_1(_3L_5,_8y_2$	)	reduce $S \to id$
(*L*,***S***) ←	$_1(_3L_5,_8S_9$	)	reduce $L \to L, S$
(***L***) ←	$_1(_3L_5$	)	shift, goto 6
(***L***) ←	$_1(_3L_5)_6$		reduce $S \to (L)$
S	$_1S_4$	$	*done*

Optimization

- Don't need to rerun DFA from beginning on every reduction
- On reducing $X \to \gamma$ with stack $\alpha\gamma$:
  - pop γ off stack, revealing prefix α *and state*
  - take single step in DFA from top state
  - push *X* onto stack with new DFA state

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

Implementation: LR parsing table



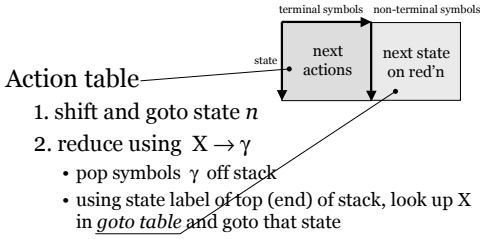
Action table
Used at every step to decide whether to shift or reduce

a → $X \to \gamma$.

Goto table
Used only when reducing, to determine next state

→ *X*

Shift-reduce parsing table



1. shift and goto state *n*
2. reduce using $X \rightarrow \gamma$
   - 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 parsing table

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

Shift-reduce parsing

- Grammars can be parsed bottom-up using a DFA + stack
  - DFA processes stack σ to decide what reductions might be possible given
  - *shift-reduce parser* or *push-down automaton (PDA)*
  - Compactly represented as *LR parsing table*
- State construction converts grammar into states that decide action to take

Checkpoint

- Limitations of LR(0) grammars
- SLR, LR(1), LALR parsers
- automatic parser generators

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 lookahead
- With more complex grammar, construction gives states with shift/reduce or reduce/reduce conflicts
- Need to use look-ahead to choose

ok

$L \rightarrow L, S.$

shift /reduce

$L \rightarrow L, S.$
$S \rightarrow S., L$

reduce / reduce

$L \rightarrow S, L.$
$L \rightarrow S.$

List grammar parsing table

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

An LR(0) grammar?

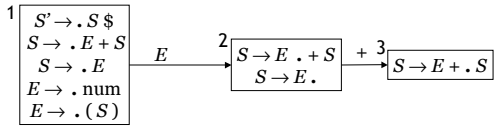
$S \rightarrow S + E \mid E$
$E \rightarrow \text{num} \mid (\ S\ )$

- Left-associative: LR(0)
- Right-associative version: not LR(0)

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

LR(0) construction

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



What to do in state 2?

	+	$	E
1			2
2	s3/S→E	S→E	

SLR grammars

- 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

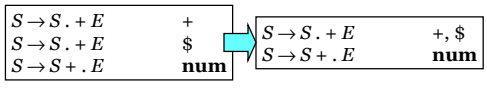
- As much power as possible out of 1 look-ahead symbol parsing table
- LR(1) grammar = recognizable by a shift/reduce parser with 1 look-ahead.
- LR(1) item = LR(0) item + look-ahead symbols possibly following production

LR(0): $S \rightarrow . S + E$

LR(1): $S \rightarrow . S + E$ +

LR(1) state

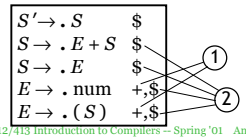
- LR(1) state = set of LR(1) items
- LR(1) item = LR(0) item + set of look-ahead symbols
- No two items in state have same production + dot configuration



LR(1) closure

Consider $A \rightarrow \beta . C \delta$ λ Closure formed just as for LR(0) *except*

1. Look-ahead 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), look-ahead symbols include look-ahead symbols of production (λ)



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

LR(1) construction

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

1

$S' \rightarrow . S$	\$
$S \rightarrow . E + S$	\$
$S \rightarrow . E$	\$
$E \rightarrow . \text{num}$	+,\$
$E \rightarrow . (S)$	+,\$

E →

2

$S \rightarrow E . + S$	\$
$S \rightarrow E .$	\$

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

	+	\$	E
1			2
2	s3	$S \rightarrow E$	

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
  - Usual technology for automatic parser generators

$S \rightarrow id . +$	+	$S \rightarrow id . \$$	= ?
$S \rightarrow E . \$$		$S \rightarrow E . +$	

Classification of Grammars