CS 4120
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 6: Bottom-Up Parsing

9/9/09

Administrivia

- Programming Assignment 1 due on Monday.
- Check class newsgroup `cornell.class.cs4120` for answers to frequently asked questions.

Bottom-up parsing

- A more powerful parsing technology
- LR grammars -- more expressive than LL
  - can handle left-recursive grammars, virtually all programming languages
  - Easier to express programming language syntax
- Shift-reduce parsers
  - construct right-most derivation of program
  - automatic parser generators (e.g., yacc, CUP, ocamlyacc)
  - detect errors as soon as possible
  - allows better error recovery

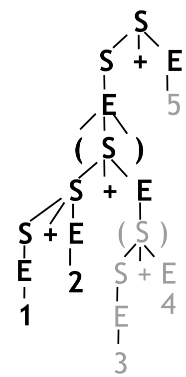
Top-down parsing

(1+2+(3+4))+5

$$S \rightarrow S\ |\ S + E\ |\ E$$

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

$$S \rightarrow S{+}E \rightarrow E{+}E \rightarrow (S){+}E \rightarrow (S{+}E){+}E \rightarrow (S{+}E{+}E){+}E \rightarrow (E{+}E{+}E){+}E \rightarrow (1{+}E{+}E){+}E \rightarrow (1{+}2{+}E){+}E \ldots$$



- In left-most derivation, entire tree above a token (2) has to be expanded when encountered
- Must be able to predict productions!

Bottom-up parsing

- Right-most derivation -- backward
  - Start with the tokens
  - End with the start symbol

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

(1+2+(3+4))+5 ← (*E*+2+(3+4))+5 ← (*S*+2+(3+4))+5 ←(S+*E*+(3+4))+5 ← (*S*+(3+4))+5 ← (S+(*E*+4))+5 ←(S+(*S*+4))+5 ←(S+(S+*E*))+5 ← (S+(*S*))+5 ←(S +*E*)+5 ← (*S*)+5 ← *E*+5 ← *S*+*E* ← *S*

Progress of bottom-up parsing

right-most derivation

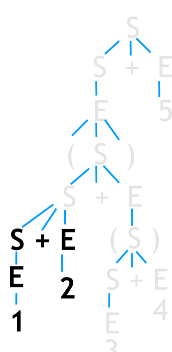
(1+2+(3+4))+5 ←		(1+2+(3+4))+5
(*E*+2+(3+4))+5 ←	(1	+2+(3+4))+5
(*S*+2+(3+4))+5 ←	(1	+2+(3+4))+5
(S+*E*+(3+4))+5 ←	(1+2	+(3+4))+5
(*S*+(3+4))+5 ←	(1+2+(3	+4))+5
(S+(*E*+4))+5 ←	(1+2+(3	+4))+5
(S+(*S*+4))+5 ←	(1+2+(3	+4))+5
(S+(S+*E*))+5 ←	(1+2+(3+4	))+5
(S+(*S*))+5 ←	(1+2+(3+4	))+5
(S+*E*)+5 ←	(1+2+(3+4)	)+5
(*S*)+5 ←	(1+2+(3+4)	)+5
E+5 ←	(1+2+(3+4))	+5
S+*E* ←	(1+2+(3+4))+5	
S	(1+2+(3+4))+5	

Bottom-up parsing

- (1+2+(3+4))+5 ←(*E*+2+(3+4))+5 ←(*S*+2+(3+4))+5 ←(S+*E*+(3+4))+5 ...

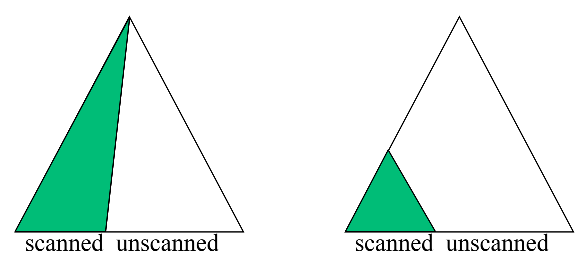
$S \rightarrow S + E \mid E$
$E \rightarrow \text{number} \mid (\ S\ )$

- **Advantage of bottom-up parsing: select productions using more information**



Top-down vs. Bottom-up

Bottom-up: Don’t need to figure out as much of the parse tree for a given amount of input



Top-down Bottom-up

Shift-reduce parsing

- Parsing is a sequence of *shift* and *reduce* operations
- Parser state is a stack of terminals and non-terminals (grows to the right)
- 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-reduce parsing

- Parsing is a sequence of *shifts* and *reduces*
- **Shift** : move lookahead token to stack. No effect on derivation.

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 $X \to \gamma$ (pop γ, push X). Reduces rightmost nonterminal.

stack	input	action
(S+E	+(3+4))+5	*reduce* $S \to S+E$
(*S*	+(3+4))+5	

Shift-reduce parsing

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

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→num
(*E*+2+(3+4))+5 ←	(E	+2+(3+4))+5	*reduce* S → 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→num
(S+*E*+(3+4))+5 ←	(S+E	+(3+4))+5	*reduce* S→ 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→num

Problem

- How do we know which action to take -- whether to shift or reduce, and which production?

- Sometimes **can** reduce but **shouldn't.**
  - e.g., $X \to \varepsilon$ can *always* be reduced
- Sometimes can reduce in more than one way.

Action Selection Problem

- Given stack σ and look-ahead symbol b, should parser:
  - **shift** b onto the stack (making it σb)
  - **reduce** some production $X \to \gamma$ assuming that stack has the form $\alpha\,\gamma$ (making it αX)
- If stack has form $\alpha\,\gamma$, should apply reduction $X \to \gamma$ (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 stacks σ (and 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

state

item

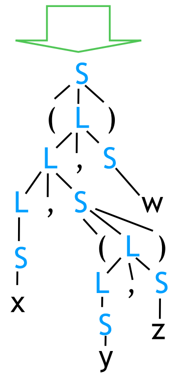
$$\boxed{\begin{array}{l} E \to \textbf{num}\ . \\ E \to (\ .\ S\ ) \end{array}}$$

- 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

An LR(0) grammar: non-empty lists

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

(x,(y,z),w)

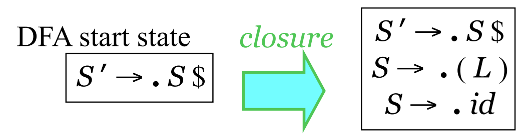


x (x,y) (x, (y,z), w)

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

Start State & Closure

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

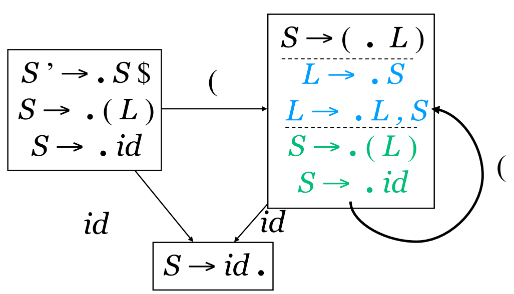


Constructing a DFA to read stack:

- 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

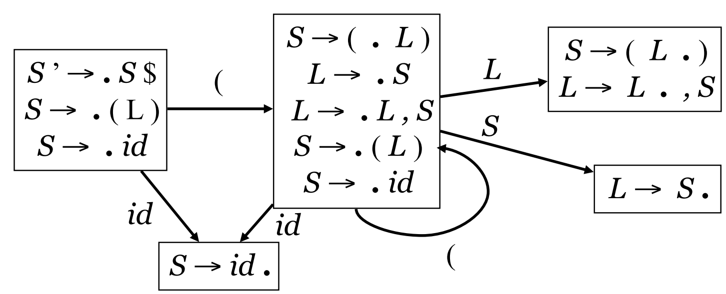
Applying terminal symbols

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



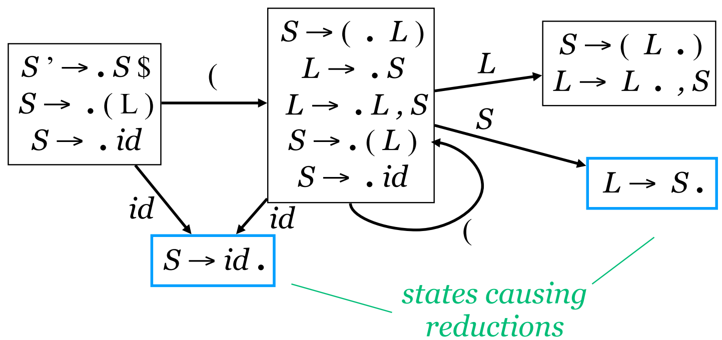
In new state, include all items that have appropriate input symbol just after dot, advance dot in those items, *and take closure.*

Applying non-terminals



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

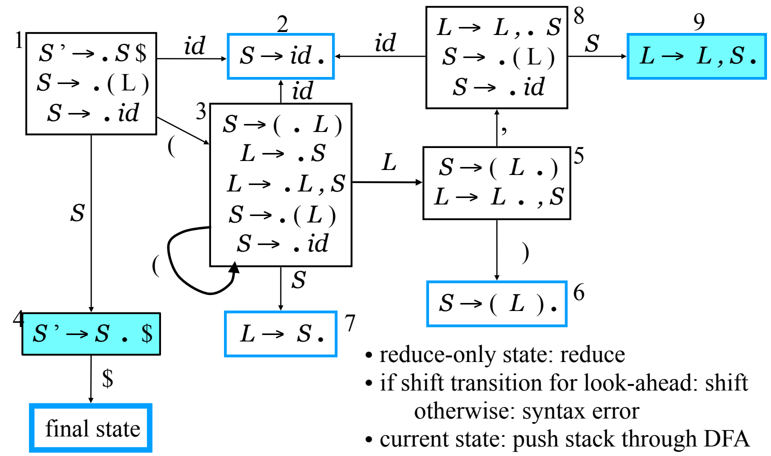
Applying reduce actions



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

Full DFA (Appel)

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



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

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

Parsing example: ((x),y)

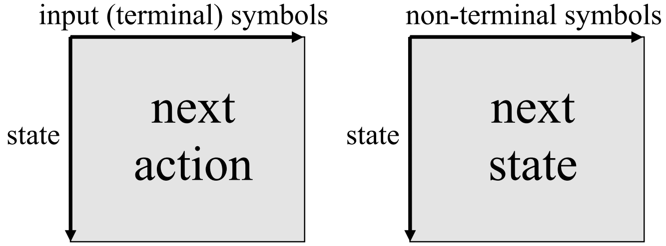
derivation	*stack*	*input*	*action*
((x),y) ←	$_1$	((x),y)	shift, goto 3
((x),y) ←	$_1(_3$	(x),y)	shift, goto 3
((x),y) ←	$_1(_3(_3$	x),y)	shift, goto 2
((x),y) ←	$_1(_3(_3x_2$	),y)	reduce $S \to id$
((**S**),y) ←	$_1(_3(_3S_7$	),y)	reduce $L \to S$
((**L**),y) ←	$_1(_3(_3L_5$	),y)	shift, goto 6
((L),y)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

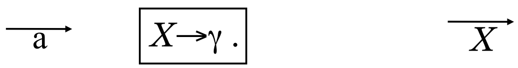
$((L)$	, y)	state = 6
$(S$	, y)	state = ?

Implementation: LR parsing table

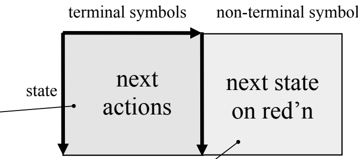


Action table
Used at every step to decide whether to shift or reduce

Goto table
Used only when reducing, to determine next state



Shift-reduce parsing table



action 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 go to 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
- Choose based on lookahead.

ok

$$L \to L \, , S \, .$$

shift /reduce

$$L \to L \, , S \, . \qquad S \to S \, . \, , L$$

reduce / reduce

$$L \to S \, , L \, . \qquad L \to S \, .$$

List grammar parsing 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	$L \to L,S$	$L \to L,S$	$L \to L,S$	$L \to L,S$	$L \to L,S$		

An LR(0) grammar?

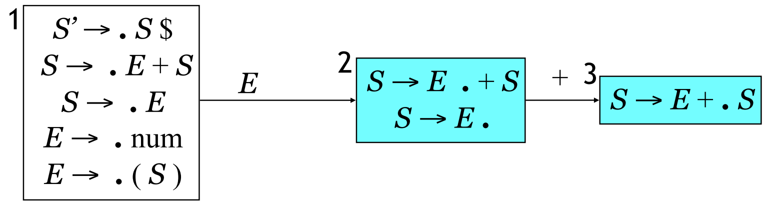
$$S \to S + E \mid E$$
$$E \to \text{num} \mid (S)$$

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

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

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/$S \rightarrow E$	$S \rightarrow E$	

SLR grammars

- Idea: Only add reduce action to table if lookahead 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	$S \rightarrow E$	