CS 4120 Introduction to Compilers

Ross Tate
Cornell University

Lecture 6: Bottom-Up Parsing

Administrivia

- Problem Set 2 out
  - Due in a week
- Programming Assignment 2 out
  - Due in two Mondays
- Mechanical Bull Riding
  - Next Wednesday

Bottom-up parsing

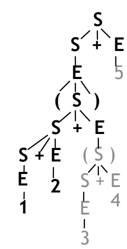
- A more powerful parsing technology
- LR grammars -- more expressive than LL
  - can handle left-recursive grammars, virtually all programming languages
  - Easier to express programming-language syntax
- Shift-reduce parsers
  - construct right-most derivation of program
  - automatic parser generators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

Top-down parsing

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

S → S + E | E
E → n | (S)

- *S* → *S*+*E* → *E*+*E* → (*S*)+*E* → (*S*+*E*)+*E* → (*S*+*E*+*E*)+*E* → (*E*+*E*+*E*)+*E* → (1+*E*+*E*)+*E* → (1+2+*E*)+*E* ...



- 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
  - Start with the tokens
  - End with the start symbol

S → S + E | E
E → n | (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

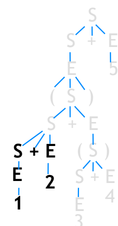
(1+2+(3+4))+5 ←		(1+2+(3+4))+5
(*E*+2+(3+4))+5 ←	(1	+2+(3+4))+5
(*S*+2+(3+4))+5 ←	(1	+2+(3+4))+5
(S+*E*+(3+4))+5 ←	(1+2	+(3+4))+5
(*S*+(3+4))+5 ←	(1+2+(3	+4))+5
(S+(*E*+4))+5 ←	(1+2+(3	+4))+5
(S+(*S*+4))+5 ←	(1+2+(3	+4))+5
(S+(S+*E*))+5 ←	(1+2+(3+4	))+5
(S+(*S*))+5 ←	(1+2+(3+4	))+5
(S+*E*)+5 ←	(1+2+(3+4)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

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

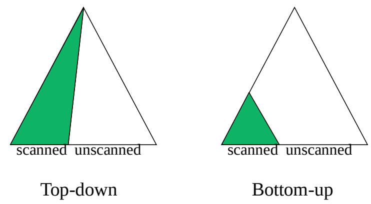
- (1+2+(3+4))+5 ←
  (*E*+2+(3+4))+5 ←
  (*S*+2+(3+4))+5 ←
  (S+*E*+(3+4))+5 …

- **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



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

Shift-reduce parsing

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

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* $E \to n$

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)

Parser States

- Goal: know which reductions are legal at any given point.
- Idea: summarize all possible stacks σ 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)

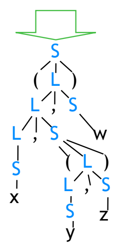
LR(0) parser

- **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...

An LR(0) grammar: non-empty lists

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

(x,(y,z),w)

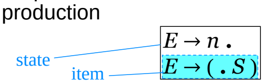


x (x,y) (x, (y,z), w)

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

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



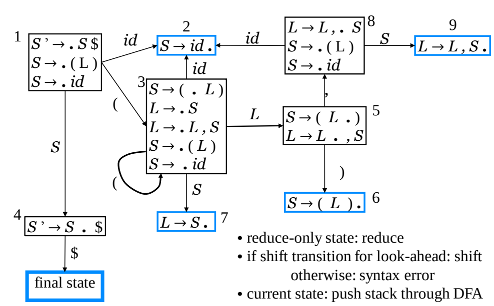
- Stuff before "**.**" is already on stack (beginnings of possible γ's to be reduced)
- Stuff after "**.**" : what we might see next

DFA-ish

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

Full DFA

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



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

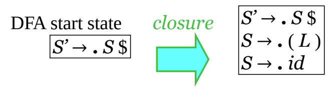
Parsing example: ((x),y)

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

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) ←	$_1 (_3 (_3 x_2$	),y)	reduce $S \rightarrow id$
((**S**),y) ←	$_1 (_3 (_3$	*S*),y)	shift, goto 7
((**S**),y) ←	$_1 (_3 (_3 S_7$	),y)	reduce $L \rightarrow S$
((**L**),y) ←	$_1 (_3 (_3$	*L*),y)	shift, goto 5
((**L**),y) ←	$_1 (_3 (_3 L_5$	),y)	shift, goto 6
((*L*),y) ←	$_1 (_3 (_3 L_5)_6$	,y)	reduce $S \rightarrow (L)$
(**S**,y) ←	$_1 (_3$	*S*,y)	shift, goto 7
(**S**,y) ←	$_1 (_3 S_7$	,y)	reduce $L \rightarrow S$
(**L**,y) ←	$_1 (_3$	*L*,y)	shift, goto 5
(**L**,y) ←	$_1 (_3 L_5$	,y)	shift, goto 8
(*L*,y) ←	$_1 (_3 L_5 ,_8$	y)	shift, goto 9
(*L*,y) ←	$_1 (_3 L_5 ,_8 y_2$	)	reduce $S \rightarrow id$
(*L*,**S**) ←	$_1 (_3 L_5 ,_8$	*S*)	shift, goto 9
(*L*,**S**) ←	$_1 (_3 L_5 ,_8 S_9$	)	reduce $L \rightarrow L , S$
(**L**) ←	$_1 (_3$	*L*)	shift, goto 5
(**L**) ←	$_1 (_3 L_5$	)	shift, goto 6
(**L**) ←	$_1 (_3 L_5)_6$		reduce $S \rightarrow (L)$
S	$_1$	*S*	shift, goto 4
S	$_1 S_4$	$	*done*

Start State & Closure

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

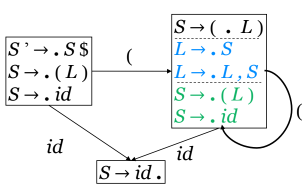


Constructing a DFA to read stack:

- First step: augment grammar with production $S' \rightarrow S\ \$$
- 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

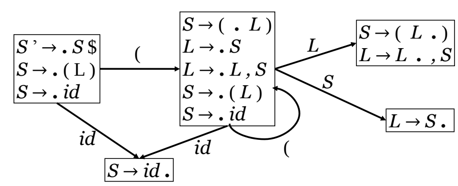
Applying terminal symbols

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



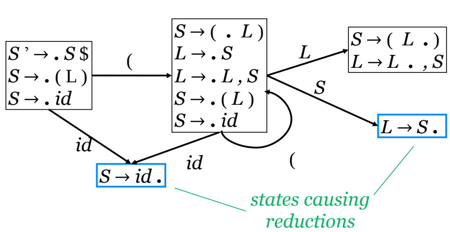
In new state, include all items that have appropriate input symbol just after dot, advance dot in those items, *and take closure.*

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))

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	shift /reduce	reduce / reduce
$L \rightarrow L , S\ .$	$L \rightarrow L , S\ .$ $S \rightarrow S\ .\ , L$	$L \rightarrow S , L\ .$ $L \rightarrow S\ .$

An LR(0) grammar?

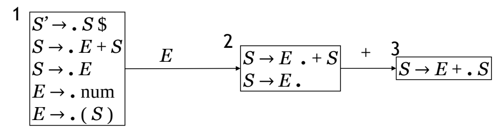
$S \rightarrow S + E \mid E$
$E \rightarrow n \mid (S)$

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

$S \rightarrow E + S \mid E$
$E \rightarrow n \mid (S)$

LR(0) construction

$S \rightarrow E + S \mid E$
$E \rightarrow n \mid (S)$



What to do in state 2?

	+	$	E
1			s2
2	s3/S→E	S→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→E	S→E	