CS412/413

Introduction to
Compilers and Translators
Spring '00

Lecture 5: Bottom-up parsing

Outline

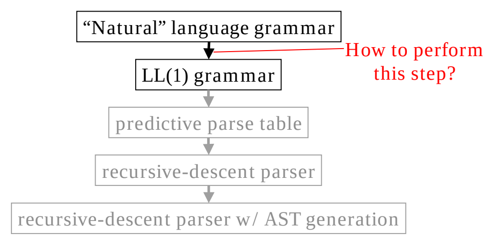
- More tips for LL(1) grammars
- Bottom-up parsing
- LR(0) parser construction

Administrivia

- Programming Assignment 1 due next class (Friday)
  - should be well under way -- leave time for testing, documentation
  - do not need to construct DFA!
- All group assignments should have settled out
- Homework 2 due next Friday
- Reading: finish Chapter 3 of Appel

Review

- Can make recursive descent parsers for LL(1) grammars



Grammars

- Have been using grammar for language of "sums with parentheses" (1+(3+4))+5
- Simple grammar w/ left associativity:

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

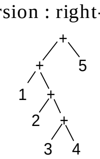
- LL(1) grammar for same language:

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

Left vs. Right Recursion

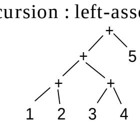
Right recursion : right-associative

$S \rightarrow E\, S'$
$S' \rightarrow \varepsilon \mid + S$ $\approx$ $S \rightarrow E + S$
$S \rightarrow E$



Left recursion : left-associative

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



$S \to S + E$
$S \to E$

Left-recursive vs Right-recursive

- Left-recursive grammars don't work with top-down parsing: arbitrary amount of look-ahead needed

derived string	lookahead	read/unread	
S	1		1 + 2 + 3 + 4
S + E	1		1 + 2 + 3 + 4
S + E + E	1		1 + 2 + 3 + 4
S + E + E + E	1		1 + 2 + 3 + 4
E + E + E + E	1		1 + 2 + 3 + 4
1 + ***E*** + E + E	2	1 +	2 + 3 + 4
1 + 2 + ***E*** + E	3	1 + 2 +	3 + 4
1 + 2 + 3 + ***E***	4	1 + 2 + 3 +	4
1 + 2 + 3 + 4	$	1 + 2 + 3 + 4	

How to create an LL(1) grammar

- Write a right-recursive grammar

$$S \to E + S$$
$$S \to E$$

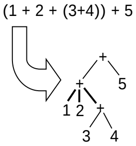
- *Left-factor* common prefixes, place suffix in new non-terminal

$$S \to E\ S'$$
$$S' \to \varepsilon$$
$$S' \to +\ S$$

EBNF

- **E**xtended **B**ackus-**N**aur **F**orm: allows some regular expression syntax on RHS
  - *, +, *( )*, *?* operators (Iota spec: *? = [ ]*)
  - BNF:| operator at top level

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



$$S \to E\ S'$$
$$S' \to \varepsilon \mid +\ S$$

$$S \to E\ (+\ E)^*$$

- EBNF version: no position on + associativity

Top-down parsing EBNF

- Recursive-descent code can directly implement the EBNF grammar:

$$S \to E\ (+\ E)^*$$

```
void parse_S () { // parses sequence of E + E + E ...
   parse_E ();
   while (true) {
        switch (token) {
           case '+': token = input.read(); parse_E();
                     break;
           case ')': case EOF: return;
           default: throw new ParseError();
   }
}
```


Building a left-associative AST

```
Expr parse_S() {
   Expr result = parse_E();
   while (true) {
    switch (token) {
          case '+': token = input.read();
                    result = new Add(result, parse_E());
                    break;
      case ')': case EOF: return result;
      default: throw new ParseError();
   }
}
```


Summary

- Now have complete recipe for building a parser

Language grammar
↓
LL(1) grammar
↓
predictive parse table
↓
recursive-descent parser
↓
recursive-descent parser w/ AST generation

Bottom-up parsing

- A more powerful parsing technology
- LR grammars -- more expressive than LL
  - can handle left-recursive grammars, virtually all programming languages
  - More natural expression of programming language syntax
- Shift-reduce parsers
  - automatic parser generators (*e.g.* yacc,CUP)
  - detect errors as soon as possible
  - allows better error recovery

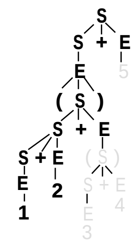
Top-down parsing

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

$S \to S \mid E$... grammar: $S \to S + E \mid E$, $E \to$ number | (S)

$S \to S+E \to E+E \to (S)+E \to (S+E)+E \to (S+E+E)+E \to (E+E+E)+E \to (1+E+E)+E \to (1+2+E)+E$ ...

- In left-most derivation, entire tree above a token (2) has been 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

$S \to S + E \mid E$
$E \to$ number | (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 (read upward)

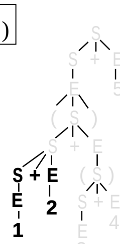
(1+2+(3+4))+5 ←		(1+2+(3+4))+5
(**E**+2+(3+4))+5 ←	(1	+2+(3+4))+5
(**S**+2+(3+4))+5 ←	(1	+2+(3+4))+5
(S+**E**+(3+4))+5 ←	(1+2	+(3+4))+5
(**S**+(3+4))+5 ←	(1+2+(3	+4))+5
(S+(**E**+4))+5 ←	(1+2+(3	+4))+5
(S+(**S**+4))+5 ←	(1+2+(3	+4))+5
(S+(S+**E**))+5 ←	(1+2+(3+4	))+5
(S+(**S**))+5 ←	(1+2+(3+4	))+5
(S+**E**)+5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

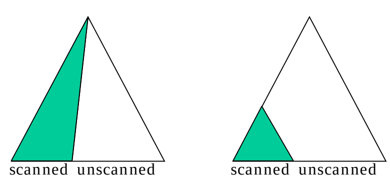
$S \to S + E \mid E$
$E \to$ number | (S)

- Advantage of bottom-up parsing: can select productions based on 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 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 $X \to \gamma$ (pop γ, push X)

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

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

Shift-reduce parsing

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 different ways

Action Selection Problem

- Given stack σ and look-ahead symbol b, should we
  - **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 $X \to \gamma$ depending on what stack prefix α is -- but α is different for different possible reductions, since γ's have different length. How to keep track?

Parser States

- Goal: know what reductions are legal at any given point
- Idea: summarize all possible stack prefixes α as a parser *state*
- Parser state is defined by a DFA that reads in the stack α
- 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 (including this one)
- But will help us understand how to build better parsers

An LR(0) grammar: non-empty lists

$S \rightarrow (L)$

$S \rightarrow id$

$L \rightarrow S$

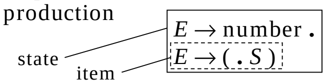
$L \rightarrow L , S$

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

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

LR(0) states

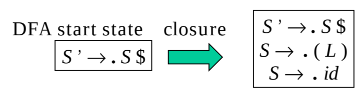
- A state is a set of *items*
- An *LR(0) item* is a production from the language with a separator "**.**" somewhere in the RHS of the production



- Stuff before "**.**" 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

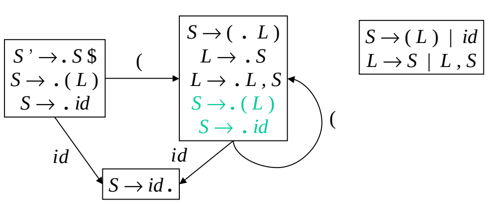
Start State & Closure

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



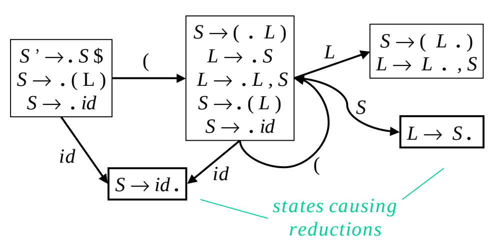
- First step: augment grammar with prod'n $S' \rightarrow S\ \$$
- Start state of DFA: empty stack = $S' \rightarrow .\ S\ \$$
- ***Closure*** of a state adds items for all productions whose LHS occurs in an item in the state, just after "**.**"
  - set of possible productions to be reduced next
  -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

Applying symbols



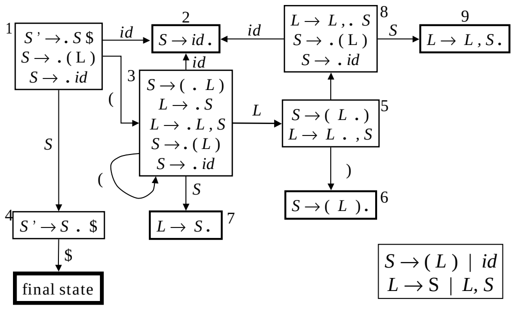
In new state, include all items that have appropriate input symbol just after dot, and advance dot in those items (*and take closure.*)

Applying reduce actions



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

Full DFA (Appel p. 63)



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

- Optimization: stack is labeled w/state
- Let's try parsing ((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 $S \rightarrow id$
((***S***),y) ←	${}_1({}_3({}_3S_7),y)$		reduce $L \rightarrow S$
((***L***),y) ←	${}_1({}_3({}_3L_5$	),y)	shift, goto 6
((*L*),y) ←	${}_1({}_3({}_3L_5)_6$	,y)	reduce $S \rightarrow (L)$
(***S***,y) ←	${}_1({}_3S_7$	,y)	reduce $L \rightarrow S$
(***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 $S \rightarrow id$
(*L*,***S***) ←	${}_1({}_3L_5,{}_8S_9$	)	reduce $L \rightarrow L , S$
(***L***) ←	${}_1({}_3L_5$	)	shift, goto 6
(***L***) ←	${}_1({}_3L_5)_6$		reduce $S \rightarrow (L)$
S	${}_1S_4$	$	*done*

Bottom-up parsing

- Grammars can be parsed bottom-up using a DFA + stack
- State construction converts grammar into states that capture information needed to know what action to take
- *Next time:* shift-reduce parsing tables SLR, LR(1) parsers, automatic parser generators