CS 412
Introduction to Compilers

Andrew Myers
Cornell University

Lecture 6: AST construction and bottom-up parsing

5 Feb 01

Administrivia

- Programming Assignment 1 due on Wednesday
- Check class newsgroup cornell.class.cs412 for answers to frequently asked questions

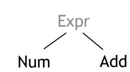
Completing the parser

Now we know how to construct a recursive-descent parser for an LL(1) grammar.

Can we use recursive descent to build an abstract syntax tree too?

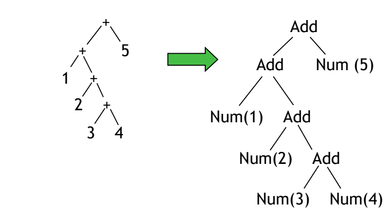
Creating the AST

```
abstract class Expr { }

class Add extends Expr {
  Expr left, right;
  Add(Expr L, Expr R) { left = L; right = R; }
}

class Num extends Expr {
  int value;
  Num (int v) { value = v);
}
```



AST Representation

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



How can we generate this structure during recursive-descent parsing?

Creating the AST

- Just add code to each parsing routine to create the appropriate nodes!
- Works because parse tree and call tree have same shape
- parse_S, parse_S', parse_E all return an Expr:
  - void parse_E() ⇒ Expr parse_E()
  - void parse_S() ⇒ Expr parse_S()
  - void parse_S'() ⇒ Expr parse_S'()

AST creation code

```
Expr parse_E() {
    switch(token) {
      case num: // E → number
            Expr result = Num (token.value);
            token = input.read(); return result;
      case '(': // E → ( S )
            token = input.read();
            Expr result = parse_S();
            if (token != ')') throw new ParseError();
            token = input.read(); return result;
      default: throw new ParseError();
    }
}
```


parse_S

$S \rightarrow E\,S'$
$S' \rightarrow \varepsilon \mid +S$
$E \rightarrow \textbf{num} \mid (\,S\,)$

```
Expr parse_S() {
   switch (token) {
        case num:
        case '(':
            Expr left = parse_E();
            Expr right = parse_S'();
            if (right == null) return left;
            else return new Add(left, right);
        default: throw new ParseError();
   }
}
```


Or…an Interpreter!

$S \rightarrow E\,S'$
$S' \rightarrow \varepsilon \mid +S$
$E \rightarrow \textbf{num} \mid (\,S\,)$

```
int parse_E() {
      switch(token) {
            case number:
                  int result = token.value;
                  token = input.read(); return result;
            case '(':
                  token = input.read();
                  int result = parse_S();
                  if (token != ')') throw new ParseError();
                  token = input.read(); return result;
                  default: throw new ParseError(); }}

int parse_S() {
      switch (token) {
            case number:
            case '(':
               int left = parse_E();
               int right = parse_S'();
               if (right == 0) return left;
               else return left + right;
            default: throw new ParseError(); } }
```


Grammars

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

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

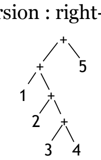
- LL(1) grammar for same language:

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

Left vs. Right Recursion

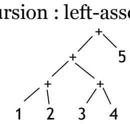
Right recursion : right-associative

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



Left recursion : left-associative

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



Left-recursive vs Right-recursive

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

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

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

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

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

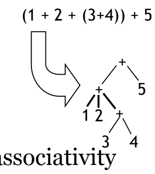
$$S \rightarrow E\ S'$$
$$S' \rightarrow \varepsilon$$
$$S' \rightarrow +\ S$$

EBNF

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

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

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



- EBNF version: no position on + associativity

Top-down parsing EBNF

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

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

```
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();
   }
}
```


Reassociating the 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

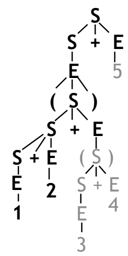
- A more powerful parsing technology
- LR grammars -- more expressive than LL
  - can handle left-recursive grammars, virtually all programming languages
  - Easier to express programming language syntax
- Shift-reduce parsers
  - construct right-most derivation of program
  - automatic parser generators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

Top-down parsing

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

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

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



- 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

$S \rightarrow S + E \mid E$
$E \rightarrow$ 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 ↑

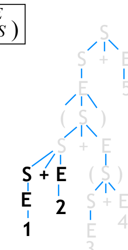
(1+2+(3+4))+5 ←		(1+2+(3+4))+5
(*E*+2+(3+4))+5 ←	(1	+2+(3+4))+5
(*S*+2+(3+4))+5 ←	(1	+2+(3+4))+5
(S+*E*+(3+4))+5 ←	(1+2	+(3+4))+5
(*S*+(3+4))+5 ←	(1+2+(3	+4))+5
(S+(*E*+4))+5 ←	(1+2+(3	+4))+5
(S+(*S*+4))+5 ←	(1+2+(3	+4))+5
(S+(S+*E*))+5 ←	(1+2+(3+4	))+5
(S+(*S*))+5 ←	(1+2+(3+4	))+5
(S+*E*)+5 ←	(1+2+(3+4)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

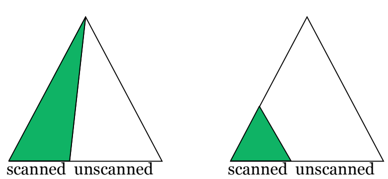
$S \rightarrow S + E \mid E$
$E \rightarrow$ number | (S)

- **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 $X \rightarrow \gamma$ (pop γ, push X)

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

Shift-reduce parsing

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

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., $X \rightarrow \varepsilon$ can *always* be reduced
- Sometimes can reduce in different ways

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 α γ (making it α*X*)
- If stack has form α γ, 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 α
- Accept states of DFA: unique reduction!
- Summarizing discards information
  - affects what grammars parser handles
  - affects size of DFA (number of states)