# CS412/413

Introduction to Compilers
Radu Rugina

Lecture 7: AST Construction and
Bottom-up Parsing
3 Feb 03

## Top-Down Parsing

- Now we know:
  - how to build a parsing table for an LL(1) grammar (use FIRST/FOLLOW sets)
  - how to construct a recursive-descent parser from the parsing table
- Can we use recursive descent to build an abstract syntax tree too?



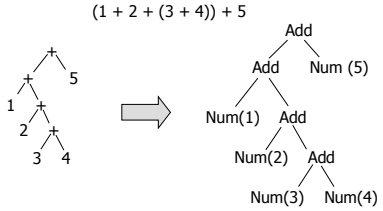
## Creating the AST

```
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); }
}
```

Class Hierarchy

## 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: parse_E

```
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();
    }
}
```

## AST Creation: parse_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();
   }
}
```

S → E S'
S' → ε | + S
E → **num** | ( S )



## Or…an Interpreter!

```
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(); } }
```

S → E S '
S ' → ε | + S
E → **num** | ( S )



## Grammars

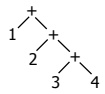
- Have been using grammar for language of "sums with parentheses" e.g., (1+(3+4))+5
- Started with simple, right-associative grammar:
  S → E + S | E
  E → num | ( S )
- Transformed it to an LL(1) grammar by left-factoring:
  S → ES'
  S' → ε | + S
  E → number | ( S )
- What if we start with a left-associative grammar?
  S → S + E | E
  E → num | ( S )



## Left vs. Right Associativity

Right recursion : right-associative

S → E + S
S → E
E → num




Left recursion : left-associative

S → S + E
S → E
E → num

## Left Recursion

- Left-recursive grammars don't work with top-down parsing: we don't know where to stop the recursion

| 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 |



## Left-Recursive Grammars

- Left-recursive grammars are not LL(1) !

  S → S α
  S → β

- FIRST(β) ⊆ FIRST(Sα)
- Both productions will appear in the predictive table, at row S in all the columns corresponding to symbols in FIRST(β)

## Eliminate Left Recursion

- Method for left-recursion elimination:
  Replace
  $X \rightarrow X\ \alpha_1 \mid \ldots \mid X\ \alpha_m$
  $X \rightarrow \beta_1 \mid \ldots \mid \beta_n$
  with
  $X \rightarrow \beta_1\ X' \mid \ldots \mid \beta_n\ X'$
  $X' \rightarrow \alpha_1\ X' \mid \ldots \mid \alpha_m\ X' \mid \varepsilon$
- (See the complete algorithm in the Dragon Book)



## Creating an LL(1) Grammar

- Start with a left-recursive grammar:
  $S \rightarrow S + E$
  $S \rightarrow E$
  and apply left-recursion elimination algorithm:
  $S \rightarrow E\ S'$
  $S' \rightarrow +E\ S' \mid \varepsilon$
- Start with a right-recursive grammar:
  $S \rightarrow E + S$
  $S \rightarrow E$
  and apply left-factoring to eliminate common prefixes:
  $S \rightarrow E\ S'$
  $S' \rightarrow +\ S \mid \varepsilon$



## EBNF

- Extended Backus-Naur Form = a form of specifying grammars which allows some regular expression syntax on RHS
  – *, +, ( ), ? operators (also [X] means X?)

$S \rightarrow E\ S'$
$S' \rightarrow \varepsilon \mid +\ S$ ⇒ $S \rightarrow E\ (\ +\ E\ )^*$

- 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();
   }
}
```

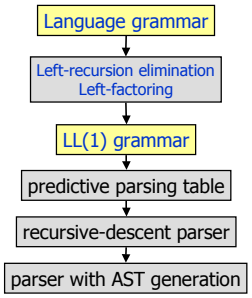


## Reassociating the AST

```
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();
     }
   }
```



## Top-Down Parsing Summary



## Next: Bottom-up Parsing

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



## Bottom-up Parsing

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

S → S + E | E
E → num | ( 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 |

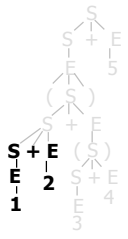


## Bottom-up Parsing

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

S → S + E | E
E → num | ( S )

- Advantage of bottom-up parsing: can postpone the selection of productions until more of the input is scanned





## Top-down Parsing

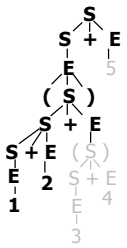
(1+2+(3+4))+5

S → num | ( S )

S → S + E | E

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 …

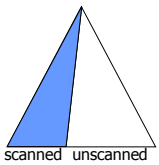
- In left-most derivation, entire tree above a token (2) has been expanded when encountered



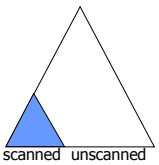


## Top-down vs. Bottom-up

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 actions: is a sequence of shift and reduce operations
- Parser state: a stack of terminals and non-terminals (grows to the right)
- Current derivation step = 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 $\gamma$ from top of stack with non-terminal symbol X, corresponding to production $X \rightarrow \gamma$ (pop $\gamma$, push X)

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



## Shift-reduce Parsing

S → S + E | E
E → num | ( 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?
- Issues:
  - Sometimes can reduce but shouldn't
  - Sometimes can reduce in different ways



## Action Selection Problem

- Given stack $\sigma$ and look-ahead symbol b, should parser:
  - shift b onto the stack (making it $\sigma$b)
  - reduce $X \rightarrow \gamma$ assuming that stack has the form $\alpha\gamma$ (making it $\alpha X$)
- If stack has form $\alpha\gamma$, should apply reduction $X \rightarrow \gamma$ (or shift) depending on stack prefix $\alpha$
  - $\alpha$ is different for different possible reductions, since $\gamma$'s have different length.