Today's Topics

Last Time

- Context-free grammars - BNF, terminal vs nonterminal symbols, relation to S/SL, sentential forms, derivations, parse trees, ambiguity

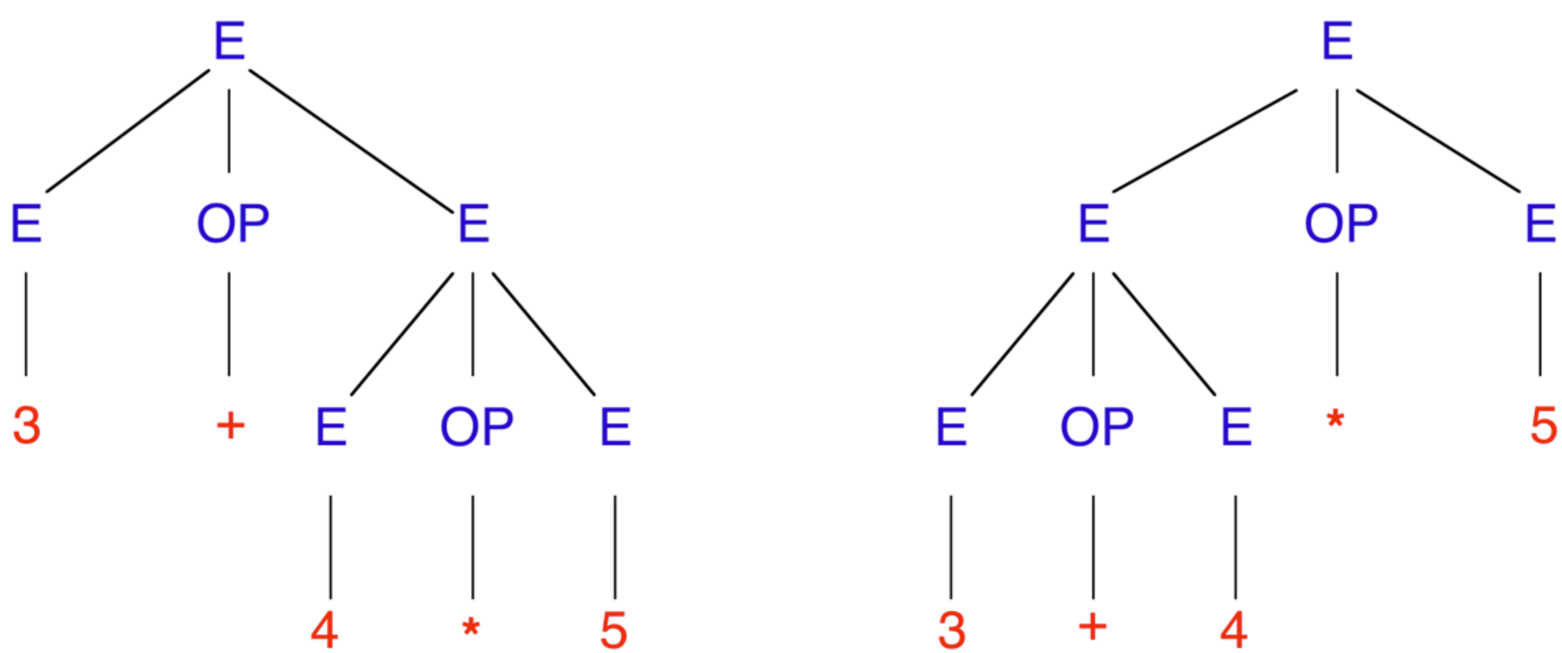
Today

- Resolving ambiguity using precedence and associativity

Ambiguity

Which Parse Tree?

- An *ambiguous* grammar is one in which there is more than one leftmost derivation (i.e. more than one parse tree) for some sentence of the language
- Our example grammar G is ambiguous because it allows more than one parse tree / leftmost derivation of the sentence 3 + 4 * 5
- Ambiguity also causes problems for deterministic parsing - which parse tree is the right one?

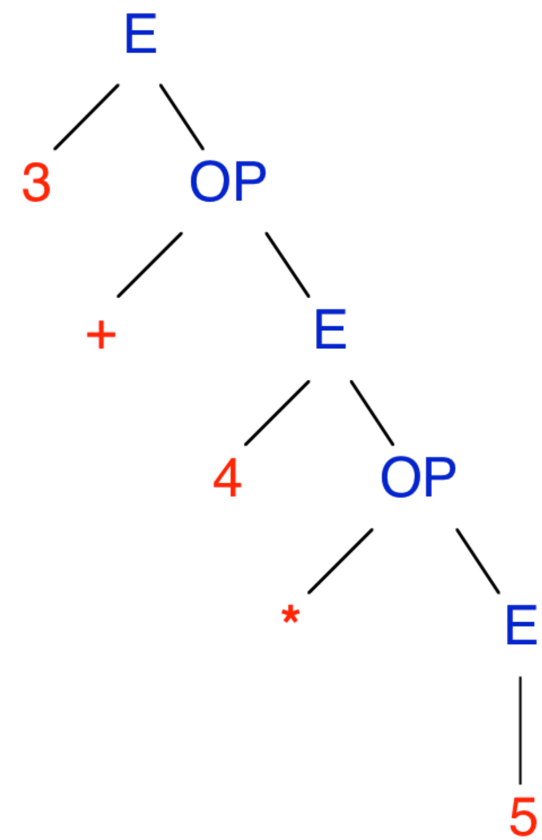


Parse Trees & Ambiguity in S/SL

- SL "grammars" are *never* ambiguous, because there is only one path through an SL program for each input symbol

```
E:
    [
        | '-':
           @E
        | '(':
           @E ')'
        | '0','1','2','3','4',
          '5','6','7','8','9':
    ]
    @OP;

OP:
    [
        | '=','-','*','/':
           @E
        | *:
    ];
```



Precedence

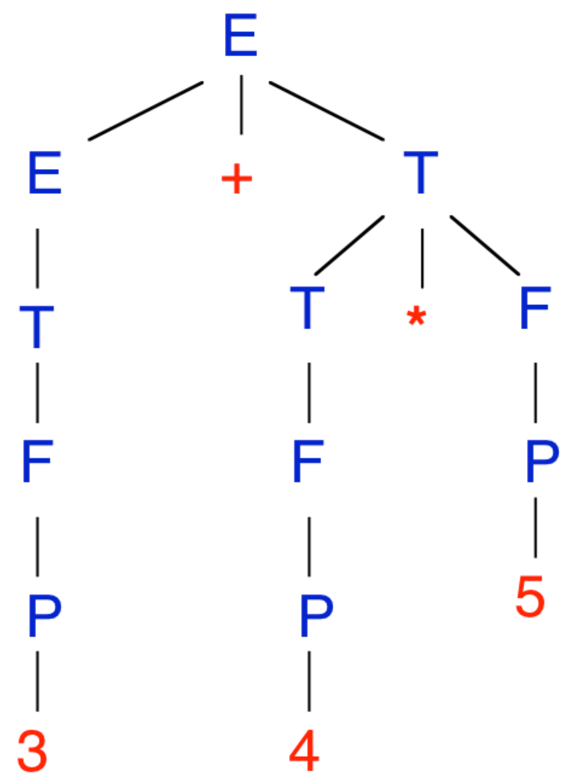
- *Precedence* is the most common way to remove ambiguity from a grammar
- Operators with *higher* precedence (more tightly *binding*) occur lower in the tree
- In BNF, one nonterminal is used for each *precedence level*

```
E  → E + T
   | E - T
   | T

T  → T * F
   | T / F
   | F

F  → -P
   | P

P  → (E)
   | 0|1|...|9
```



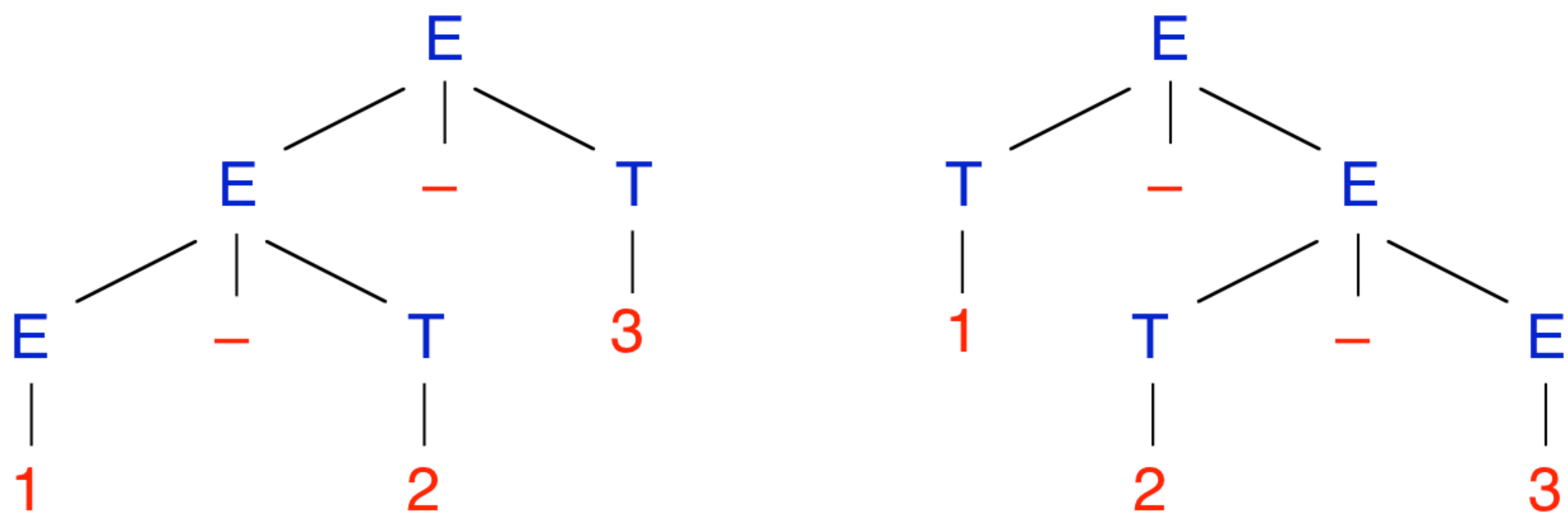
Associativity

- The structure of the grammar also specifies the *associativity* (grouping of sequences) of operators, another source of ambiguity

  $E \rightarrow E - T$ left associativity;
  $1 - 2 - 3$ groups as $(1 - 2) - 3$

  $E \rightarrow T - E$ right associativity;
  $1 - 2 - 3$ groups as $1 - (2 - 3)$

- Elements *lower* in the tree are evaluated before elements higher up in the tree



Precedence & Associativity in S/SL

- Precedence is specified in SL grammars using a rule for each level
- Left associativity is specified using iteration instead of recursion

```
E:
    @T
    {[
        |‘+’:
            @T
        |‘-’:
            @T
        |*:
            >
    ]};

T:
    @F
    {[
        |‘*’:
            @F
        |‘/’:
            @F
        |*:
            >
    ]};
```

```
F:
    [
        |‘-’:
            @P
        |*:
            @P
    ];

P:
    [
        |‘(’:
            @E  ‘)
        |‘0’,‘1’,‘2’,‘3’,‘4’,
         ‘5’,‘6’,‘7’,‘8’,‘9’:
    ];
```

Right Associativity in S/SL

- Right associativity is accomplished in SL using explicit recursion
- The assignment operator in C and Java is considered an expression (that returns the assigned value), and is right associative
- Example:

  `a = b = c = 5;` means `a = (b = (c = 5));`
  which sets all three of a, b and c to the value 5

```
AssignExpr:
    @LeftExpr
    [
        | '=':
            @AssignExpr
        | *:
    ];
```

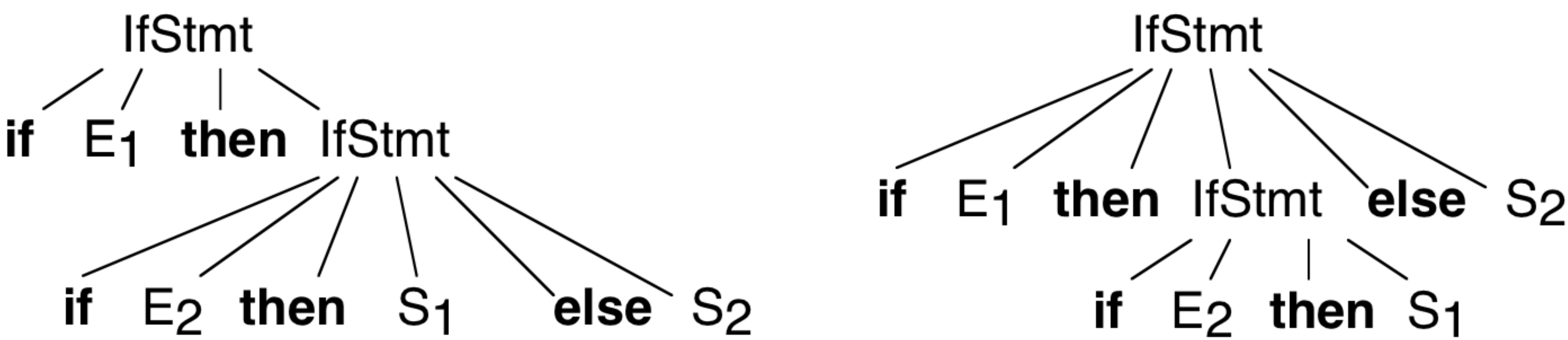
The If-Then-Else Ambiguity

- Many block-structured languages (e.g., C, Java, Pascal) have an ambiguity with nested **if** statements

```
IfStmt → if Expn then Stmt
       | if Expn then Stmt else Stmt
```

- So what does this mean?

if E_1 **then** **if** E_2 **then** S_1 **else** S_2



The If-Then-Else Ambiguity

- S/SL grammars are never ambiguous since they have a strict left-to-right evaluation order
- It's literally impossible to write an ambiguous S/SL grammar, because S/SL grammars are *ordered grammars* - nonterminals are always applied in the order they appear

```
IfStmt:
   @Expn 'then' @Stmt
   [
      | 'else':
        @Stmt
      | *:
   ];

Stmt:
   [
      | 'if':
        @IfStmt
      | ...
   ];
```

Resolving the If-Then-Else Ambiguity

- The if-then-else ambiguity can be resolved in various ways
- One way is to require that nested ifs be balanced, that is, change the grammar such that inner **if**s must have an **else** clause

```
IfStmt → if Expn then Stmt
       | if Expn then BalancedStmt else Stmt
```

  where BalancedStmt allows only if-then-else with balanced statements, and not if-then

  Effectively, this gives inner **else**s higher precedence than outer ones by forcing them down the parse tree

- Another way is to disallow directly nested if statements, requiring that they be inside blocks

```
IfStmt → if Expn then BlockStmt
       | if Expn then BlockStmt else BlockStmt

BlockStmt → { Stmt* }
```

  Of course, this changes the language ...

Resolving Ambiguities by Language Design

- Syntactic ambiguities are not just difficult for parsers - they are also difficult for people
- So some languages (e.g., Euclid, Ada, Turing, Ruby) are designed to address the if-then-else problem by language design - the if statement explicitly delimits its own boundaries, rather than leaving it to the parser
- In these languages the cases that are ambiguous in C, Java and Pascal look very different

```
ifStmt  →  if Expn then Stmt* end
        |  if Expn then Stmt* else Stmt* end
```

```
if E1 then                    if E1 then
   if E2 then                    if E2 then
      S1                            S1
   else                          end
      S2                      else
   end                           S2
end                           end
```

Summary

Ambiguity

- Resolving ambiguity in context-free grammars using precedence and associativity
- Levels of precedence, left and right associativity
- The if-then-else ambiguity

Next Week

- Quiz #1: Chapters 1-6 (Lectures 1-7) inclusive - 20 minutes

  Be on time!

  Covers: basic concepts and definitions, compiler structure and phases, S/SL language and usage, Scanners & Screeners, in S/SL

  Example Quiz #1 on course web site, solution on forum!

- Then: All about Parsing