中山大學 SUN YAT-SEN UNIVERSITY 计算机学院（软件学院） SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

Compilation Principle
编 译 原 理

第6讲：语法分析(3)

张献伟

xianweiz.github.io

DCS290, 3/14/2023

Review Questions

- Grammar G: *stmt* → **if** (*expr*) *stmt* **else** *stmt*
  | **while** (*expr*) *stmt* | *v*
  expr → true | false

  N = { *stmt expr* }

- Is **if** (true) *stmt* **else** *v* an sentence of grammar G?

  NO. It is a sentential form (句型), as *stmt* is non-terminal symbol.

- Is **while** (false) **else** *v* an sentence of G?

  NO. It cannot be derived using the production rules.

- Describe the languages generated by G: *list* → *list*, **id** | **id**?

  A list of one or more ids separated by commas.

- How does parse tree relate to derivation?

  Parse tree is a graphical representation of derivation. No orders kept.

Ambiguity[二义性]

- Grammar G is ambiguous if
  - It produces more than one parse tree for some sentence
  - i.e., there exist a string *str* ∈ L(G) such that
  - more than one parse tree derives *str*

    ≡ more than one leftmost derivation derives *str*

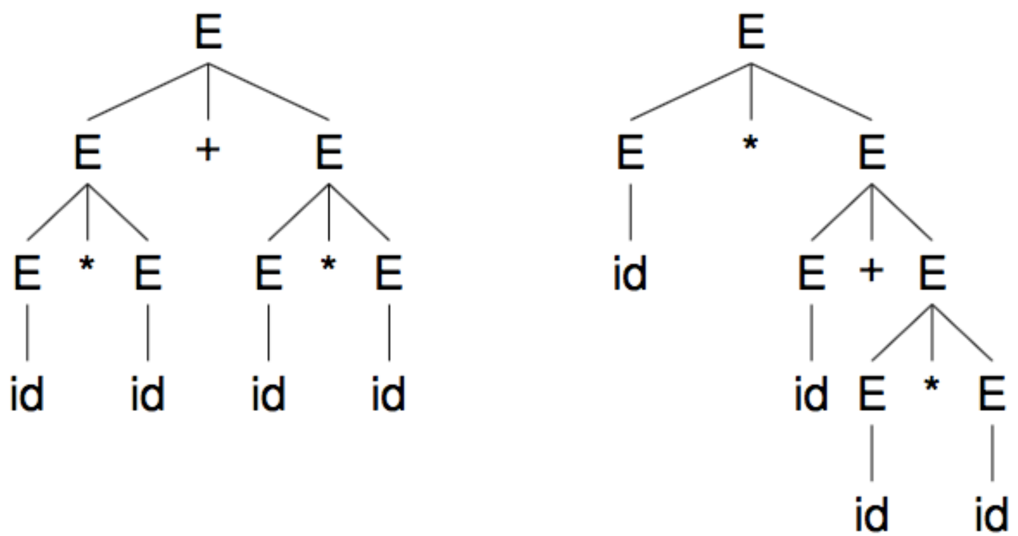
    ≡ more than one rightmost derivation derives *str*
- Unambiguous grammars are preferred for most parsers[文法最好没有歧义性]
  - Ambiguity of the grammar implies that at least some strings in its language have different structures (parse trees)
  - Thus, such a grammar is unlikely to be useful for a programming language, because two structures for the same string (program) implies two different meanings (executable equivalent programs) for this program

Ambiguity (cont.)

- Ambiguity is the property of the grammar, not the language
  - Just because G is ambiguous, does not mean L(G) is inherently ambiguous
  - A G′ can exist where G′ is unambiguous and L(G′) ≡ L(G)
- Impossible to convert ambiguous to unambiguous grammar automatically[歧义不能自动消除]
  - It is (often) possible to rewrite grammar to remove ambiguity
  - Or, use ambiguous grammar, along with disambiguating rules to "throw away" undesirable parse trees, leaving only one tree for each sentence (as in YACC)
    - A parse tree would be used subsequently for semantic analysis
    - Thus, more than one parse tree would imply several interpretations

Review Ambiguity Example

- Grammar *E→E*E | E+E | (E) | id* is ambiguous[二义的]
  - String id * id + id * id can result in 3 parse trees (and more)



The deepest sub-tree is traversed first, thus highest precedence

- Grammar can apply different rules to derive same string
  - Meaning of parse tree 1: (id * id) + (id * id)
  - Meaning of parse tree 2: id * (id + (id * id))
  - Meaning of parse tree 3: id * ((id + id) * id)

Remove Ambiguity[消除二义性]

- Specify precedence[指定优先级]
  - The higher level of the production, the lower priority of operator
  - The lower level of the production, the higher priority of operator
- Specify associativity[指定结合性]
  - If the operator is left associative, induce left recursion in its production
  - If the operator is right associative, induce right recursion in its production

$E \rightarrow E * E \mid E + E \mid (E) \mid \text{id}$

$$\begin{aligned} E &\rightarrow E + E \mid T \\ T &\rightarrow T * T \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

Possible to get id + (id + id) and (id + id) + id

// lowest precedence +
// middle precedence *
// highest precedence ()

$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \text{id} \end{aligned}$$

Now, can only have more '+' on left

// + is left-associative
// * is left-associative

Remove Ambiguity[消除二义性]

```
int a, b = 1, c = 2;
a = b = c;
b = a - b - c;

b = ???
```

- Specify precedence[指定优先级]
  - The higher level of the production, the lower priority of operator
  - The lower level of the production, the higher priority of operator
- Specify associativity[指定结合性]
  - If the operator is left associative, induce left recursion in its production
  - If the operator is right associative, induce right recursion in its production

$E \rightarrow E * E \mid E + E \mid (E) \mid \text{id}$

$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid \text{id}$

Possible to get id + (id + id) and (id + id) + id

// lowest precedence +
// middle precedence *
// highest precedence ()

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$

Now, can only have more '+' on left

// + is left-associative
// * is left-associative

Remove Ambiguity (cont.)

$E \rightarrow E * E \mid E + E \mid (E) \mid \text{id}$

$$
\begin{aligned}
E &\rightarrow E + E \mid T \\
T &\rightarrow T * T \mid F \\
F &\rightarrow (E) \mid \text{id}
\end{aligned}
\qquad \Rightarrow \qquad
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \text{id}
\end{aligned}
$$

- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id)
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)

SUN YAT-SEN UNIVERSITY

Remove Ambiguity (cont.)

$E \rightarrow E{*}E \mid E{+}E \mid (E) \mid \text{id}$

$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid \text{id}$

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$

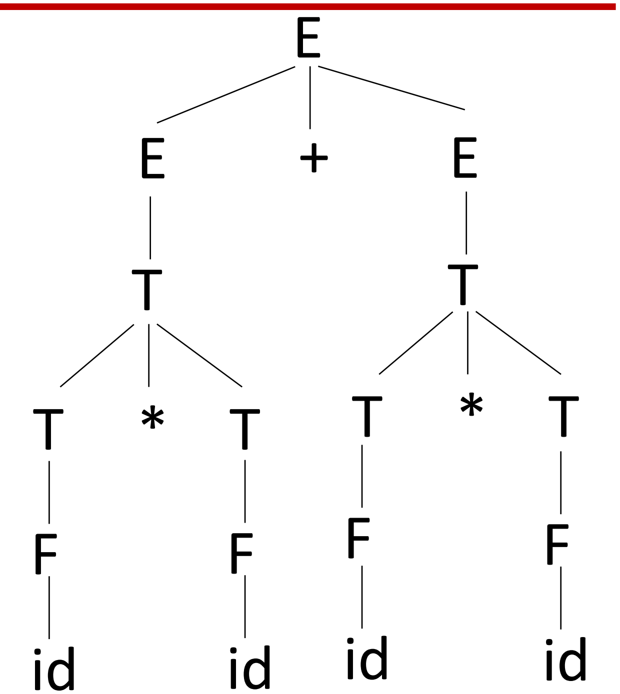
- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id)
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)

Remove Ambiguity (cont.)

$E \rightarrow E*E \mid E+E \mid (E) \mid \text{id}$

$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid \text{id}$

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$



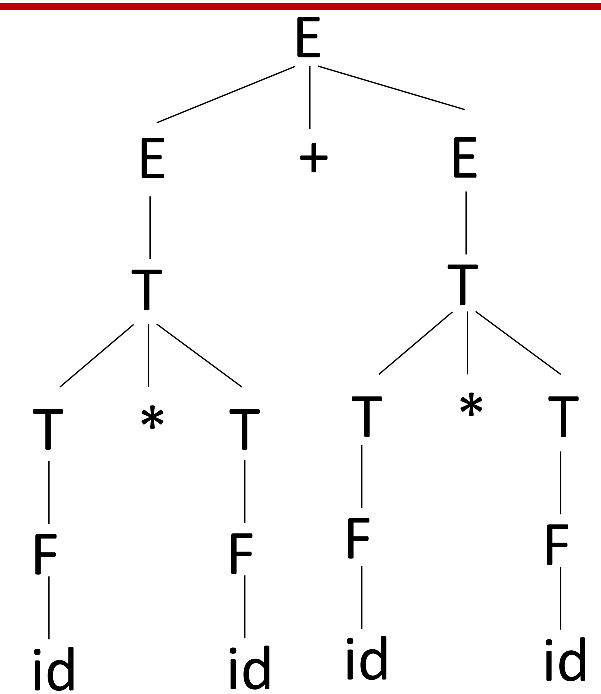
- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id)
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)

Remove Ambiguity (cont.)

$E \rightarrow E * E \mid E + E \mid (E) \mid \text{id}$

$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid \text{id}$

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$



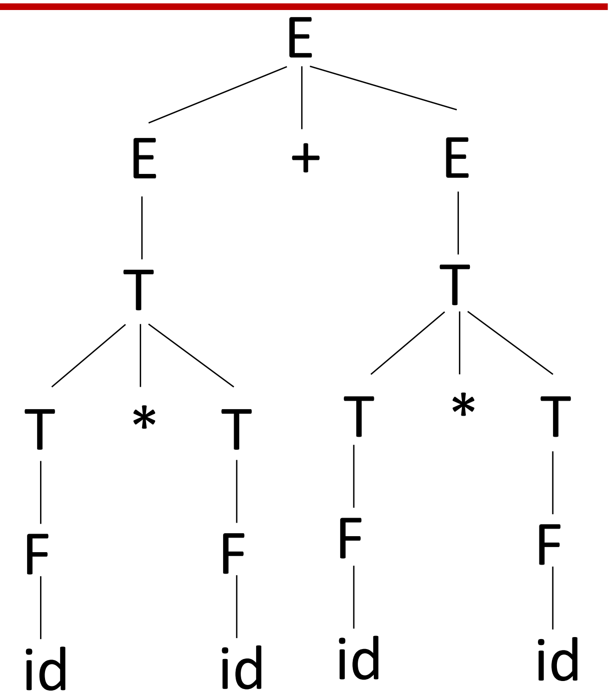
- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id) ✓
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)

Remove Ambiguity (cont.)

$E \rightarrow E * E \mid E + E \mid (E) \mid \text{id}$

$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid \text{id}$

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$



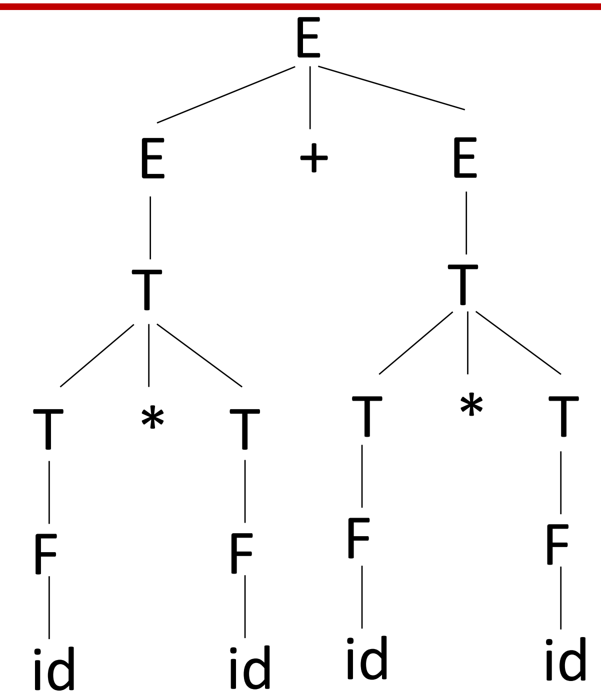
- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id) ✓
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)
- String id + id + id

Remove Ambiguity (cont.)

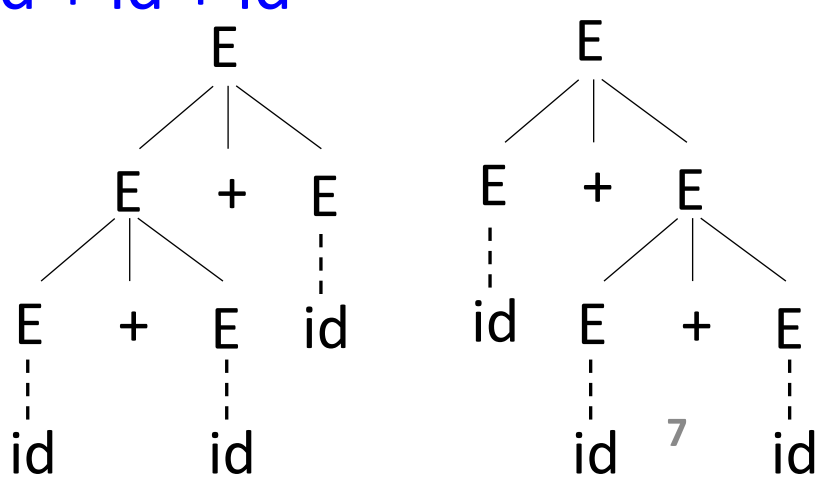
$E \rightarrow E*E \mid E+E \mid (E) \mid \text{id}$

$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid \text{id}$

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$



- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id) ✓
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)
- String id + id + id

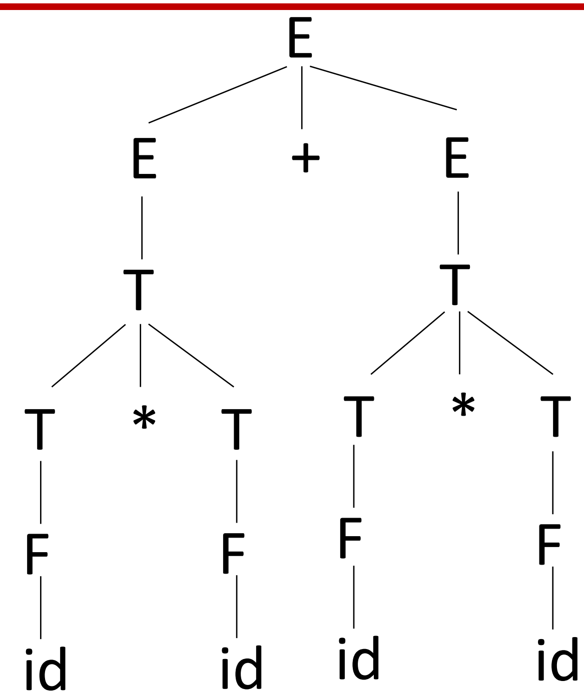


Remove Ambiguity (cont.)

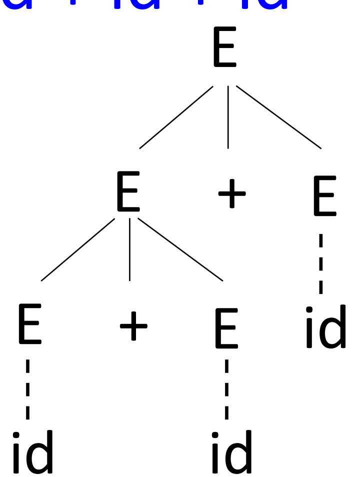
$E \rightarrow E*E \mid E+E \mid (E) \mid \text{id}$

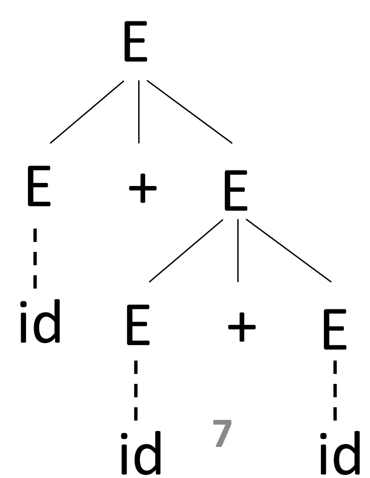
$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid \text{id}$

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \text{id}$



- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id) ✓
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)
- String id + id + id



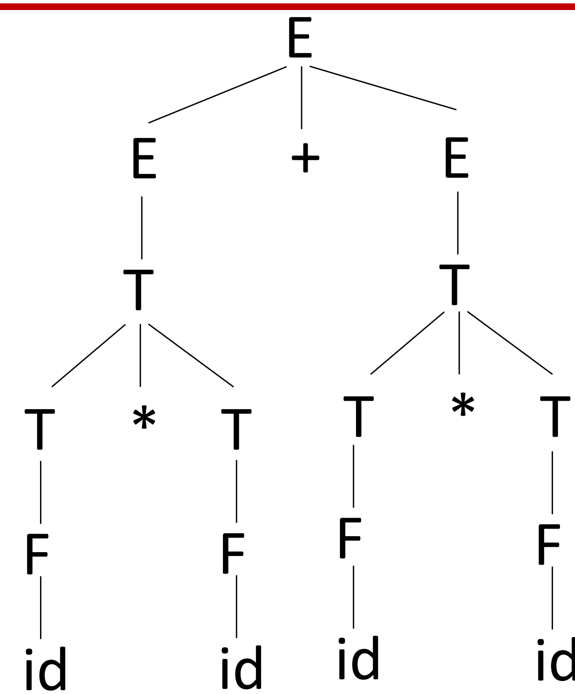


Remove Ambiguity (cont.)

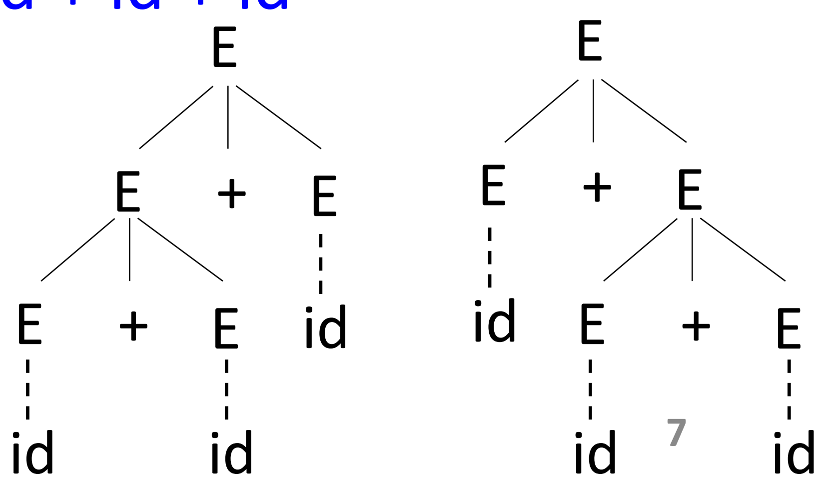
$E \rightarrow E{*}E \mid E{+}E \mid (E) \mid \mathrm{id}$

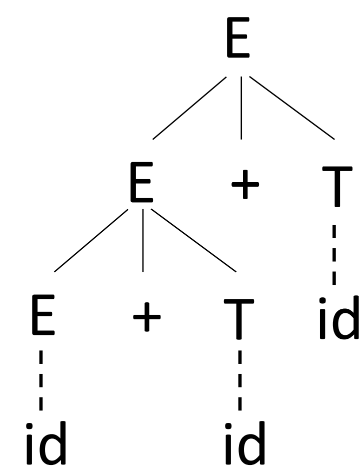
$E \rightarrow E + E \mid T$
$T \rightarrow T * T \mid F$
$F \rightarrow (E) \mid \mathrm{id}$

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \mathrm{id}$



- String id * id + id * id can result in
  - Meaning 1: (id * id) + (id * id) ✓
  - Meaning 2: id * (id + (id * id))
  - Meaning 3: id * ((id + id) * id)
- String id + id + id



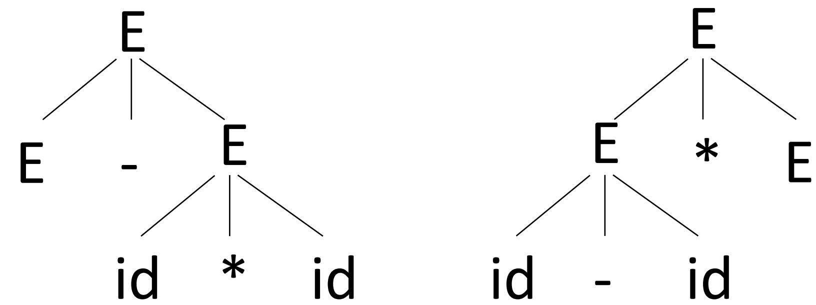


Example

$E \to E * E \mid E - E \mid id$

//precedence: * - same

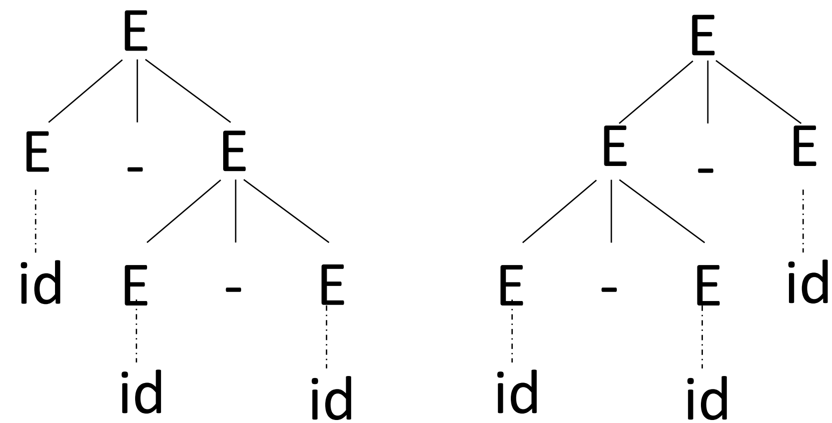
id - id * id



$E \to E - E \mid T$
$T \to T * T \mid F$
$F \to id$

//precedence: * is high

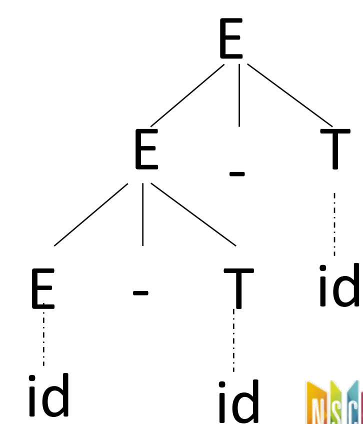
id - id - id



$E \to E - T \mid T$
$T \to T * F \mid F$
$F \to id$

//precedence: * is high
//associativity: - is left

id - id - id



Grammar → Parser[文法到分析器]

- What exactly is **parsing**, or syntax analysis?[语法分析]
  - To process an input string for a given grammar,
  - and compose the derivation if the string is in the language
  - Two subtasks
    - determine if string can be derived from grammar or not
    - build a representation of derivation and pass to next phase
- What is the best representation of derivation?[推导表示]
  - Can be a parse tree or an abstract syntax tree
- An abstract syntax tree is[抽象语法树]
  - Abbreviated representation of a parse tree
  - Drops some details without compromising meaning
    - some terminal symbols that no longer contribute to semantics are dropped (e.g. parentheses)
    - internal nodes may contain terminal symbols

Example: Abstract Syntax Tree

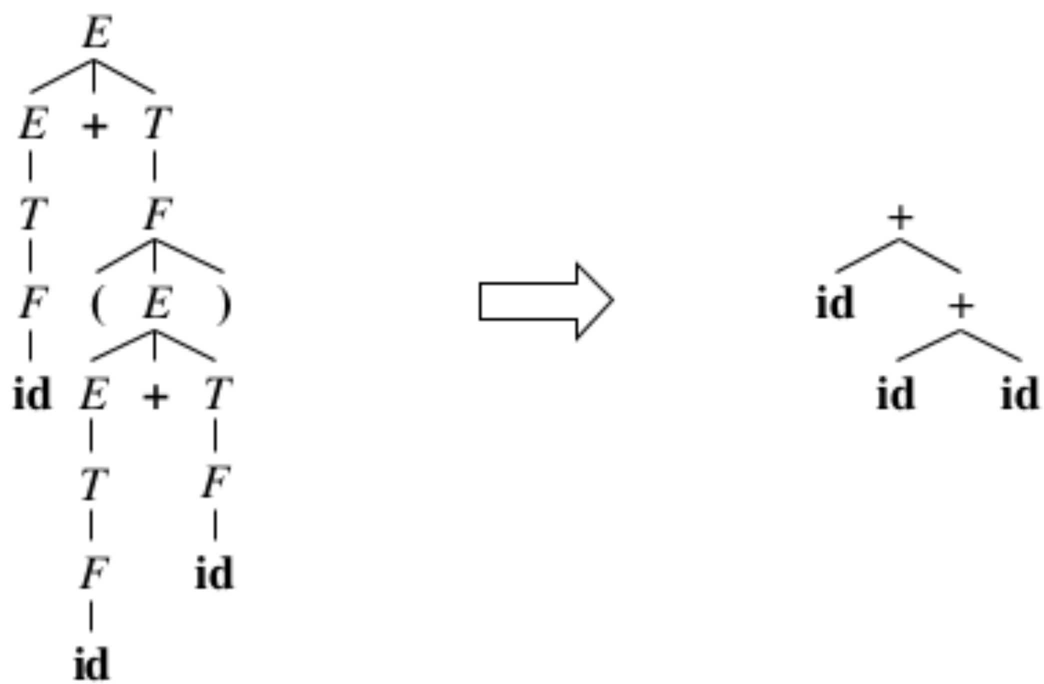
- AST: condensed form of parse tree
  - Operators and keywords do not appear as leaves (e.g., +)
  - Chains of single productions are collapsed (e.g., E -> T)

G:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \mathrm{id}$

Input:

id + (id + id)



parse tree　　　　AST

Summary of CFG[小结]

- Compilers specify program structure using CFG
  - Most programming languages are not context free
  - Context sensitive analysis can easily be separated out to semantic analysis phase

- A parser uses CFG to
  - … group lexical tokens to form expressions, statements, etc
  - ... answer if an input *str* ∈ L(G)
  - ... and build a parse tree
    - ... or build an AST instead
  - ... and pass it to the rest of compiler
    - ... or give an error message stating that *str* is invalid

Parser Implementation: Yacc + lex

parser.y

```
%{
 #include <ctype.h>
 #include <stdio.h>
 #define YYSTYPE double /* double type for Yacc stack */
%}
%token NUMBER

%left '+' '-'
%left '*' '/'

%%

lines : lines expr '\n' { printf("= %g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
expr : expr '+' expr { $$ = $1 + $3; }
     | expr '-' expr { $$ = $1 - $3; }
     | expr '*' expr { $$ = $1 * $3; }
     | expr '/' expr { $$ = $1 / $3; }
     | '(' expr ')' { $$ = $2; }
     | NUMBER
     ;

%%

/*
int yylex() {
    int c;
    while ((c = getchar()) == ' ') ;
    if ((c == '.') || isdigit(c)) {
        ungetc(c, stdin);
        scanf("%lf", &yylval);
        return NUMBER;
    }
    return c;
}
*/

int main() {
    if (yyparse() != 0)
        fprintf(stderr, "Abnormal exit\n");
    return 0;
}

int yyerror(char *s) {
    fprintf(stderr, "Error: %s\n", s);
}
```

E → E+E|E-E|E*E|E/E|(E)|num

lexer.l

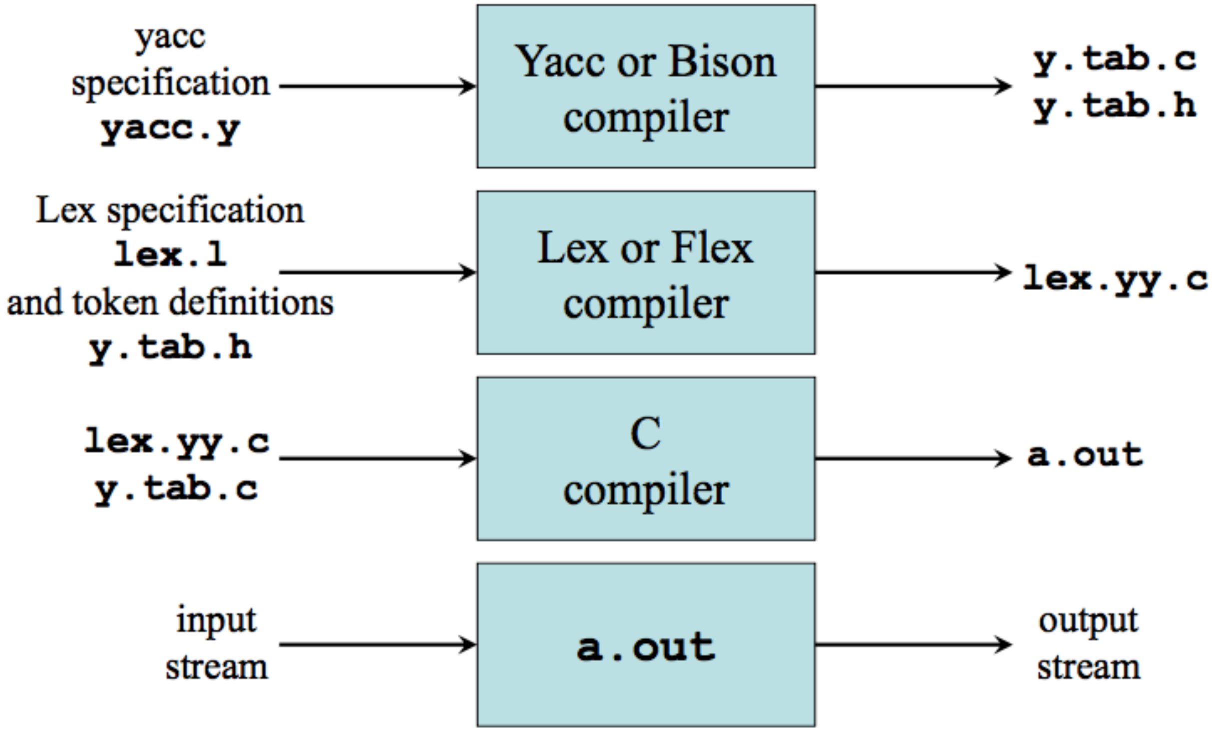
```
%{
 #define YYSTYPE double
 #include "y.tab.h"
 extern double yylval;
%}
number [0-9]+\.?|[0-9]*\.[0-9]+

%%

[ ]          { /* skip blanks */ }
{number}     { sscanf(yytext, "%lf", &yylval);
                  return NUMBER; }
\n|.         { return yytext[0]; }

%%

int yywrap(void) {
  return 1;
}
```

Generated by Yacc

Defined in y.tab.c

Yacc + Lex

- Lex was designed to produce lexical analyzers that could be used with Yacc
- Yacc generates a parser in y.tab.c and a header y.tab.h
- Lex includes the header and utilizes token definitions
- Yacc calls yylex() to obtain tokens



Example: Yacc + Lex (cont.)

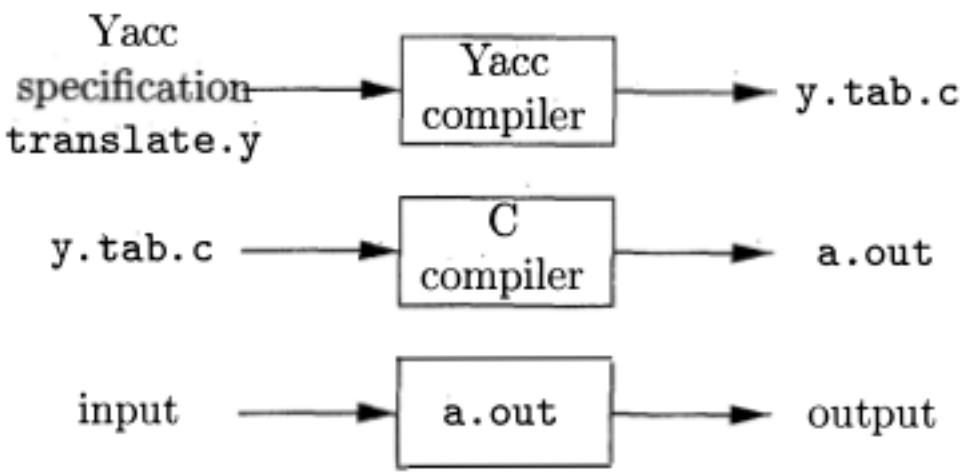
- Compile
  - $yacc -d *parser*.y
  - $lex *lexer*.l
  - $clang -o *test* y.tab.c lex.yy.c

- Run
  - $./*test* < exprs.txt

```
1 + 5
1 * 2 + 10
10 - 2 -3
```

Yacc Overview

- Yacc is an LALR(1) parser generator
  - YACC: Yet Another Compiler-Compiler
  - Parse a language described by a context-free grammar (**CFG**)
  - Yacc constructs an **LALR(1)** table
- Available as a command on the UNIX system
  - Bison: free GNU project alternative to Yacc



Yacc Specification

- **Definitions** section[定义]:
  - C declarations within %{ %}
  - Token declarations
- **Rules** section[规则]:
  - Each rule consists of a grammar production and the associated semantic action
- **Subroutines** section[辅助函数]:
  - User-defined auxiliary functions

```
%{
  #include …
%}
%token NUM VAR
- - - - - - - - - - - - - - - - - -
%%
production { semantic action }
…
- - - - - - - - - - - - - - - - - -
%%
…
```

Write a Grammar in Yacc

- A set of productions <head> → <body>$_1$ | … | <body>$_n$ would be written in YACC as:

  <head> : <body>$_1$ { <semantic action>$_1$ }
  …
  : <body>$_n$ { <semantic action>$_n$ }
  ;

- Usages
  - Tokens that are single characters can be used directly within productions, e.g. '+'
  - Named tokens must be declared first in the declaration part using %token *TokenName*

Write a Grammar in Yacc (cont.)

- Semantic actions may refer to values of the synthesized attributes of terminals and non-terminals in a production:

  $$X : Y_1\ Y_2\ Y_3 \ldots Y_n \ \{\ action\ \}$$

  - $$ refers to the value of the attribute of X (non-terminal)
  - $i refers to the value of the attribute of Y_i (terminal or non-terminal)
  - Normally the semantic action computes a value for $$ using $i's

- Example: $E \rightarrow E + T \mid T$

```
expr : expr '+' term { $$ = $1 + $2 }
     | term
     ;
```

default action: {$$ = $1 }

Example: $E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid (E) \mid num$

```
%{
 #include <ctype.h>
 #include <stdio.h>
 #define YYSTYPE double /* double type for Yacc stack */
%}
%token NUMBER

%left '+' '-'
%left '*' '/'

%%

lines : lines expr '\n' { printf("= %g\n", $2); }
      | lines '\n'
      | /* empty */
      ;
expr : expr '+' expr { $$ = $1 + $3; }
     | expr '-' expr { $$ = $1 - $3; }
     | expr '*' expr { $$ = $1 * $3; }
     | expr '/' expr { $$ = $1 / $3; }
     | '(' expr ')' { $$ = $2; }
     | NUMBER
     ;
```

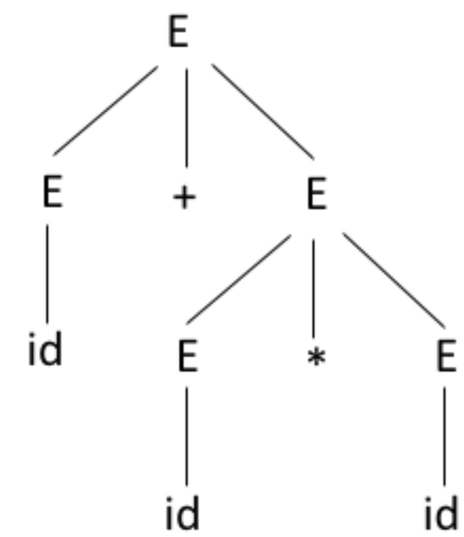
Can we remove those two lines?

Allow to evaluate a sequence of expressions, one to a line

ε

Parser Types[分析器类型]

- **Grammar** is used to derive string or construct **parser**
- Most compilers use either **top-down** or **bottom-up** parsers
- Top-down parsing[自顶向下分析]
  - Starts from root and expands into leaves
    - Tries to expand start symbol to input string
    - Finds leftmost derivation[最左推导]
  - In each step
    - Which non-terminal to replace?
    - Which production of the non-terminal to use?
  - Parser code structure closely mimics grammar
    - Amenable to implementation by hand
    - Automated tools exist to convert to code (e.g. ANTLR)



Parser Types (cont.)

- Bottom-up parser[自底向上分析]
  - Starts at leaves and builds up to root
    - Tries to reduce the input string to the start symbol
    - Finds reverse order of the rightmost derivation[最右推导的逆 → 最左归约, 也称为**规范归约**]
  - Parser code structure nothing like grammar
    - Very difficult to implement by hand
    - Automated tools exist to convert to code (e.g. Yacc, Bison)
    - LL ⊂ LR (Bottom-up works for a larger class of grammars)
- Top-down vs. bottom-up[对比]
  - Top-down: easier to understand and implement manually
    - E.g., ANTLR
  - Bottom-up: more powerful, can be implemented automatically
    - E.g., YACC/Bison

Example

- Consider a CFG grammar G

  $S \rightarrow AB$ $A \rightarrow aC$ $B \rightarrow bD$ $D \rightarrow d$ $C \rightarrow c$

- This language has only one sentence: L(G) = {acbd}

Top-down (leftmost derivation)

$S \Rightarrow AB$ (1)
$\Rightarrow aCB$ (2)
$\Rightarrow acB$ (3)
$\Rightarrow acbD$ (4)
$\Rightarrow acbd$ (5)

Bottom-up (reverse of rightmost derivation)

$S \Rightarrow AB$ (5)
$\Rightarrow AbD$ (4)
$\Rightarrow Abd$ (3)
$\Rightarrow aCbd$ (2)
$\Rightarrow acbd$ (1)

Example

- Consider a CFG grammar G

  $S \to AB$ $A \to aC$ $B \to bD$ $D \to d$ $C \to c$

- This language has only one sentence: L(G) = {acbd}

Top-down (leftmost derivation)

$S \Rightarrow AB$ (1)
$\Rightarrow aCB$ (2)
$\Rightarrow acB$ (3)
$\Rightarrow acbD$ (4)
$\Rightarrow acbd$ (5)

S

Bottom-up (reverse of rightmost derivation)

$S \Rightarrow AB$ (5)
$\Rightarrow AbD$ (4)
$\Rightarrow Abd$ (3)
$\Rightarrow aCbd$ (2)
$\Rightarrow acbd$ (1)

Example

- Consider a CFG grammar G

  S→AB A→aC B→bD D→d C→c

- This language has only one sentence: L(G) = {acbd}

Top-down (leftmost derivation)

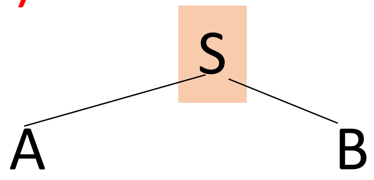
$S \Rightarrow AB$ (1)

$\Rightarrow aCB$ (2)

$\Rightarrow acB$ (3)

$\Rightarrow acbD$ (4)

$\Rightarrow acbd$ (5)



Bottom-up (reverse of rightmost derivation)

$S \Rightarrow AB$ (5)

$\Rightarrow AbD$ (4)

$\Rightarrow Abd$ (3)

$\Rightarrow aCbd$ (2)

$\Rightarrow acbd$ (1)

Example

- Consider a CFG grammar G

  S→AB A→aC B→bD D→d C→c

- This language has only one sentence: L(G) = {acbd}

Top-down (leftmost derivation)

$S \Rightarrow AB$ (1)

$\Rightarrow aCB$ (2)

$\Rightarrow acB$ (3)

$\Rightarrow acbD$ (4)

$\Rightarrow acbd$ (5)

Bottom-up (reverse of rightmost derivation)

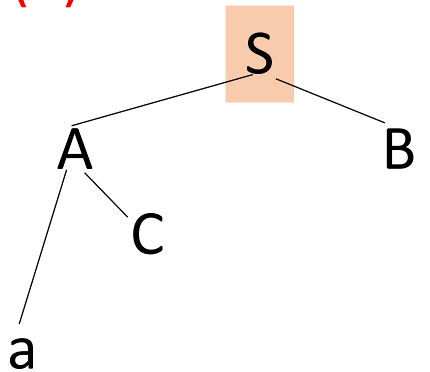
$S \Rightarrow AB$ (5)

$\Rightarrow AbD$ (4)

$\Rightarrow Abd$ (3)

$\Rightarrow aCbd$ (2)

$\Rightarrow acbd$ (1)



Example

- Consider a CFG grammar G

  S→AB   A→aC   B→bD   D→d   C→c

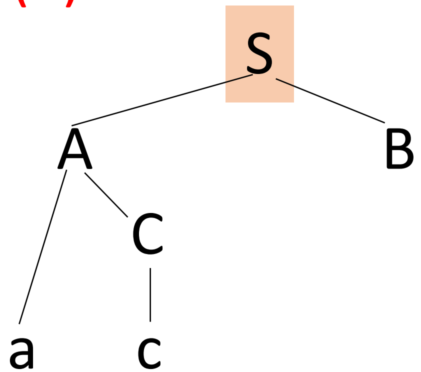
- This language has only one sentence: L(G) = {acbd}

Top-down (leftmost derivation)

S ⇒ AB (1)
⇒ aCB (2)
⇒ acB (3)
⇒ acbD (4)
⇒ acbd (5)

Bottom-up (reverse of rightmost derivation)

S ⇒ AB (5)
⇒ AbD (4)
⇒ Abd (3)
⇒ aCbd (2)
⇒ acbd (1)



中山大學 SUN YAT-SEN UNIVERSITY

Example

- Consider a CFG grammar G

  S→AB A→aC B→bD D→d C→c

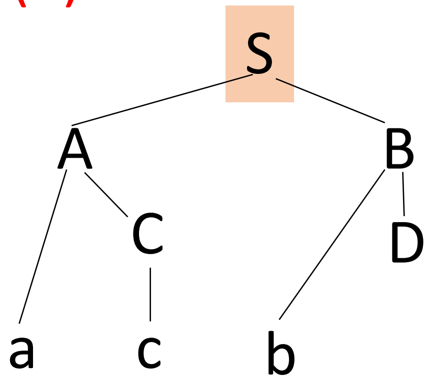
- This language has only one sentence: L(G) = {acbd}

Top-down (leftmost derivation)

$S \Rightarrow AB$ (1)
$\Rightarrow aCB$ (2)
$\Rightarrow acB$ (3)
$\Rightarrow acbD$ (4)
$\Rightarrow acbd$ (5)

Bottom-up (reverse of rightmost derivation)

$S \Rightarrow AB$ (5)
$\Rightarrow AbD$ (4)
$\Rightarrow Abd$ (3)
$\Rightarrow aCbd$ (2)
$\Rightarrow acbd$ (1)



Example

- Consider a CFG grammar G

  S→AB   A→aC   B→bD   D→d   C→c

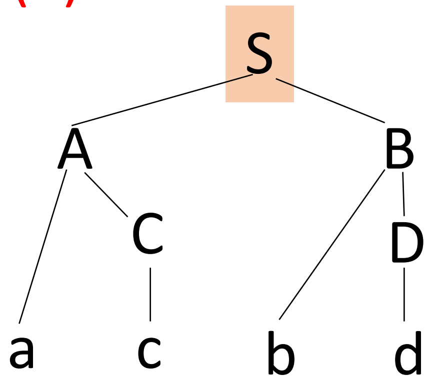
- This language has only one sentence: L(G) = {acbd}

Top-down (leftmost derivation)

$S \Rightarrow AB$ (1)
$\Rightarrow aCB$ (2)
$\Rightarrow acB$ (3)
$\Rightarrow acbD$ (4)
$\Rightarrow acbd$ (5)

Bottom-up (reverse of rightmost derivation)

$S \Rightarrow AB$ (5)
$\Rightarrow AbD$ (4)
$\Rightarrow Abd$ (3)
$\Rightarrow aCbd$ (2)
$\Rightarrow acbd$ (1)



Example

- Consider a CFG grammar G

  S→AB A→aC B→bD D→d C→c

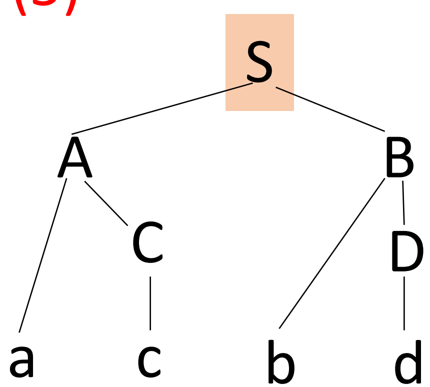
- This language has only one sentence: L(G) = {acbd}

Top-down (leftmost derivation)

$S \Rightarrow AB$ (1)
$\Rightarrow aCB$ (2)
$\Rightarrow acB$ (3)
$\Rightarrow acbD$ (4)
$\Rightarrow acbd$ (5)

Bottom-up (reverse of rightmost derivation)

$S \Rightarrow AB$ (5)
$\Rightarrow AbD$ (4)
$\Rightarrow Abd$ (3)
$\Rightarrow aCbd$ (2)
$\Rightarrow acbd$ (1)



a c b d

Example

- Consider a CFG grammar G

  $S \rightarrow AB$ $A \rightarrow aC$ $B \rightarrow bD$ $D \rightarrow d$ $C \rightarrow c$

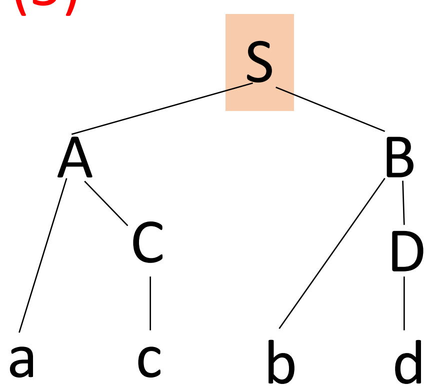
- This language has only one sentence: L(G) = {acbd}

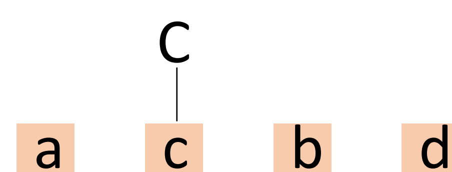
Top-down (leftmost derivation)

$S \Rightarrow AB$ (1)
$\Rightarrow aCB$ (2)
$\Rightarrow acB$ (3)
$\Rightarrow acbD$ (4)
$\Rightarrow acbd$ (5)

Bottom-up (reverse of rightmost derivation)

$S \Rightarrow AB$ (5)
$\Rightarrow AbD$ (4)
$\Rightarrow Abd$ (3)
$\Rightarrow aCbd$ (2)
$\Rightarrow acbd$ (1)





Example

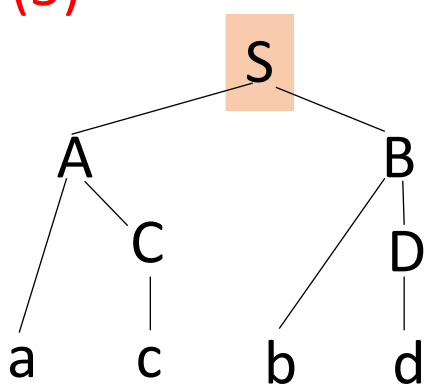
- Consider a CFG grammar G

  $S \rightarrow AB$ $A \rightarrow aC$ $B \rightarrow bD$ $D \rightarrow d$ $C \rightarrow c$

- This language has only one sentence: L(G) = {acbd}

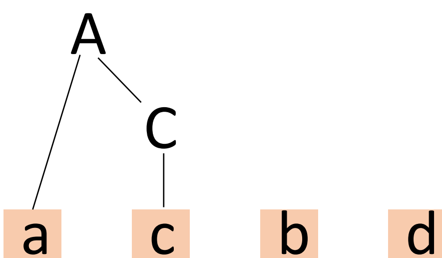
Top-down (leftmost derivation)

$S \Rightarrow AB$ (1)
$\Rightarrow aCB$ (2)
$\Rightarrow acB$ (3)
$\Rightarrow acbD$ (4)
$\Rightarrow acbd$ (5)



Bottom-up (reverse of rightmost derivation)

$S \Rightarrow AB$ (5)
$\Rightarrow AbD$ (4)
$\Rightarrow Abd$ (3)
$\Rightarrow aCbd$ (2)
$\Rightarrow acbd$ (1)



Example

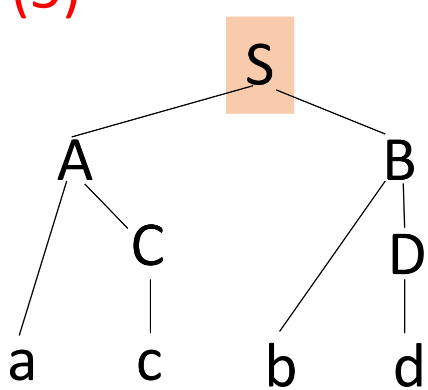
- Consider a CFG grammar G

  S→AB A→aC B→bD D→d C→c

- This language has only one sentence: L(G) = {acbd}

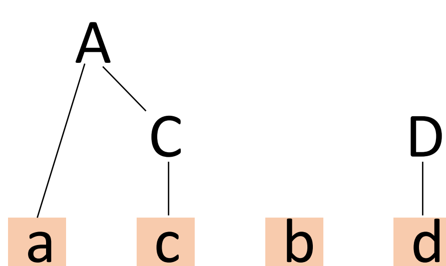
Top-down (leftmost derivation)

$S \Rightarrow AB$ (1)
$\Rightarrow aCB$ (2)
$\Rightarrow acB$ (3)
$\Rightarrow acbD$ (4)
$\Rightarrow acbd$ (5)



Bottom-up (reverse of rightmost derivation)

$S \Rightarrow AB$ (5)
$\Rightarrow AbD$ (4)
$\Rightarrow Abd$ (3)
$\Rightarrow aCbd$ (2)
$\Rightarrow acbd$ (1)



Example

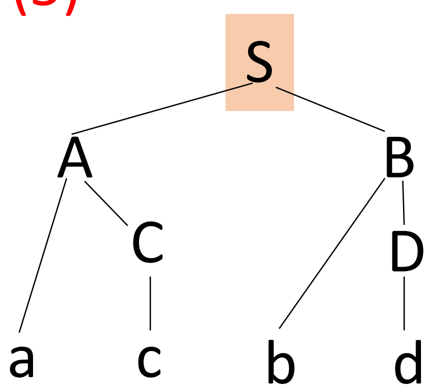
- Consider a CFG grammar G

  S→AB　A→aC　B→bD　D→d　C→c

- This language has only one sentence: L(G) = {acbd}

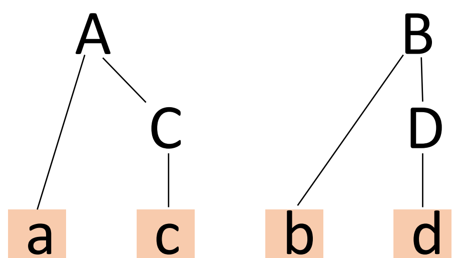
Top-down (leftmost derivation)

$S \Rightarrow AB$ (1)
$\Rightarrow aCB$ (2)
$\Rightarrow acB$ (3)
$\Rightarrow acbD$ (4)
$\Rightarrow acbd$ (5)



Bottom-up (reverse of rightmost derivation)

$S \Rightarrow AB$ (5)
$\Rightarrow AbD$ (4)
$\Rightarrow Abd$ (3)
$\Rightarrow aCbd$ (2)
$\Rightarrow acbd$ (1)



Example

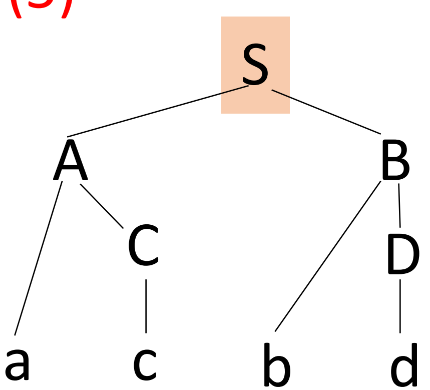
- Consider a CFG grammar G

  S→AB   A→aC   B→bD   D→d   C→c

- This language has only one sentence: L(G) = {acbd}

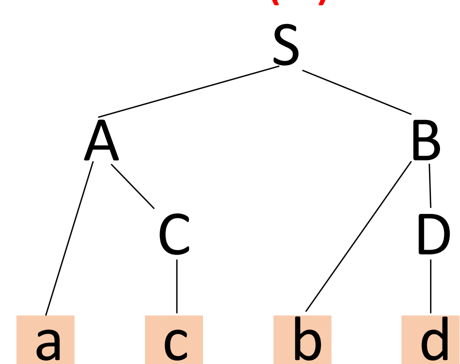
Top-down (leftmost derivation)

S ⇒ AB (1)
⇒ aCB (2)
⇒ acB (3)
⇒ acbD (4)
⇒ acbd (5)



Bottom-up (reverse of rightmost derivation)

S ⇒ AB (5)
⇒ AbD (4)
⇒ Abd (3)
⇒ aCbd (2)
⇒ acbd (1)



Preview: Bottom-up Parsing[自低向上]

- Consider a CFG grammar G

$S \rightarrow AB$ $\quad A \rightarrow aC$ $\quad B \rightarrow bD$ $\quad D \rightarrow d$ $\quad C \rightarrow c$

Bottom-up (reverse of rightmost derivation)

$S \Rightarrow AB$ (5)

$\Rightarrow AbD$ (4)

$\Rightarrow Abd$ (3)

$\Rightarrow aCbd$ (2)

$\Rightarrow acbd$ (1)

Preview: Bottom-up Parsing[自低向上]

- Consider a CFG grammar G

S→AB A→aC B→bD D→d C→c

Stack	Input	Action
$	acbd$	Shift
$a	cbd$	Shift
$ac	bd$	Reduce
$aC	bd$	Reduce
$A	bd$	Shift
$Ab	d$	Shift
$Abd	$	Reduce
$AbD	$	Reduce
$AB	$	Reduce
$S	$	SUCCESS!

Bottom-up (reverse of rightmost derivation)

S ⇒ AB (5)

⇒ AbD (4)

⇒ Abd (3)

⇒ aCbd (2)

⇒ acbd (1)

Preview: Bottom-up Parsing[自低向上]

- Consider a CFG grammar G

S→AB   A→aC   B→bD   D→d   C→c

Stack	Input	Action
$	acbd$	Shift
$a	cbd$	Shift
$ac	bd$	Reduce
$aC	bd$	Reduce
$A	bd$	Shift
$Ab	d$	Shift
$Abd	$	Reduce
$AbD	$	Reduce
$AB	$	Reduce
$S	$	SUCCESS!

Bottom-up (reverse of rightmost derivation)

S ⇒ AB (5)

⇒ AbD (4)

⇒ Abd (3)

⇒ aCbd (2)

⇒ acbd (1)

SUN YAT-SEN UNIVERSITY 中山大學

a c b d

NSCC GZ

Preview: Bottom-up Parsing[自低向上]

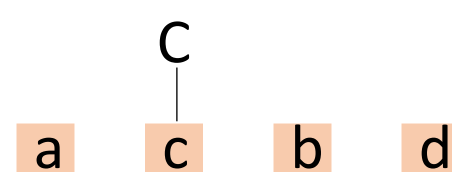
- Consider a CFG grammar G

  S→AB A→aC B→bD D→d C→c

Stack	Input	Action
\$	acbd\$	Shift
\$a	cbd\$	Shift
\$ac	bd\$	Reduce
\$aC	bd\$	Reduce
\$A	bd\$	Shift
\$Ab	d\$	Shift
\$Abd	\$	Reduce
\$AbD	\$	Reduce
\$AB	\$	Reduce
\$S	\$	SUCCESS!

Bottom-up (reverse of rightmost derivation)

S ⇒ AB (5)
⇒ AbD (4)
⇒ Abd (3)
⇒ aCbd (2)
⇒ acbd (1)



Preview: Bottom-up Parsing[自低向上]

- Consider a CFG grammar G

S→AB　　A→aC　　B→bD　　D→d　　C→c

Stack	Input	Action
$	acbd$	Shift
$a	cbd$	Shift
$ac	bd$	Reduce
$aC	bd$	Reduce
$A	bd$	Shift
$Ab	d$	Shift
$Abd	$	Reduce
$AbD	$	Reduce
$AB	$	Reduce
$S	$	SUCCESS!

Bottom-up (reverse of rightmost derivation)

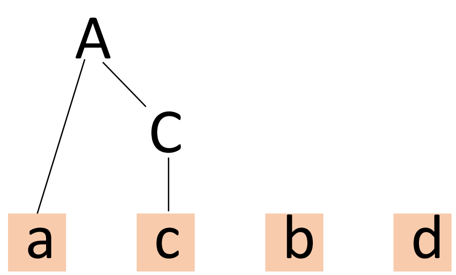
$S \Rightarrow AB$ (5)

$\Rightarrow AbD$ (4)

$\Rightarrow Abd$ (3)

$\Rightarrow aCbd$ (2)

$\Rightarrow acbd$ (1)



Preview: Bottom-up Parsing[自低向上]

- Consider a CFG grammar G

S→AB   A→aC   B→bD   D→d   C→c

Stack	Input	Action
$	acbd$	Shift
$a	cbd$	Shift
$ac	bd$	Reduce
$aC	bd$	Reduce
$A	bd$	Shift
$Ab	d$	Shift
$Abd	$	Reduce
$AbD	$	Reduce
$AB	$	Reduce
$S	$	SUCCESS!

Bottom-up (reverse of rightmost derivation)

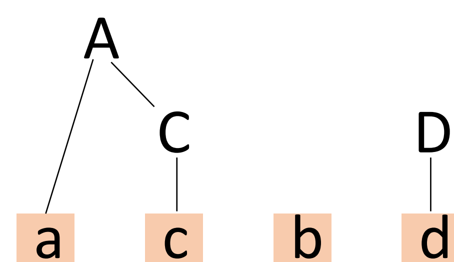
$S \Rightarrow AB$ (5)

$\Rightarrow AbD$ (4)

$\Rightarrow Abd$ (3)

$\Rightarrow aCbd$ (2)

$\Rightarrow acbd$ (1)



Preview: Bottom-up Parsing[自低向上]

- Consider a CFG grammar G

  S→AB A→aC B→bD D→d C→c

Stack	Input	Action
$	acbd$	Shift
$a	cbd$	Shift
$ac	bd$	Reduce
$aC	bd$	Reduce
$A	bd$	Shift
$Ab	d$	Shift
$Abd	$	Reduce
$AbD	$	Reduce
$AB	$	Reduce
$S	$	SUCCESS!

Bottom-up (reverse of rightmost derivation)

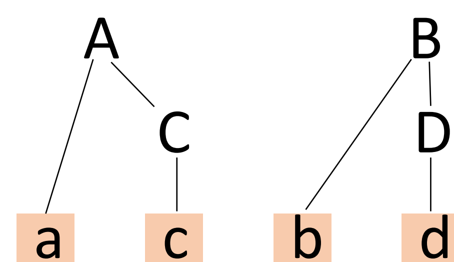
$S \Rightarrow AB$ (5)

$\Rightarrow AbD$ (4)

$\Rightarrow Abd$ (3)

$\Rightarrow aCbd$ (2)

$\Rightarrow acbd$ (1)



Preview: Bottom-up Parsing[自低向上]

- Consider a CFG grammar G

$S \rightarrow AB$ $A \rightarrow aC$ $B \rightarrow bD$ $D \rightarrow d$ $C \rightarrow c$

Stack	Input	Action
$	acbd$	Shift
$a	cbd$	Shift
$ac	bd$	Reduce
$aC	bd$	Reduce
$A	bd$	Shift
$Ab	d$	Shift
$Abd	$	Reduce
$AbD	$	Reduce
$AB	$	Reduce
$S	$	SUCCESS!

Bottom-up (reverse of rightmost derivation)

$S \Rightarrow AB$ (5)

$\Rightarrow AbD$ (4)

$\Rightarrow Abd$ (3)

$\Rightarrow aCbd$ (2)

$\Rightarrow acbd$ (1)